Introduction à Matlab

L'objectif de ces quelques pages est d'apporter une aide pour débuter avec Matlab, en particulier pour mettre en œuvre ensuite des méthodes numériques. Pour aller plus loin, profitez de l'efficacité de l'aide de Matlab.

MATLAB (MATrix LABoratory) est un logiciel de calcul numérique ; les matrices (à coefficients réels ou complexes) en sont l'outil de base.
Comme les langages de programmation tels que Pascal, Fortran ou C, il possède les structures algorithmiques habituelles comme `for`, `while`, `if` .... Il comporte également
– des opérations et des fonctions de calcul numérique ;
– un environnement graphique permettant la visualisation bi- ou tri-dimensionnelle de données numériques.

1. Pour commencer

1.1. L'environnement.

Parmi les fenêtres de l'environnement MATLAB, notons celles intitulées :

1. `Command Window` : il s'agit de la fenêtre dans laquelle on tape des commandes Matlab ;
2. `Workspace` : il s'agit de la fenêtre où figure la liste des variables actuellement en mémoire ;
3. `Current Directory` : il s'agit de la fenêtre contenant la liste des fichiers et répertoires présents dans le répertoire dont le chemin est précisé en haut de la fenêtre. Il est souhaitable de toujours commencer par se placer dans son répertoire de travail, puis éventuellement dans des sous-dossiers pour chacun des TPs.

On verra par la suite d'autres fenêtres telles que l'éditeur de fichiers ou les figures.

1.2. L'aide.

Matlab dispose d'une aide (en anglais) sous plusieurs formes :

➠ En tapant `help nom_de_commande` dans la fenêtre de commande, on obtient une information succincte concernant le fonctionnement de la commande `nom_de_commande`. On peut également ouvrir la fenêtre d'aide avec `helpwin nom_de_commande` ou en appuyant sur `F1` (le curseur étant sur la fonction recherchée).
➠ En activant l'aide (menu déroulant ou `helpwin`), on pourra aller dans :

- `Function Browser` si on connaît le nom de la commande (directement avec `Maj+F1`) ;
- `Contents > MATLAB`, où un parcours balisé devrait vous fournir les informations cherchées. On pourra s'intéresser particulièrement à :
  - `Getting Started` pour de nombreuses informations de base,
  - `Functions` où figurent les fonctions classées par catégories,
  - `Demos` pour voir sur quelques exemples certains aspects de Matlab ;
  - `User Guide` où il y a plusieurs rubriques intéressantes comme : `Mathematics` pour les manipulations mathématiques possibles ; `Programming Fundamentals > Basic Program Components` pour quelques notions de bases utilisées dans un programme Matlab (en particulier l'algorithmique) ; `Programming Fundamentals > Functions and Scripts` pour l'utilisation des M-Files ; `Graphics > Plots and Plotting Tools` et `Graphics > Basic Plotting Commands` pour le graphisme.

1.3. **Informations utiles.**

- On retrouve les commandes antérieures à l'aide des flèches `↑` et `↓`.
- L'interruption d'un programme se fait par `ctrl+c`.
- Des constantes existent en Matlab : `eps` valant $2^{-52} \approx 2.10^{-16}$, `pi` une valeur (approchée) de π.

1.4. **A savoir pour programmer.**

- Matlab distingue les majuscules et les minuscules.
- Il n'y a pas de déclaration de variables ; les variables utilisées dans une suite de commandes Matlab peuvent même changer de type en cours de "programme".
  Chaque fois qu'une variable est créée par une affectation (*cf* ci-dessous), dans la fenêtre de commande ou dans un script (*cf* 2.1.), elle est "stockée" dans le `Workspace`.
  Pour faire afficher la liste des variables stockées, taper `whos`. Pour réinitialiser (déstocker) une liste de ces variables, faire : `clear variable1, variable2, variable3`
  Pour toutes nettoyer la mémoire (déstocker tous les variables) taper `clear` ou `clear all`.
- L'affectation se fait par l'intermédiaire du symbole `=` :
  `nomvariable = <expression>`, où le résultat de l'*expression* peut être un scalaire, une matrice, une fonction...
- En Matlab, les séparateurs de commandes sont : le saut de ligne (touche entrée), la virgule ou le point virgule. L'absence du symbole `;` à la suite d'une instruction d'affectation provoque l'affichage à l'écran du nom de la variable et de son contenu.
  Une commande peut être poursuivie sur la ligne suivante en plaçant 3 points consécutifs à la fin de la ligne.
- Dans le cas d'une division par 0 par exemple, Matlab donne un résultat : le résultat de $1/0$ est `Inf`, celui de $-1/0$ est `-Inf`, celui de $0/0$ est `NaN` (Not a Number). Matlab pourra même poursuivre des calculs avec de tels résultats.

➠ Pour produire un affichage qui n'est pas celui par défaut on peut utiliser les commandes `display`, éventuellement en combinaison avec `sprintf`, ou `fprintf` sans oublier `'\n'` pour la passage à la ligne.

Exemples :

`display('Question 1)')` peut être utilisé pour marquer le début d'une question.

`display(sprintf('%u < %.2f',1,pi))`
`fprintf('%u < %.2f\n',1,pi)`
} sont identiques et affichent `1 < 3.14`.

`title(sprintf('Le graphe pour k=%.1f',k))` permet de mettre le titre d'une figure en fonction d'un paramètre `k`.

2. Stockage d'une suite de commandes Matlab

Il est possible de regrouper une suite de commandes dans des fichiers appelés `M-files` ; leur nom comporte l'extension `.m`.

Pour activer l'éditeur de texte, on pourra aller dans `File/New/M-file` pour créer un nouveau fichier ou cliquer sur le nom du fichier à ouvrir dans la fenêtre `Current Directory`. Ne pas oublier de faire une sauvegarde pour prendre en enumie les dernières modifications avant chaque exécution de la suite de commandes en cours d'édition (ou faire `Debug/Save and Run` pour les scripts seulement (voir plus bas)).

Dans les M-files, ce qui suit sur une ligne le caractère `%`, est un commentaire.

Les commentaires placés en début de fichier sont accessibles par la commande `help` suivie du nom du fichier.

Il existe deux types de `M-files` : les scripts et les fonctions.

2.1. **Les scripts.**

Il s'agit de programmes contenant une suite de commandes travaillant sur les variables du `Workspace` et qu'on exécute en tapant le nom du fichier dans la fenêtre de commande (ou avec `Debug/Save and Run`, ou avec la touche `F5`).

Afin d'éviter des risques d'interférence, lors de l'exécution d'un script, entre les variables du script et celles présentes dans le `Workspace` avant l'appel, on placera souvent l'instruction `clear` en début de script.

2.2. **Les fonctions.**

Il s'agit de sous-programmes classiques.

Les fonctions comportent un en-tête de la forme :

```
function[s1, s2, ...] = nomfonction(e1, e2, ...);
```

où *s1, s2,...* (respectivement *e1, e2,...*) représente les valeurs de sortie (resp. les paramètres d'entrée) de la fonction. Ces paramètres (sortie et entrée) sont séparés par des virgules.

Le *nom du fichier* contenant une fonction est nécessairement `nomfonction.m`

Les paramètres de l'entête sont formels et les autres variables intervenant dans la partie instructions de la fonction sont locales (sauf mention contraire) ; ils n'ont pas de signification en dehors de la fonction, ni aucun rapport avec une éventuelle variable de même nom dans le `Workspace`.

On appelle une fonction, dans un script ou dans la fenêtre de commande, par l'instruction :

```
[S1, S2] = nomfonction(E1, E2, ...);
```

pour obtenir les deux première valeurs de sortie.
Et pour obtenir seulement la première valeur de sortie on peut faire :

```
S1 = nomfonction(E1, E2, ...);
```

où on a remplacé les paramètres formels de l'entête par les paramètres effectifs.

Lors d'un appel, la valeur des paramètres effectifs d'entrée (s'ils ne sont pas également paramètres effectifs de sortie) n'est pas modifiée (même si dans l'entête de la fonction, un paramètre formel de sortie porte le même nom qu'un paramètre formel d'entrée) ; ils peuvent figurer sous forme d'une constante, d'une variable du `Workspace` ou d'une expression.

Les paramètres effectifs de sortie sont des variables ; s'il y a moins de paramètres effectifs de sortie que de paramètres formels de sortie, les paramètres effectifs renvoyés correspondent aux premiers paramètres formels de l'entête.

Il est possible de placer plusieurs fonctions dans un même fichier ; toutefois, seule la première est "visible" de l'extérieur (les autres ne pouvant qu'être appelées entre elles). Il est donc préférable de se limiter à une fonction par fichier.

Exemple :

La fonction du fichier `somProd.m` :

```
function [s,p] = somProd(a,b);
    s = a+b;
    p = a*b;
```

Un script appelant la fonction :

```
clear
u = sqrt(2); v = sqrt(3);
[somme, prod] = somProd(u,v)
```

Rappelons que c'est l'outil (numérique) de base de Matlab.

Certaines manipulations sur des tableaux de valeurs peuvent être traitées de façon simple et efficace (en temps) en utilisant les possibilités de Matlab comme celles présentées dans cette partie. En particulier, ces outils Matlab permettent, dans certains cas, d'éviter l'utilisation des boucles "classiques".

3.1. **Introduction.**

Une matrice est un tableau à 2 indices, le premier désignant une ligne et le second une colonne. Rappelons que toutes les lignes ont le même nombre de coefficients, et qu'il en est de même pour toutes les colonnes.

> En Matlab, les indices de ligne (resp. de colonne) d'une matrice varient nécessairement entre 1 et le nombre de lignes (resp. de colonnes).

Par la suite, on parlera de vecteur ligne si la matrice n'a qu'une ligne et de vecteur colonne si elle n'a qu'une colonne. Une matrice 1×1 sera considérée comme un scalaire.

Si `A` est une variable de type matrice (éventuellement un vecteur ou un scalaire), l'ensemble de ses valeurs est accessible par `A`.

La valeur du coefficient sur la ligne `i` et la colonne `j` de `A` est : `A(i,j)`.

Si `x` est un vecteur, la composante `i` de `x` (`x(1,i)` pour un vecteur ligne, `x(i,1)` pour un vecteur colonne) peut également être obtenue par `x(i)` ; ceci s'explique par le fait que Matlab numérote également les coefficients d'une matrice colonne par colonne.

3.2. **Créer une matrice.**

Matlab offre plusieurs façons de créer une matrice :

- Création d'une matrice vide : `a=[];`
- Création d'un scalaire : `s=3.1;`
- Création d'une matrice 2×2 composante par composante :
  Exemple : `a(1,1)=1.2 ; a(1,2)=4.5 ; a(2,1)=-2 ; a(2,2)=1.2e1 ;`
- Création globale :
  Exemples : `a=[1.2, 4.5 ; -2, 1.2e1];` ou sans les virgules : `a=[1.2 4.5 ; -2 1.2e1];`
  Ici, le symbole `;` ou le passage à la ligne suivante signifie le passage à la ligne suivante de la matrice `a` (ou aux lignes suivantes pour des matrices blocs).
- Création globale de quelques matrices particulières :
  `zeros` créer une matrice à coefficients 0. *Exemples* :
  - `zeros(2,4)` est la matrice nulle 2×4 ;
  - `zeros(3)` est la matrice nulle d'ordre 3.

– `zeros(size(A))` est la matrice nulle de mêmes dimensions que `A`.

`eye` créer une matrice identité.

– `eye(4)` est la matrice identité d'ordre 4 ;

– `eye(3,2)` est la matrice 3×2 contenant la matrice identité d'ordre 2 suivi d'une ligne nulle.

– `eye(size(A))` est la matrice identité de mêmes dimensions que `A`.

`ones`. créer la matrice coefficients égaux à 1.

– `ones(3,2)` est la matrice 3×2 de coefficients égaux à 1.

– `ones(2)` est la matrice 2×2 de coefficients égaux à 1.

`rand` créer une matrice *aléatoire*.

– `rand(3,2)` est une matrice 3×2 à coefficients réels choisis "aléatoirement" dans $]0, 1[$ (selon la loi uniforme).

– `rand(2)*3` est une matrice 2×2 à coefficients réels choisis "aléatoirement" dans $]0, 3[$.

`hilb(n)` créer la matrice de Hilbert de taille $n \times n$ qui est connu pour être très mal conditionnée et dont les entrés sont $\frac{1}{i+j-1}$.

$\longrightarrow$ voir l'aide pour d'autres matrices particulières. Par exemple `gallery` permet de créer différentes matrices.

➠ Création de vecteurs lignes particuliers :

`x=m:n` définit un vecteur ligne `x` égal à $(m,\ m+1, \ldots,\ m+k)$ avec k maximal tel que $k \leq n$.

– `x=1:10` définit un vecteur ligne `x` égal à $(1,\ 2,\ \ldots,\ 10)$.

– `x=0.1:10` définit un vecteur ligne `x` égal à $(0.1,\ 1.1,\ \ldots,\ 9.1)$: valeurs partant de 0.1 et $\leq$10 par pas de 1.

– `x=2:1` affecte à `x` la matrice vide `[]`.

`x=m:p:n` définit un vecteur ligne `x` égal à $(m,\ m+p,\ \ldots,\ m+kp)$ avec k maximal tel que $m + kp \leq n$.

– `x=10:-2:0` définit un vecteur ligne `x` égal à $(10,\ 8,\ \ldots,\ 0)$.

– `x=0:0.1:1.1` définit un vecteur ligne `x` égal à $(0,\ 0.1,\ \ldots,\ 1.1)$: valeurs partant de 0 et $\leq$1.1 par pas de 0.1.

`x=linspace(m,n,k)` affecte à `x` le vecteur ligne à k éléments équidistribués entre m et n : $(\mathrm{m}, \mathrm{m{+}d}, \ldots, \mathrm{n})$ où $d = \frac{n-m}{\max\{k-1,1\}}$.

– `x=linspace(-0.2,1.2,15)` affecte à `x` le vecteur ligne $(-0.2,\ -0.1,\ \ldots,\ 1.2)$ c'est-à-dire celui des points de la subdivision régulière de $[-0.2, 1.2]$ à 15 points, bornes comprises.

$\longrightarrow$ voir également la fonction `logspace` pour un espacement logarithmique.

3.3. **Construire une matrice à partir d'autres matrices.**

3.3.1. *Matrices particulières.*

`A'` est la matrice adjointe de `A` (c'est-à-dire $(A')_{i,j} = \overline{A_{j,i}}$), qui coïncide avec la matrice transposée pour une matrice à valeurs réelles.

`diag(A)`, pour `A` une matrice $n \times m$ avec $n \geq 2$, $m \geq 2$, est le vecteur colonne de coefficients `a(1,1),a(2,2),...,a(p,p)`, où $p = \min(n, m)$.
Plus généralement, voir `diag(A,k)`, où `k` est un entier, pour la "diagonale `k`".

`diag(x)`, pour `x` un vecteur, est la matrice diagonale (carrée) dont la diagonale correspond au vecteur `x`. Plus généralement, voir `diag(x,k)`.

Ainsi pour obtenir la matrice diagonale qui a la même diagonale que `A` il suffit de faire `diag(diag(A))`.

`triu(A)` (resp. `tril(A)`) est la "partie triangulaire supérieure" (resp. "inférieure") de `A`, de mêmes dimensions que `A`. Plus généralement, voir `triu(A,k)` et `tril(A,k)`.

3.3.2. *Matrices blocs (notion "généralisée").*

`[A, B]`, pour `A` une matrice $m \times n$, et `B` une matrice $m \times p$, est la matrice bloc $m \times (n+p)$ dont les n premières colonnes sont celles de `A` et les p suivantes celles de `B`. Autrement dit `[A, B]` est la matrice obtenu en mettant `B` à côté de `A`. On peut aussi utiliser `[A B]`, sans la virgule.

`[A; B]`, pour `A` une matrice $m \times p$, et `B` une matrice $n \times p$, est la matrice bloc $(m+n) \times p$ dont les m premières lignes sont celles de `A` et les n suivantes celles de `B`. Autrement dit `[A; B]` est la matrice obtenu en mettant `B` sous `A`.

`repmat(A,m,n)` permet de construire une matrice par blocs ($m \times n$ blocs de la taille de `A`) en répliquant plusieurs fois la même matrice sur les lignes et les colonnes.

3.3.3. *Sous-matrices (notion "généralisée").*

De façon générale, on note `A` une matrice $n \times m$, `x` et `y` deux vecteurs (ligne ou colonne) à respectivement p composantes entières entre 1 et n et q composantes entières entre 1 et m. Alors, `A(x,y)` désigne la matrice à p lignes et q colonnes, de coefficients :

$$(\texttt{A(x,y)})_{i,j} = \texttt{A}_{x_i,y_j}, \quad i \in \{1, \ldots, p\},\ j \in \{1, \ldots, q\}$$

De plus, `A(:,y)` (resp. `A(x,:)`) est la matrice formée des colonnes (resp. lignes) de `A` repérées par les valeurs des composantes de `y` (resp. `x`).

Le mot clé `end` en indice de `A` désigne le nombre de lignes ou de colonnes de `A` selon qu'il intervient au niveau du premier ou du second indice.

Ajoutons que les extractions de sous-matrices peuvent être affectées c'est-à-dire situées à gauche du symbole `=` (lorsque chacun des vecteurs `x` et `y` est à composantes distinctes).

Exemples :

`A(1,3)` est le coefficient ligne 1 colonne 3 de `A`.

`A(1:3,5)` est le vecteur colonne formé des coefficients des lignes 1, 2 et 3 (`1:3` est le vecteur `[1,2,3]`) de la colonne 5 de `A`.

`A(:,4)` est le vecteur colonne égal à la colonne 4 de `A`.

`A([1 3 4],:)` est la matrice formée des lignes 1, 3 et 4 de `A`.

`A([1 3 4],1:3)` est la matrice d'ordre 3 extraite de `A` formée des coefficients situés sur les lignes 1, 3 et 4 et les colonnes 1, 2 et 3.

`A(2:end-1,1)` est le vecteur colonne formé des coefficients de la première colonne de `A` à partir du deuxième et jusqu'à l'avant dernier.

`A(ones(1,3),2:3)` est la matrice formée de 3 lignes identiques égales à `A(1,2:3)`.

`B(:,[1 4 5])=A(:,1:3)` remplace les colonnes 1, 4 et 5 de `B` par les colonnes 1, 2 et 3 de `A` pour deux matrices `A` et `B` ayant le même nombre de lignes (si `B` n'existait pas avant l'instruction, elle est créée, ses colonnes 2 et 3 étant nulles).

`A([2 3],:)=A([3 2],:)` permute les lignes 2 et 3 de la matrice `A`.

Notons également que si `x` est un vecteur (ligne), `x(:)` est le vecteur colonne des composantes de `x`.

3.4. **Opérateurs.**

`+` `-` Entre deux matrices de mêmes dimensions, il s'agit de l'addition et la soustraction des matrices. De plus, si l'une des matrices est un scalaire, l'opération aura lieu entre chaque coefficient de la matrice et le scalaire.

`*` Il s'agit de la multiplication matricielle entre deux matrices lorsque le nombre de colonnes de la première est égal au nombre de lignes de la seconde. De plus, si l'une des matrices est un scalaire, l'opération aura lieu entre chaque coefficient de la matrice et le scalaire.

`^` Il s'agit de la puissance matricielle (pour les matrices carrées). Par exemple `A^2` est la même chose que `A*A`.

`.*` `./` Utilisés entre deux matrices de mêmes dimensions, il s'agit de l'opération termes à termes (*i.e.* entre les coefficients des matrices de même ligne et même colonne). Ainsi par exemple `A*eye(size(A))` renvoie, tout comme `diag(diag(A))`, la matrice diagonale qui a la diagonale de `A`.

`.^` Utilisé entre deux matrices de mêmes dimensions, il s'agit de l'élévation à la puissance terme à terme. Utilisé entre une matrice et un scalaire, il s'agit de l'élévation à la puissance de chaque coefficient de la matrice par le scalaire.

`\` `A\B`, où `A` désigne une matrice inversible d'ordre n et `B` une matrice à n lignes, vaut (une approximation, calculée en général par la méthode de Gauss, de) $\texttt{A}^{-1}\texttt{B}$.

Attention : si `A` n'est pas carrée, `X=A\B` est défini ; il s'agit d'une solution approchée de `A*X=B` au sens des moindres carrés.

`/` `B/A=(A'\B')'` (*i.e.* `BA`$^{-1}$ si `A` est inversible).
De plus, pour une matrice `A` et un scalaire `s`, `A/s`, tout comme `A./s`, correspond à la matrice des coefficients de `A` divisés par `s`.

3.5. **Dimensions.**

Si `A` est une matrice $n \times m$ ($n \geq 1$, $m \geq 1$), ses dimensions sont obtenues par :

`I=size(A)` `I` vaut le vecteur ligne `[n,m]`,

ou `[p,q]=size(A)` `p` vaut n et `q` vaut m

De plus, `length(A)` vaut $\max(n, m)$ et `numel(A)` vaut nm, le nombre d'éléments de `A`. Donc si `x` est un vecteur, aussi bien `length(x)` que `numel(x)`, renvoient le nombre de composantes de `x`.

3.6. **Quelques fonctions.**

Il existe d'autres fonctions que celles présentées ici ; n'hésitez pas à voir l'aide selon vos besoins.

3.6.1. *Quelques fonctions "scalaires".*

Il s'agit de fonctions usuelles en mathématiques lorsqu'elles sont appliquées à un scalaire. Avec Matlab, leur application à une matrice correspond à l'application de la fonction à chaque coefficient de la matrice.

```
sin     asin    cos     acos    tan     atan
sinh    asinh   cosh    acosh   tanh    atanh
exp     log     log10   log2    sqrt    abs
```

Pour arronir un nombre, ou les coefficient d'une matrice, on peut utiliser :

`round(x)` qui renvoie l'entier le plus proche à `x` ;

`fix(x)` qui renvoie l'entier vers 0 le plus proche à `x`. Ainsi par exemple `fix(rand(3)*(10-eps))` renvoie une matrice aléatoire à coefficients entiers dans $\{0, \ldots, 9\}$.

3.6.2. *Quelques fonctions "vectorielles".*

Lorsqu'elles sont appliquées à un vecteur, ces fonctions renvoient un résultat scalaire ; appliquées à une matrice $n \times m$ avec $n \geq 2$ et $m \geq 2$, elles renvoient un vecteur ligne résultat du calcul colonne par colonne.

`max` est plus grand élément.
Par exemple `max([1 2])` est `2`, et `max([1 2; 3 4])` est `[2 4]`.

`[m, I]=max(x)` renvoie en plus `I`, le premier indice correspondant au max `m`.
Par exemple après `[m I]=max([1 7 0])` on a `m=7` et `I=2`.

`min` est le plus petit élément ; elle s'utilise comme `max`.
Pour obtenir le plus petit élément d'une matrice `A` qui n'est pas un vecteur il faut faire `max(max(A))`.

`sum` est la somme des composantes.
Par exemple `sum([1 2; 3 4])` est `[4 6]`, et `sum(sum([1 2; 3 4]))` est `10`.

`prod` est le produit des composantes. Par exemple `prod([1 2; 3 4])` est `[3 8]`.

3.6.3. *Quelques fonctions "matricielles".*

Ces fonctions, généralement basées sur des méthodes numériques, calculent diverses quantités sur des matrices :

`det(A)` est le déterminant de `A`.

`inv(A)` est la matrice inverse de `A`.

`eig(A)` renvoie le valeurs propres (et vecteurs propres) de `A`.

`null(A)` renvoie une matrices dont les colonnes forment une base du noyau de `A`.

`poly(A)` donne le polynôme caractéristique de `A`.

`norm(x)` est la normes (euclidienne) du vecteur `x`, ou plus généralement `norm(x,k)` est la norme $\|x\|_k$.

`norm(A)` est la normes matricielle (euclidienne) de `A`, ou plus généralement `norm(A,k)`, avec `k=1`, `k=2` ou `k=inf`.

`cond(A)` est le conditionnement (euclidien) de `A`, ou plus généralement on peut aussi calculer `cond(A,k)`.

`chol(A)` donne la factorisation de Choleski de la matrice symétrique définie positive `A`.

4. Les (fonctions) polynômes

Avec Matlab, un polynôme est représenté sous forme d'un vecteur dont les composantes correspondent aux coefficients du polynôme classés par ordre des puissances décroissantes (la dernière composante correspond nécessairement au terme constant, éventuellement nul).

Exemple : `P=[ 1, 0, 2, 3]` correspond au polynôme $P(X) = X^3 + 2\,X + 3$.

Quelques fonctions :

`roots(P)` est le vecteur colonne des valeurs (approchées) des zéros du polynôme `P`.

`poly(x)` pour un vecteur `x`, est le polynôme unitaire (sous la forme d'un vecteur ligne) dont les zéros sont les composantes de `x`.

`polyval(P,A)` pour un polynôme `P` et une matrice `A`, est la matrice des évaluations de `P` en chaque coefficient de `A`.

`polyvalm(P,A)` pour un polynôme `P` et une matrice `A`, est la matrice $P(A)$.

`conv(P,Q)` est le produit des deux polynômes `P` et `Q`

5. Quelques notions pour la programmation

5.1. **Comparaison et connecteurs logiques.**

5.1.1. *Les symboles de comparaison.*

`<` `<=` `>` `>=` pour $<$, $\leq$, $>$ et $\geq$.

`==` égal

`~=` différent (n'est pas égal)

Appliqués entre deux scalaires, le résultat d'une comparaison vaut 1 ou 0 selon que la comparaison est vraie ou fausse. Ce résultat est dit de type logique.

> Lorsque la comparaison a lieu entre deux matrices de mêmes dimensions (non scalaires) ou entre une matrice et un scalaire, le résultat est une matrice de type logique ; elle est composée de 0 et de 1, résultats des comparaisons coefficient par coefficient ou entre les coefficients et le scalaire.

Le résultat d'un test dans un `if` ou un `while` sera vrai si la matrice logique associée au test ne contient **que** des 1, sinon, il sera faux.

On peut utiliser le fait que le résultat d'une comparaison entre deux matrices ou une matrice et un scalaire est une matrice logique pour extraire certains coefficients : si `A` est une matrice et `B` une matrice logique de mêmes dimensions, `A(B)` est le vecteur colonne des coefficients $A_{i,j}$ tels que $B_{i,j} = 1$, ordonnés selon les colonnes de `A`.

Exemple : si `x=[2,0,3]` est un vecteur, `x(x>0)` est le vecteur colonne dont les composantes sont celles de `x` qui sont positives, c'est-à-dire `[2; 3]`.

`any(x)` renvoie `1` (vrai) si au moins un élément du vecteur `x` est non nul, `0` (faux) sinon.

`any(A)` est un vecteur ligne qui est le résultat de `any` appliqué aux vecteurs colones de `A`.

`all(x)` renvoie `1` (vrai) si tous les éléments de `x` sont non nuls, `0` (faux) sinon.

`all(A)` est un vecteur ligne qui est le résultat de `all` appliqué aux vecteurs colones de `A`.

Exemple : `all(any(A>5))` est vrai si dans tous les colones de la matrice `A` il a au moins un coefficient > 5.

Signalons également la fonction `find` retournant les indices linéaire correspondant à des coefficients non nuls. Par exemple `find([0 1; 0 1])` va nous donner `[2;4]`. Ainsi pour obtenir les indices des coefficients positifs d'un vecteur on peut faire `find(x(x>0))`.

5.1.2. *Les connecteurs logiques.*

`&` **et** logique élément par élément

`|` **ou** logique élément par élément

`&&` **et** logique scalaire version court-circuit

`||` **ou** logique scalaire version court-circuit

`~` **négation**

Les connecteurs logiques `&&` et `||` opèrent sur des valeurs logiques scalaires avec court-circuit dans l'évaluation de l'expression, c'est-à-dire avec arrêt de l'évaluation des expressions logiques qui suivent dès que le résultat est connu. Par exemple : supposons que `x` vaut `2.5` et que `y` vaut `1.3` et considérons l'expression `x>5 && y<2`. Puisque $x \leq 5$, son résultat sera nécessairement faux et le test $y < 2$ ne sera même pas évalué.

Les connecteurs logiques version élément par élément opèrent sur des valeurs logiques pouvant être des matrices de mêmes dimensions (sauf par exemple pour leur utilisation dans un `if` ou un `while`) avec comme résultat une matrice logique. L'aspect "cout-circuit" dépend de leur utilisation.

5.2. **Structures algorithmiques.**

Les choix `if ... else ... end` :

```
if test
  instructions
else
  instructions
end
```

```
if test
  instructions
end
```

```
if test else
  instructions
end
```

⟶ voir l'aide pour la structure `switch` permettant un choix entre plus de 2 possibilités.

Les boucles `while ... end` et `for ... end` :

```
while test
  instructions
end
```

```
for var = expr
  instructions
end
```

Pour la boucle `for` de Matlab, la valeur de l'expression `expr` est calculée en début de for.

En général, le résultat d'`exp` sera un vecteur et la variable `var` prendra successivement comme valeurs les composantes du vecteur. Les instructions entre `for` et `end` seront exécutées avec ces valeurs de `var`.

Si le résultat d'`exp` est la matrice vide `[]`, alors on n'entre pas dans la boucle. Si le résultat de `exp` est une matrice au sens strict, la variable `var` prendra successivement comme valeurs les vecteurs colonnes de `exp`.

Exemples de boucle :

```
for i=1:10
  x(i)=i^2;
  y(i)=x(i)+1;
end
```

```
x=round(10*rand(1,20));
a=6;ind=[];
for i=length(x):-1:1
  if x(i)==a
    ind=[ind;i];
  end
end
```

```
som=0; k=0;
while som<100
  k=k+1;
  som=som+k;
end
```

Lorsque des outils Matlab (les opérations matricielles, les fonctions `linspace`, `max`, `sum` ...) permettent un calcul similaire sans avoir recours aux boucles, le programme sans les boucles sera nettement plus efficace (en temps).

Exemple : Il est préférable d'écrire le premier exemple précédent sans la boucle `for` sous la forme : `x=(1:10).^2; y=x+1;`

La commande `break` arrête une boucle en passant à l'instruction suivant le `end` de la boucle où elle apparaît. La commande `return` permet de sortir d'une fonction (ou d'un script).

5.3. Fonctions `inline` et `@`.

Lorsque l'on définit une fonction, il est préférable d'utiliser systématiquement les opérateurs terme à terme `.*`, `./` et `.^` au lieu de `*`, `/` et `^`, si l'on veut que cette fonction puisse s'appliquer à des vecteurs et des matrices.

`inline` : Cette fonction permet de définir une fonction simple à partir d'une chaîne de caractères.

Exemples : `f=inline('t.*sin(t)+2')` définit la fonction : $f(t) = t\sin(t) + 2$.

`g=inline('exp(x.*y)','x','y')` définit la fonction : $g(x, y) = \exp(xy)$.

Le `.*` est ici essentiel : on peut alors évaluer ces fonctions sur des vecteurs ou des matrices :

```
t=linspace(0,pi,100); y=f(t);}
[x,y]=meshgrid(-1:0.1:1,-1:0.1:1); z=g(x,y)}
```

`@` : Cette fonction est similaire à `inline` mais elle présente l'avantage de pouvoir passer des variables comme paramètres.

Exemples : `a=2; b=3;`

`f=@(x)[a*x+b*sin(x)]` définie la fonction $f(x) = 2x + 3\sin(x)$;

`g=@(x)[a*x, cos(b*x)]` définie $g(x) = (2x, \cos(3x))$ à valeurs dans $\mathbb{R}^2$.

Si une fonction n'est pas prévue pour fonctionner avec des vecteurs (comme par exemple `inline('x*x')`) on peut l'appliqué à un vecteur quand même en utilisant `arrayfun`.

6. Un peu de graphisme bidimensionnel

Matlab permet la visualisation de valeurs numériques dans des fenêtres `figure`.

Sans précision, c'est dans la `figure 1` que se font les tracés. On accède à (ou on ouvre) une autre fenêtre, `figure 2` par exemple, par la commande `figure(2)`. La fermeture de la même fenêtre (resp. de toutes les fenêtres `figure`) peut se faire par la commande `close(2)` (resp. `close all`).

La commande de base pour du graphisme bidimensionnel est `plot(x,y)`, où `x` et `y` sont deux vecteurs à composantes réelles de même dimension n. Elle ouvre une fenêtre `figure` (`figure 1` par défaut) dans laquelle vont figurer les points de coordonnées $(x_1, y_1), \ldots, (x_n, y_n)$ reliés par des segments de droite dans un repère cartésien. Si `y` est une matrice, plusieurs courbes seront tracées (une pour chaque ligne de `y`).

Les commandes `loglog`, `semilogx` et `semilogy` fonctionnent de la même manière que `plot` mais avec des échelles logarithmiques en x et y, en x seulement ou en y seulement.

Il est possible d'ajouter des options (spécifications) pour le tracé :

`plot(x,y,':or')` : le symbole `:` pour relier les points par des traits en pointillés, le `o` pour repérer les points (x_i, y_i) par un petit rond et le `r` pour un tracé en rouge.

`plot(x,y,'+')` : ici, l'absence du style de trait dans la spécification provoquera l'absence de trait entre les points (seul le marquage des points par le symbole `+` sera effectué).

$\longrightarrow$ Voir l'aide dans `LineSpec` pour les spécifications possibles.

Avec `plot`, on peut également tracer plusieurs courbes sur la même figure :

`plot(x1,y1,'-vb',x2,y2,':xg')` pour tracer sur un même graphe, `y1` suivant `x1` en bleu par un trait plein avec des symboles $\triangledown$, et `y2` suivant `x2` en vert par un trait pointillé avec des symboles $\times$.

Citons quelques commandes que l'on peut utiliser en les plaçant après la commande `plot` :

`legend('courbe1','courbe2',...)` pour placer une légende pour les différentes courbes ;

`xlabel('texte')` (ou `ylabel`) pour mettre la signification de l'axe des abscisses (ou des ordonnées) ;

`title('texte')` pour donner un titre à la figure ;

`axis([xmin xmax ymin ymax])` pour définir les bornes des deux axes.

Si au moins une fenêtre `figure` est ouverte, la commande `plot` effectue le tracé dans la dernière fenêtre `figure` ouverte, en remplaçant l'éventuel graphe déjà présent dans cette fenêtre. On conseille donc de toujours débuter les programmes par la commande `close all`.

Une fenêtre `figure` peut également être partagée en plusieurs zones sous forme d'un tableau pour y faire figurer plusieurs graphes grâce à la commande :

`subplot(n,m,pos)` partage la fenêtre des figures en `n` lignes et `m` colonnes et fix la position de la prochain `plot` en la position `pos`.