VII. Traitements mathématiques

1. Introduction

Matlab a des outils très puissants pour travailler avec des polynômes et effectuer des optimisations. Nous verrons comment trouver les zéros d'un polynôme et résoudre des systèmes d'équations. Finalement, la gestion du temps et des dates sera présentée.

2. Polynômes

Dans Matlab, les polynômes sont traités comme des vecteurs de coefficients. L'équation suivante

$$2x^5 + 4x^4 + x^3 - 6x^2 - x + 5 = 0$$

…prend cette forme :

```
poly = [2 4 1 -6 -1 5];
```

Matlab permet de travailler aisément avec ces polynômes et offre de multiples fonctions [2] :

Fonctions	Description
`poly`	Construction de polynômes à partir des racines
`roots`	Calcul des racines
`polyval`	Évalue à un point
`polyvalm`	Évaluation en une matrice de points
`deconv`	Division de polynômes
`conv`	Multiplication de polynômes
`residue`	Décomposition en résidus
`polyfit`	Approximation du polynôme
`polyint`	Intégration du polynôme
`polyder`	Dérivée du polynôme

2.1. Fonction roots

Comme nous l'avons vu précédemment, les polynômes peuvent être facilement résolus. Par exemple, reprenons une équation polynomiale d'ordre 5 :

$$2x^5 + 4x^4 + x^3 - 6x^2 - x + 5 = 0$$

Trouver les zéros de cette équation est alors très simple à l'aide de la fonction `roots` :

```
>> myres = roots(poly)
```

```
myres =
  -1.2530 + 1.0941i
  -1.2530 - 1.0941i
  -1.1156
   0.8108 + 0.3906i
   0.8108 - 0.3906i
```

Il est ensuite possible de tester si les résultats sont des racines réelles à l'aide de la fonction `isreal`:

```
>> isreal(myres(1))

ans =
     0

>> isreal(myres(3))

ans =
     1
```

2.2. Fonction polyval

La fonction polyval permet d'évaluer un polynôme aux x désirés.

```
y = polyval(p,x)
```

Par exemple, pour le polynôme $p(x) = 6x^2 + 3x + 1$, les valeurs de y aux point x = 1, 10 et 50 peuvent être obtenus par la commande suivante :

```
>> p = [6 3 1];
>> polyval(p,[1 10 50])

ans =
          10         631       15151
```

2.3. Fonction polyfit

Pour rappel, Matlab permet d'effectuer une régression sur des données (Voir cours Les Fonctions). La solution peut être obtenue directement grâce à la fonction `polyfit` :

```
>> x = [0,3,4,7,8,10]
>> y = [1,4,5,6,7,10]
>> resmodel = polyfit(t,x,1)

resmodel =
    0.8020    1.2228
```

Ce qui signifie : $y^{est} = 0.8020 \cdot x + 1.2228$

2.4. Fonction polyder

La fonction `polyder` calcule la dérivée du polynôme.

```
k = polyder(p)
```

La dérivée du polynôme $p(x) = 9x^3 - 6x^2 + 3x + 1$ s'obtient ainsi :

```
>> p = [9 -6 3 1];
>> k = polyder(p)

k =
    27   -12     3
```

3. Fonctions d'optimisation

Les fonctions d'optimisation disponibles sont les suivantes [2]:

Fonctions	Description
`fsolve`	Résolution d'un système d'équations non-linéaires
`fzero`	Trouve les zéros d'une fonction à une variable
`fminbnd`	Minimisation d'une fonction à une variable
`fminsearch`	Minimisation d'une fonction à plusieurs variables

3.1. Fonction fsolve

La fonction `fsolve` résout (trouve le zéro) un système d'équations non linéaires [1] (avec des variables multiples). Les deux paramètres obligatoires sont la fonction elle-même (`fun`) et le point de départ de la résolution (`x0`) :

```
x = fsolve(fun,x0)
x = fsolve(fun,x0,options)
```

Les options sont facultatives et sont définies par la commade `optimset` (`optimget` permet de connaître les options actuelles). D'autres sorties peuvent également être récupérées [1] :

```
[x,fval,exitflag,output] = fsolve(...)
```

Où `fval` est la valeur de la fonction à la solution x, `exitflag` est une indication de la raison de la fin de l'optimisation (convergence vers la solution x, nombre d'optimisations limite dépassé, …), `output` contient des informations sur l'optimisation.

Exemple avec le système d'équation :

$$\begin{cases} 2x_1 - x_2 - e^{-x_1} = 0 \\ -x_1 + 2x_2 - e^{-x_2} = 0 \end{cases}$$

```
function F = myfun(x)
F = [5*x(1) - 2*x(2) - exp(-x(1));
      -2*x(1) + 10*x(2) + exp(-x(2))];
```

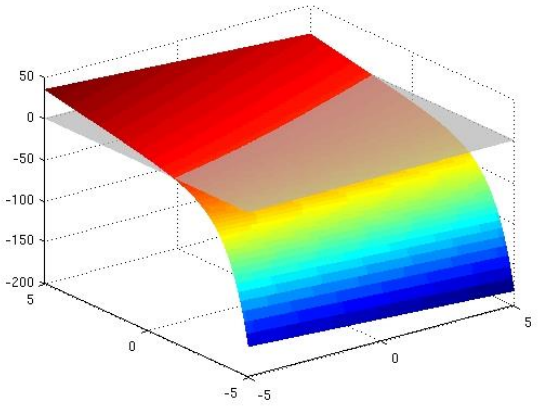

Fig 1. Eq. 1

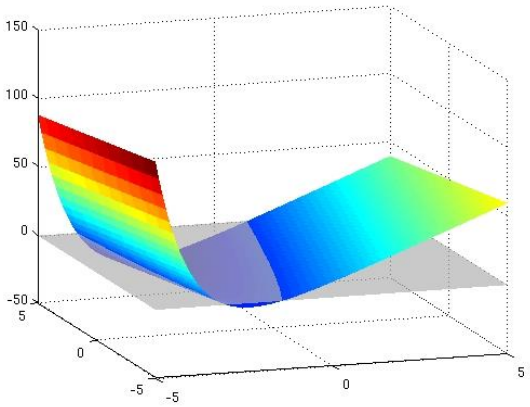

Fig 2. Eq. 2

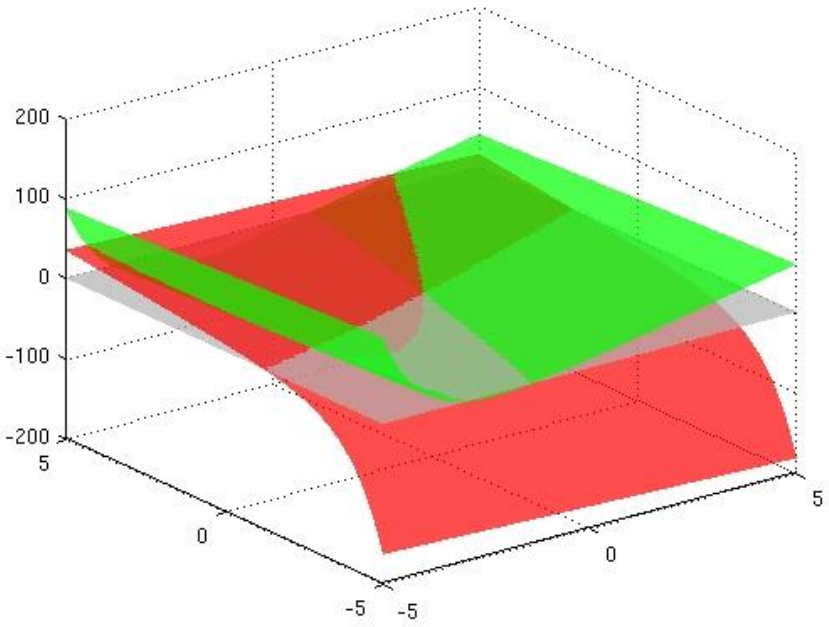

Fig 3. Système d'équations

Cette fonction a plusieurs solutions (Fig 3). Celle que nous allons approcher dépend donc de la valeur initiale que nous fixons !

Avec x_0 = [-10 -10], la solution est x = [-0.906 -3.503] :

```
>> x0 = [-10; -10];
>> [x,fval,exitflag,output] = fsolve(@myfun,x0)

Optimization terminated: first-order optimality is less than
options.TolFun.

x =
   -0.9062
   -3.5031

fval =
   1.0e-14 *
   -0.7994
         0

exitflag =
     1

output =
       iterations: 15
        funcCount: 48
        algorithm: 'trust-region dogleg'
    firstorderopt: 5.9752e-14
          message: 'Optimization terminated: first-order
optimality is less than options.TolFun.'
```

Avec x_0 = [10 10], la solution est x = [0.142 -0.080] :

```
>> x0 = [10; 10];
>> [x,fval,exitflag,output] = fsolve(@myfun,x0)

Optimization terminated: first-order optimality is less than
options.TolFun.

x =
    0.1416
   -0.0800

fval =
   1.0e-13 *
   -0.7427
    0.0333

exitflag =
     1

output =
       iterations: 7
        funcCount: 24
        algorithm: 'trust-region dogleg'
    firstorderopt: 4.4250e-13
```

A l'aide de l'option `Display`, il est possible d'afficher toute les étapes de l'itération :

```
>> x0 = [10; 10];
>> options=optimset('Display','iter');
>> [x,fval] = fsolve(@myfun,x0,options)

                                         Norm of      First-order   Trust-region
 Iteration  Func-count     f(x)          step         optimality    radius
     0          3            7300                                    740                1
     1          6         5924.68             1                      636                1
     2          9         3382.64           2.5                      378              2.5
     3         12         799.791          6.25                      136             6.25
     4         15         1.24938       6.43098                     7.04             15.6
     5         18     0.000159635      0.162424                   0.0745             16.1
     6         21     5.52223e-12     0.00237679                  1.4e-05             16.1
     7         24     5.52771e-27    4.34761e-07                 4.42e-13             16.1
Optimization terminated: first-order optimality is less than options.TolFun.
```

3.2. Fonction fzero

La fonction `fzero` cherche une racine d'une fonction continue à une variable :

```
x = fzero(fun,x0)
x = fzero(fun,x0,options)
```

Les paramètres d'entrée sont les mêmes que pour `fsolve`. La valeur de départ x0 peut également être un intervalle de recherche.

Les autres sorties possibles sont :

```
[x,fval,exitflag,output] = fzero(...)
```

Exemple:

```
>> fun = inline('x.^3 + 10.*x','x');
>> [x,fval,exitflag,output] = fzero(fun,10)

x =
  -2.6833e-24

fval =
  -2.6833e-23

exitflag =
     1

output =
    intervaliterations: 12
            iterations: 13
             funcCount: 37
             algorithm: 'bisection, interpolation'
               message: 'Zero found in the interval [-2.8, 19.05]'
```

3.3. Fonction fminbnd

La fonction `fminbnd` cherche le minimum d'une fonction à une variable sur un intervalle donné. L'intervalle est précisé par x_1 et x_2, tel que $x_1 < x < x_2$. Les options en entrées ainsi que les sorties informatives sont les mêmes que pour `fsolve` et `fzero`.

```
x = fminbnd(fun,x1,x2)
x = fminbnd(fun,x1,x2,options)
[x,fval,exitflag,output] = fminbnd(...)
```

Exemple:

```
>> fun = inline('x.^2 + 10.*x','x');
>> x1 = -10;
>> x2 = 10;
>> x = fminbnd(fun,x1,x2)

x =
   -5.0000
```

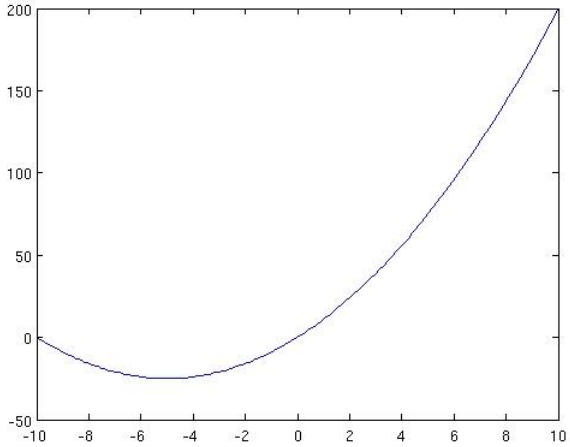

Fig 4. Illustration de la fonction (effectué à l'aide de `fplot`)

3.4. Fonction fminsearch

La fonction `fminsearch` cherche le minimum d'une fonction à plusieurs variables. Les options en entrées ainsi que les sorties informatives sont les mêmes que pour `fsolve` et `fzero`.

```
x = fminsearch(fun,x0)
x = fminsearch(fun,x0,options)
[x,fval,exitflag,output] = fminsearch(...)
```

4. Intégration d'une fonction

La fonction `trapz` permet d'évaluer l'intégrale de données par la méthode des trapèzes (Fig 5), mais ne peut pas évaluer directement l'intégrale d'une fonction. La fonction `quad` permet d'évaluer l'intégrale d'une fonction par la méthode de quadrature de Simpson (Fig 6) :

```
q = quad(fun,a,b)
```

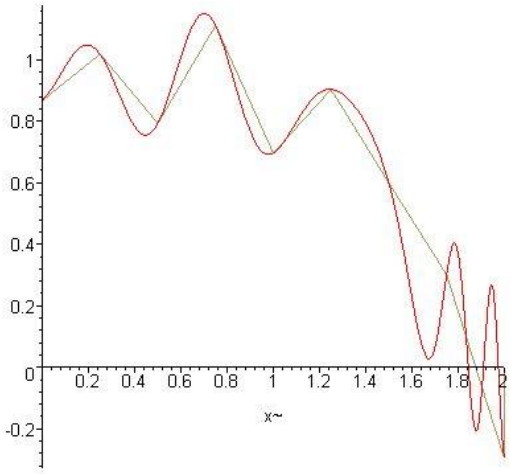

Fig 5. Intégration selon la méthode des trapèzes (source : [3])

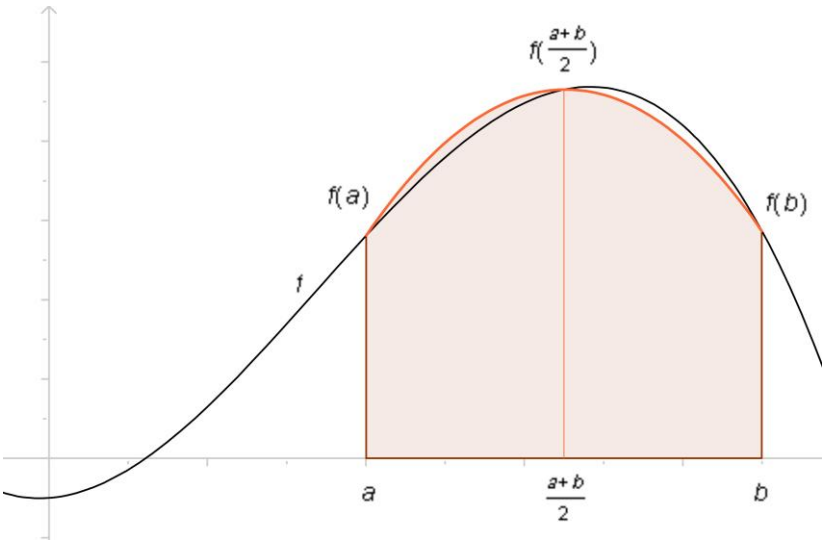

Fig 6. Intégration selon la méthode de Simpson (source : [3], auteur : Wolfgang Dvorak)

Il existe d'autres methods d'intégration implémentées dans Matlab : `dblquad`, `quadgk`, `quadl`, `quadv`.

5. Temps et dates

Matlab permet de faciliter le travail de données temporelles par une intégration d'outils spécifiques pour les dates et le temps.

5.1. Formats

Il existe trois formats de dates dans Matlab [1] :

- En chaîne de caractères : `'24-Oct-2003 12:45:07'`
- En vecteur : `[2003 10 24 12 45 07]`
- En numéro de série : `7.3188e+005`

Le numéro de série de la date (serial date number) est une valeur unique correspondant au numéro du jour depuis le 1.1.0000 00:00 (=1). C'est un format courant et pratique pour travailler avec des séries temporelles. Ce numéro de série peut être obtenu à l'aide de la fonction `datenum`, qui prend en entrée de multiples formats (textes ou vecteurs) :

```
N = datenum(...)
```

Exemples :

```
>> n = datenum('02-Jul-1999', 'dd-mmm-yyyy')
>> n = datenum(1999, 07, 02)
n =
      730303
```

La fonction `datestr` converti une date au format texte. Il existe de multiples formats de sortie (voir Help) :

```
S = datestr(...)
```

Exemples [1] :

```
>> datestr(t1)
ans =
09-Nov-2009 15:31:00

>> datestr(t1, 'mmmm dd, yyyy HH:MM:SS.FFF AM')
ans =
November 09, 2009  3:31:11.788 PM
```

Finalement, il est également possible d'obtenir la date dans le format vectoriel par la fonction `datevec` :

```
V = datevec(...)
```

Le vecteur est définit ainsi : [`année mois jour heure minute seconde`]

La fonction `datetick` permet de formater l'axe du temps sur un graphique (voir Help).

5.2. Date actuelle

La date actuelle (dd-mmm-yyyy) peut être obtenue par la commande `date` :

```
>> str = date

str =
09-Nov-2009
```

L'heure peut également être connue par la commande `now`. Le retour est alors un nombre correspondant au « numéro de série » de la date.

```
>> t = now

t =
   7.3409e+05
```

Une autre manière d'obtenir cette donnée formatée sous la forme vectorielle est la commande `clock` :

```
>> c = clock

c =
   1.0e+03 *
    2.0090    0.0110    0.0090    0.0150    0.0120    0.0156
```

5.3. Différence entre deux dates

La fonction `etime` permet de calculer la différence entre deux vecteurs temps :

```
e = etime(t2, t1)
```

6. Références

[1] MATLAB Help

[2] Youmaran Richard, Bouchard Martin, 2003. *Introduction à Matlab*

[3] Wikipedia

7. Auteur(s)

Pascal Horton (2009, 2015)