Analyse Numérique

1. Analyse de régression

L'ajustement de courbe, ou **analyse de régression**, est une technique fondamentale en statistiques et en analyse de données. Elle consiste à trouver une fonction (souvent un polynôme) qui s'ajuste au mieux à un ensemble de données expérimentales.

MATLAB propose des fonctions puissantes pour effectuer cet ajustement, notamment la fonction `polyfit`.

Ajustement polynomial avec `polyfit`

Syntaxe :

```
p = polyfit(x, y, n)
```

Où :

— `x` : vecteur des abscisses (valeurs de la variable indépendante),
— `y` : vecteur des ordonnées (valeurs mesurées),
— `n` : degré du polynôme d'ajustement,
— `p` : vecteur contenant les coefficients du polynôme obtenu.

Exemple :

```
x = [0 1 2 3 4 5];
y = [2.1 2.9 7.2 14.8 30.2 55.1];
p = polyfit(x, y, 2)
```

Le polynôme ajusté est alors :

$$P(x) = p(1)x^2 + p(2)x + p(3)$$

Pour évaluer le polynôme en différents points :

```
x2 = 0:0.1:5;
y2 = polyval(p, x2);
plot(x, y, 'o', x2, y2, '-r')
```

— L'ajustement de courbes (*curve fitting*) ne se limite pas aux polynômes. De nombreux phénomènes physiques, biologiques ou économiques suivent des lois de type exponentiel, puissance, logarithmique ou réciproque. MATLAB permet d'ajuster ces fonctions en les transformant sous une forme linéaire et en appliquant ensuite la commande `polyfit`.

1. Principe général

On cherche à relier deux variables (x, y) par une fonction non polynomiale, telle que :

$$y = f(x, a, b),$$

où a et b sont des paramètres à déterminer. Si la fonction peut être transformée en une relation linéaire entre deux nouvelles variables, on peut utiliser `polyfit` pour trouver les coefficients du modèle.

2. Types de fonctions courantes

1. **Fonction puissance :**
$$y = bx^a.$$
En prenant le logarithme des deux côtés :
$$\ln(y) = a\ln(x) + \ln(b).$$
On pose $Y = \ln(y)$ et $X = \ln(x)$, puis on ajuste Y en fonction de X par un modèle linéaire.
2. **Fonction exponentielle :**
$$y = be^{ax}.$$
En prenant le logarithme :
$$\ln(y) = ax + \ln(b).$$
On ajuste donc $\ln(y)$ en fonction de x.
3. **Fonction logarithmique :**
$$y = a\ln(x) + b.$$
Le modèle est déjà linéaire par rapport à $\ln(x)$.
4. **Fonction réciproque :**
$$y = a + \frac{b}{x}.$$
On ajuste y en fonction de $\frac{1}{x}$.

3. Exemple MATLAB : ajustement exponentiel

On considère les données suivantes :

```
x = [0 1 2 3 4 5];
y = [2.0 2.7 3.8 5.4 7.4 10.1];
```

Le modèle supposé est $y = be^{ax}$. On linéarise en posant $Y = \ln(y)$, puis on applique `polyfit`.

```
Y = log(y);
p = polyfit(x, Y, 1);
a = p(1);
b = exp(p(2));

% Valeurs ajust es
y_fit = b * exp(a * x);

% Affichage du r sultat
plot(x, y, 'o', x, y_fit, '-r')
xlabel('x'), ylabel('y')
legend('Donn es', 'Ajustement exponentiel')
title('Ajustement exponentiel : y = b e^{a x}')
```

4. Résultats et interprétation

Les coefficients a et b permettent de décrire le modèle exponentiel :

$$y = be^{ax}.$$

La courbe ajustée passe proche des points expérimentaux, illustrant la validité du modèle choisi. Ce type d'ajustement est très utilisé pour les processus de croissance, de décroissance, ou de réaction chimique.

—

Remarque : Pour d'autres types de fonctions non linéaires plus complexes, MATLAB dispose aussi d'outils dédiés comme la commande `fit` ou l'application **Curve Fitting Tool**.

2. Interpolation

L'**interpolation** consiste à construire une fonction qui passe *exactement* par les points de données donnés. Elle permet de reproduire fidèlement les valeurs connues et d'estimer des valeurs intermédiaires.

Interpolation avec `interp1`

Syntaxe :

```
yi = interp1(x, y, xi, 'methode')
```

Où :

— `x`, `y` : vecteurs des points connus,
— `xi` : valeurs où l'on veut interpoler,
— `'methode'` : méthode d'interpolation (linéaire par défaut).

Exemple :

```
x = [0 1 2 3 4];
y = [0 2 8 18 32];
xi = 0:0.5:4;
yi = interp1(x, y, xi, 'linear');
plot(x, y, 'o', xi, yi, '-r')
```

Autres méthodes disponibles :

— `'nearest'` : interpolation au plus proche,
— `'spline'` : interpolation cubique spline,
— `'cubic'` : interpolation cubique classique.

3. Intégration numérique

L'intégration numérique consiste à calculer une intégrale lorsque la fonction $f(x)$ est connue sous forme analytique ou discrète (points). MATLAB propose plusieurs fonctions d'intégration : `quad`, `quadl` et `trapz`.

3.1 Fonction `quad`

Méthode d'intégration adaptative de Simpson.
Syntaxe :

```
int = quad(@f, a, b)
```

Exemple :

```
f = @(x) x.^2;
I = quad(f, 0, 2)
```

—

3.2 Fonction `quadl`

Méthode d'intégration adaptative de Lobatto (plus efficace pour des intégrales lisses ou une précision élevée).

```
f = @(x) exp(-x.^2);
I = quadl(f, 0, 1)
```

—

3.3 Fonction `trapz`

Méthode trapézoïdale pour des données discrètes.
Syntaxe :

```
q = trapz(x, y)
```

Exemple :

```
x = 0:0.1:2*pi;
y = sin(x);
I = trapz(x, y)
```

—

4. Équations différentielles ordinaires (EDO)

MATLAB possède plusieurs **solveurs** d'équations différentielles de la forme :

$$y'(t) = f(t, y)$$

avec condition initiale $y(t_0) = y_0$.

4.1 Syntaxe générale

```
[t, y] = solver_name(ODEfun, tspan, y0)
```

Où :
— `ODEfun` : fonction qui définit le système $y' = f(t, y)$,
— `tspan = [t0 tf]` : intervalle de temps,
— `y0` : condition(s) initiale(s).

Exemple : Résolution de l'équation $y' = -2y$, avec $y(0) = 1$ sur l'intervalle $[0, 5]$:

```
f = @(t, y) -2*y;
[t, y] = ode45(f, [0 5], 1);
plot(t, y)
```

—

4.2 Principaux solveurs MATLAB

Solveur	Utilisation principale
`ode45`	Le solveur le plus utilisé (méthode de Runge–Kutta 4/5). Premier choix général.
`ode23`	Plus efficace pour les problèmes tolérant une faible précision.
`ode113`	Plus rapide que `ode45` pour les problèmes nécessitant une très haute précision.
`ode15s`	À utiliser si le problème est raide (stiff).
`ode23s`	Variante efficace pour problèmes raides avec tolérances grossières.
`ode23t`, `ode23tb`	Résolvent des équations différentielles modérément raides.
`ode15i`	Pour les équations différentielles implicites ou les équations algébro-différentielles.

—

Exemple : Système d'équations

$$\begin{cases} y_1' = y_2, \\ y_2' = -y_1, \end{cases} \qquad y_1(0) = 1,\ y_2(0) = 0.$$

```
f = @(t, y)[y(2); -y(1)];
[t, y] = ode45(f, [0 10], [1 0]);
plot(t, y(:,1), 'r', t, y(:,2), 'b')
legend('y_1', 'y_2')
```

—

Remarque : Les fonctions `odeset` et `odeget` permettent de définir ou d'ajuster les tolérances et options des solveurs (par exemple, le Jacobien, les erreurs relatives ou absolues).