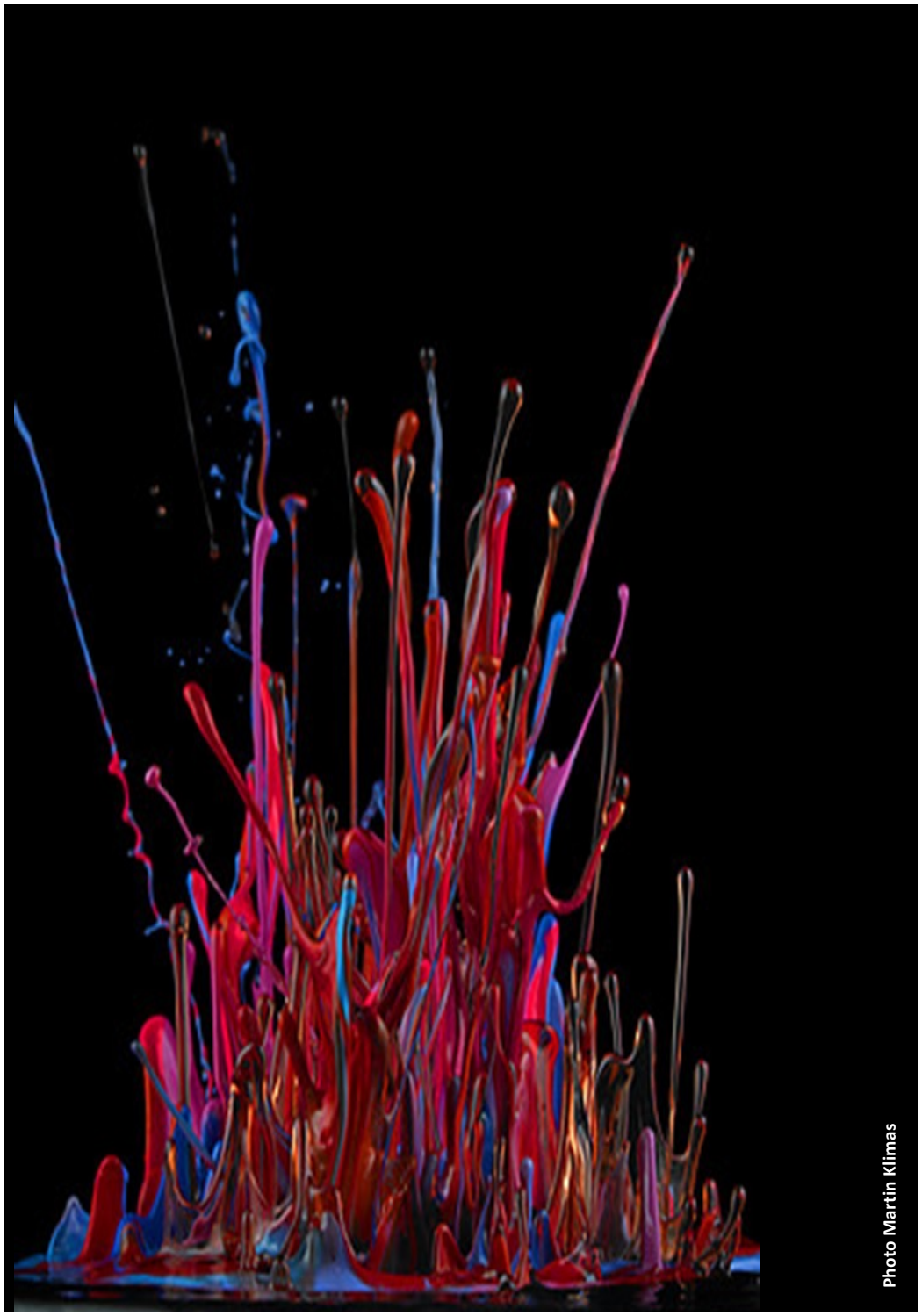


Photo Martin Klimas

EPFL

Programmation pour Ingénieur

Matlab III

ME 3e semestre

rev. 2025.r2

Christophe Salzmann

Laboratoire d'Automatique

Recap

- Programmation similaire à C/C++
  - `for n=V`
  - `while cond`
- Vectorization, find()
- Function()
- Plot pour un graphique simple
  - Handle sur les graphiques pour les changer de manière interactive
- @function, fonction anonyme

Vectorisation avec `find()`

On désire connaître le nombre d'éléments >0 d'un vecteur

La fonction `find(cond)` retourne les **indices** des éléments du vecteur/matrice pour lesquels la condition est vrai

```
>> x = [10, 0, 25, 0, 80];

>> y = find(x)

>> z = find(x~=0)

>> v = x(find(x~=0))

>> w = find(x>10 & x< 60)

>> S = length(find(x<0))
```

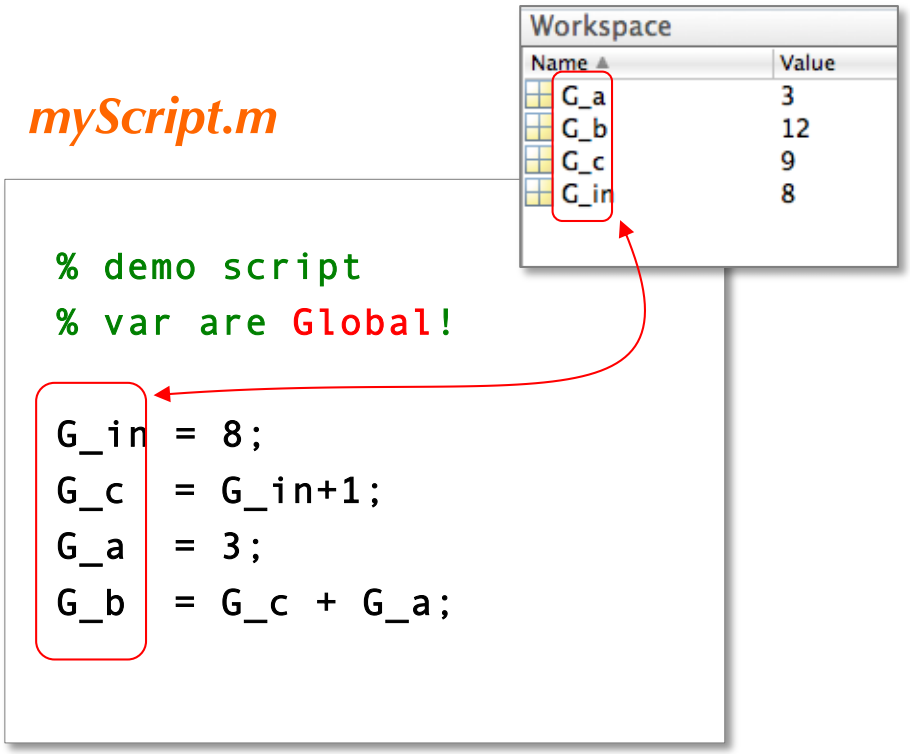
Vectorisation avec `find()`

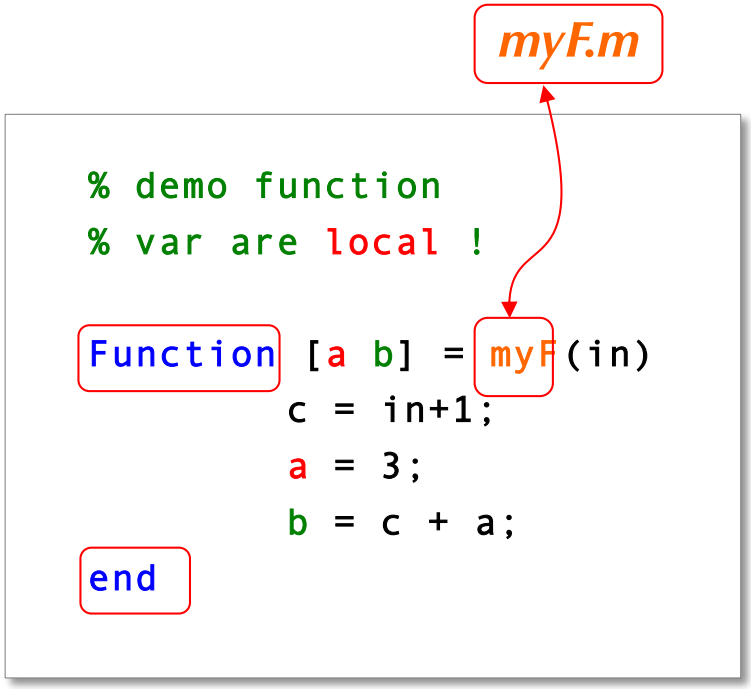
On désire connaître le nombre d'éléments >0 d'un vecteur

La fonction `find(cond)` retourne les **indices** des éléments du vecteur/matrice pour lesquels la condition est vrai

```
>> x = [10, 0, 25, 0, 80];

>> y = find(x)
                        y =  [ 1   3   5 ]
>> z = find(x~=0)
                        z =  [ 1   3   5 ]
>> v = x(find(x~=0))
                        v =  [ 10  25  80 ]
>> w = find(x>10 & x< 60)
                        w = 3
>> S = length(find(x<0))  % find returns []
                        S = 0
```

Script vs Function





Seulement des variables globales
Ecrase les variables du *workspace*!
Pas de paramètres

Function *SameAsFileName*
…
end

∆ nom de la function!

@ fonction anonyme

Matlab permet de définir une référence sur une fonction à l'aide de *@myfunc*

Définir une référence (handle) sur une fonction permet par exemple de passer cette référence en paramètre d'une autre fonction.

Une fonction annonyme est associée à une référence (handler). Le corps de la fonction est défini sur 1 ligne.

```
myHandler = @ (params) core
```

Δ Pas de '**=**' ni de ***if***, ***while, for*** dans le corps de la fonction

Exemples

```
>> mySQR= @(x) x.^2;
>> mySQR(5)
ans =  25
>> a = integral(mySQR,0,1) % mySQR est déjà une fcn. anonyme -> pas de '@'
a =  0.3333

%% dans un fichier .m
function y = myF(x)
  y = x.^3;
end
>> b = integral(@myF,0,1)  % '@' car reference sur une fonction standard existante
b = 0.25
```

Fonction vs @ fonction anonyme

```
hypoRef = @(x,y) sqrt(x.^2 +y.^2)
```

?

```
Function hyp = hypo(x,y)
    hyp = sqrt(x.^2 +y.^2);
end

hypoRef2 = @hypo;
```

```
Function [hyp, xy] = hypo2(x,y)
    hyp = sqrt(x.^2 +y.^2);
    xy = x.*y;
end
```

```
>> hypoRef(3,4)
5

>> hypoRef2(3,4)
5
```

@ fonction anonyme et '='

Ecrire une fonction anonyme qui replace la 2e valeur d'un vecteur par son inverse

```
Inv2 = @(X) X[2] = -X[2]
```

Une solution possible:

```
Inv2 = @(X) [X(1), -X(2), X(3:end)]
```

@ fonction anonyme et '='

Ecrire une fonction anonyme qui replace la 2e valeur d'un vecteur par son inverse

```
Inv2 = @(X) X[2] = -X[2]
```

Une solution possible:

```
Inv2 = @(X) [X(1), -X(2), X(3:end)]
```

Si le vecteur d'entrée a moins de 3 valeurs retourne le sans le modifier?
Rappel: pas de *if* dans les fonctions anonymes

```
Inv2 = @(X) (length(X) > 2)   * [X(1), -X(2), X(3:end)] +
            (length(X)<= 2)   *  X
```

'1' si true — * Solution_true

'0' si true — * Solution_false

=> Solution peu lisible/compréhensible, une @function n'est pas idéale dans ce cas!

@ fonction anonyme et `deal()`

`deal()` distribue les entrées sur les sorties

`[B1,...,Bn] = deal(A1,...,An)` copie A1 dans B1.. An dans Bn

`[B1,...,Bn] = deal(A)` copie A dans B1 .. Bn

`deal()` peut-être employé pour avoir un fonction anonyme avec plusieurs paramètres de retour

Ex:

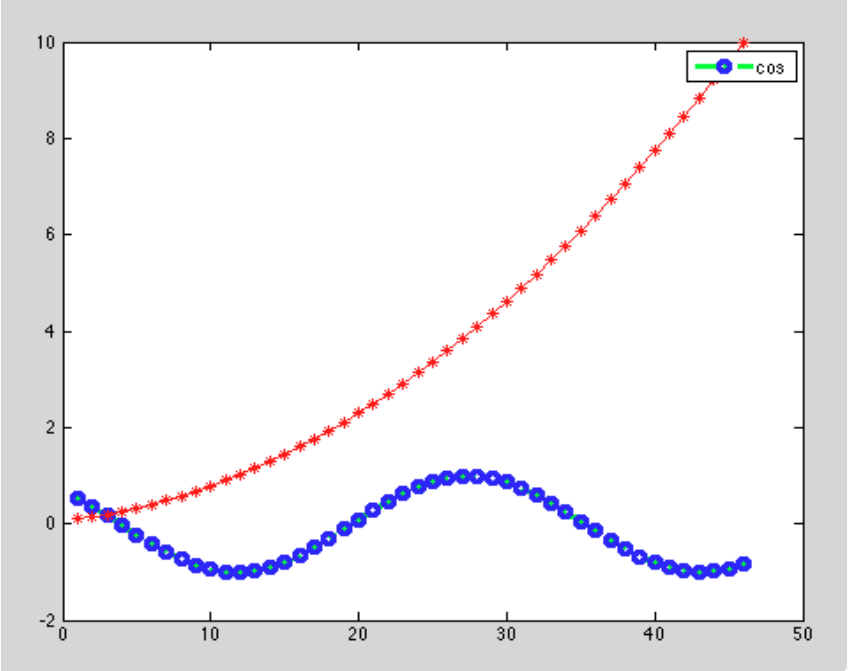
```
CarreCube = @(x) deal(x.^2, x.^3)

[x2 x3] = CarreCube(3)

>>  x2 = 9
>>  x3 = 27
```

```
% Δ nb return params must match !
```

Plot reference

La fonction plot() retourne un **handle** (référence) sur la structure du plot. Il est ainsi possible de modifier le contenu et l'apparence du plot au travers de cette référence en utilisant `set(h,…)` et `get(h,…)`, `get(h)` retourne l'ensemble des propriétés.

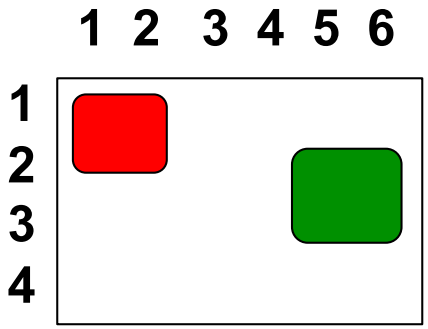
```
>> x=1:0.2:10
>> h = plot(x)
>> set(h,'YData',sin(x))

>> set(h,'LineStyle','-.')
>> set(h,'LineWidth',2)
>> set(h,'Color',[1,0,0])
>> set(h,'Marker','o')
>> set(h,'MarkerEdgeColor','b')

>> set(h,'YData',cos(x))
>> ah = get(gcf,'CurrentAxes')
>> lh = legend(ah,'show');
>> set(lh,'String',{'cos'})
>> hold on
>> h1=plot(x.^2./10,'-r*');
```



gcf: référence sur la figure courante

Q: Non-continuous matix access

Comment accéder à des cases distinctes d'une matrice en spécifiant les lignes/colonnes?




```
M([1 2],[1 2])      OK
M([1 2; 2 3],[1 2; 5 6])    Not OK !
```

Solution: convertir les coordonnées lignes/colonnes en positions

```
L= [1 1 2 2  2 2 3 3]
C= [1 2 1 2  5 6 5 6]

Pos= sub2ind([4,6],L,C)
>> Pos= 1   5   2   6  18  22  19  23

M(Pos) = 0
```

`[l c] = ind2sub(Sz,Pos)` converti **Pos** en coordonnées lignes/colonnes

Today

- Logical array
- Cell arrays
- Surface 3D et meshgrid
- Courbes paramétriques
- Interpolation
- Fit

Logical array - indexing

- Un *logical array* est une matrice de *boolean* (0,1) destinée à faire une indexation logique. Les éléments d'une matrice M pourront être sélectionnés en fonction des valeurs du *logical array*
- La fonction `logical()` transforme les éléments d'une matrice en `0,1`
- Les opérateurs logiques (`==,<,>,~, etc`) appliqués sur une matrice produisent également un *logical array*.

```
>> R = rand(2,3)
R =
    0.9502    0.4387    0.7655
    0.0344    0.3816    0.7952


>> B = R > 0.5
B =  2×3 logical array

     1     0     1
     0     0     1
```

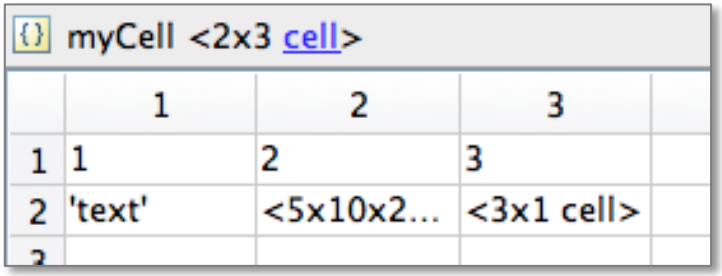
```
>> V = R(B)
V =

    0.9502
    0.7655
    0.7952

% Equivalent to

V = R(R>0.5)
```

Cells array

Les *cell array* sont similaire aux matrices à la différence près que le type de contenu de chaque case peut être variable. Il est ainsi possible d'avoir une matrice contenant des nombres, du texte et d'autres matrices.

Un *cell array* est défini à l'aide de '{' et '}' les opérations sur les *cell arrays* sont similaires aux opèrations sur les matrices.

```
>> myCell = {1, 2, 3; 'text', rand(5,10,2), {11; 22; 33}}

myCell =   2×3 cell array

    {[   1]}    {[          2]}    {[     3]}
    {'text'}    {5×10×2 double}    {3×1 cell}
```



Cells array - access

```
>> myCell = {1, 2, 3; 'text', rand(5,10,2), {11; 22; 33}}

myCell = {[   1]}    {[             2]}    {[      3]}
         {'text'}    {5×10×2 double}    {3×1 cell}
```

Accolades -> valeurs

```
>> myCell{2}
ans = 'text'

>> myCell{3}
ans = 2

>> myCell{6}
ans = 3×1 cell array
    [11]
    [22]
    [33]
```

Parenthèses -> cellules

```
>> myCell(2)
ans =  1×1 cell array
    {'text'}

>> myCell(3)
ans = 1×1 cell array
    {[2]}

>> myCell(6)
ans =  1×1 cell array
    {3×1 cell}
```

Cells array - access

```
>> myCell = {1, 2, 3; 'text', rand(5,10,2), {11; 22; 33}}
myCell =  {[   1]}    {[          2]}    {[      3]}
          {'text'}    {5×10×2 double}    {3×1 cell}


>> myCell (3) = 4
Conversion to cell from double is not possible.

>> myCell (3) = { 4 }
myCell =
    {[   1]}    {[          4]}    {[     3]}
    {'text'}    {5×10×2 double}   {3×1 cell}


>> myCell{1,3} = [ 4  5 ]
myCell = 2×3 cell array

    {[   1]}    {[          4 ]}    {[  4 5 ]}
    {'text'}    {5×10×2 double}    {3×1 cell}
```

Cells array - access

```
>> myCell = {1, 2, 3; 'text', rand(5,10,2), {11; 22; 33}}
myCell =  {[   1]}    {[          2]}    {[      3]}
          {'text'}    {5×10×2 double}    {3×1 cell}


>> myCell {3} = {}

myCell =  2×3 cell array

    {[   1]}    {0×0    cell  }    {[     3]}
    {'text'}    {5×10×2 double}    {3×1 cell}


>> myCell(1) = []

myCell =  1×5 cell array

    {'text'}    {[4]}    {5×10×2 double}    {[3]}    {3×1 cell}
```

Δ efface la cellule et non pas son contenu

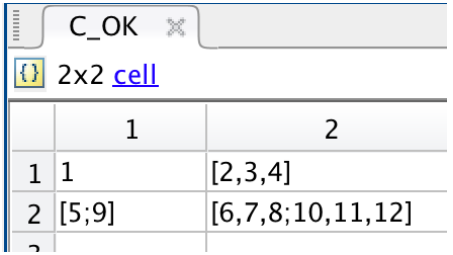
-> le cell array change de dimension

Cells array <-> matrices

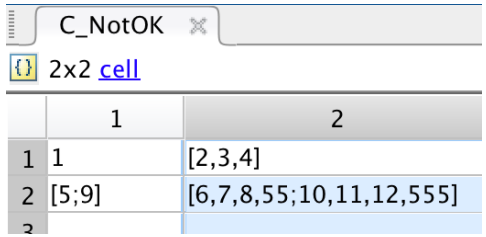
Il est possible de convertir un *cell array* en une matrice si:

- Les dimensions sont cohérentes
- Les éléments du *cell array* sont du même type

```
>> C_OK = {[1], [2 3 4]; [5; 9], [6 7 8; 10 11 12]};
>> A = cell2mat(C_OK)
A =  1     2     3     4
     5     6     7     8
     9    10    11    12
```

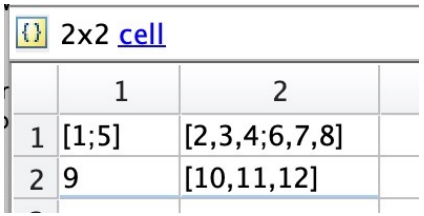



```
>> C_NotOK = {[1], [2 3 4]; [5; 9], [6 7 8 55; 10 11 12 555]}
>> B = cell2mat(C_NotOK)
Error using cat
Dimensions of matrices being concatenated are not consistent.
```



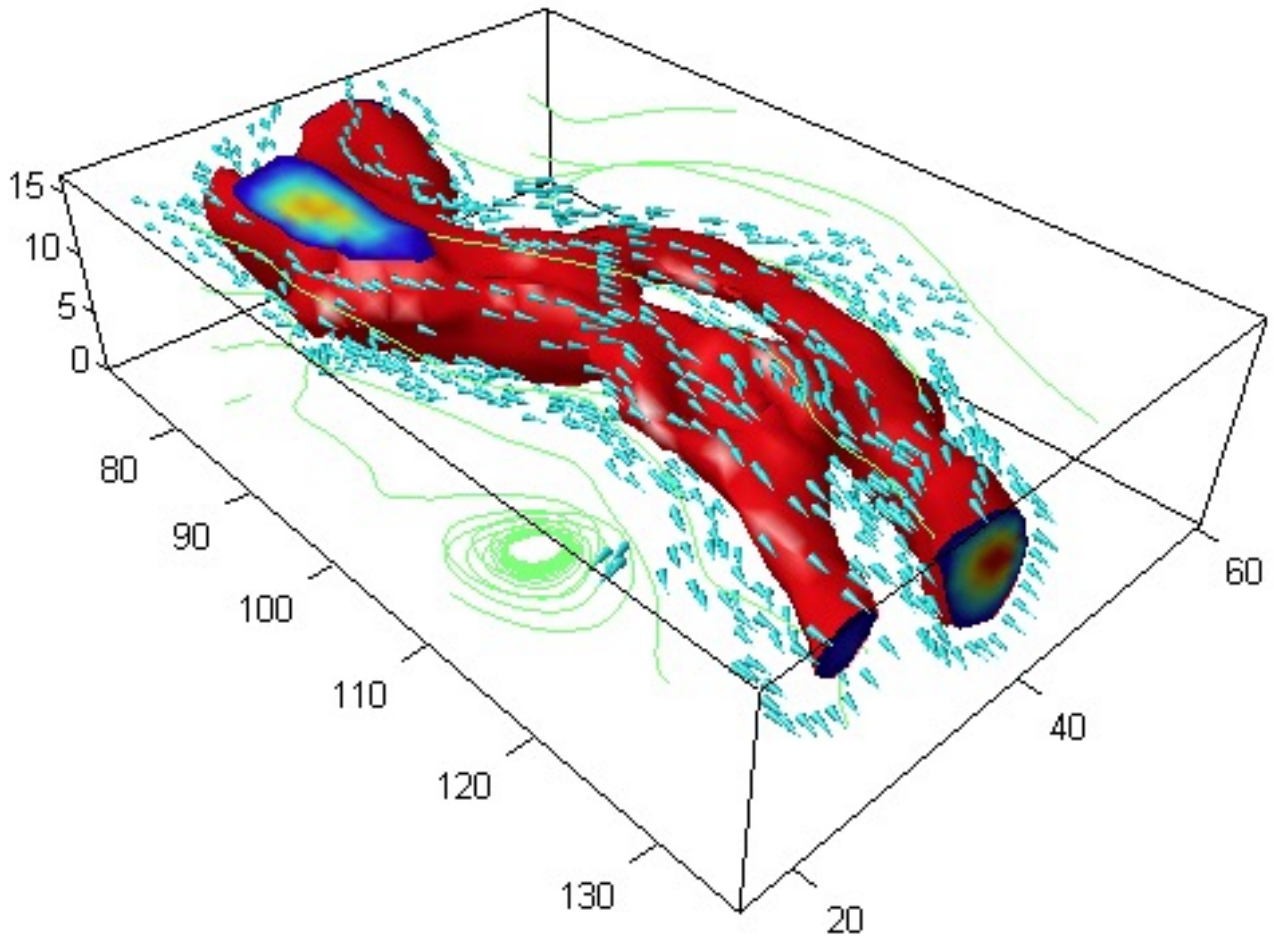
Et vice-versa convertir une matrice en *cell array* avec `mat2cell()`

```
>> C = mat2cell(A, [2 1], [1 3] )

C = 2×2 cell array
    {2×1 double}    {2×3 double}
    {[       9]}    {[10 11 12]}
```



- Demos cell array

3D & parametric graphics



Surface 3D

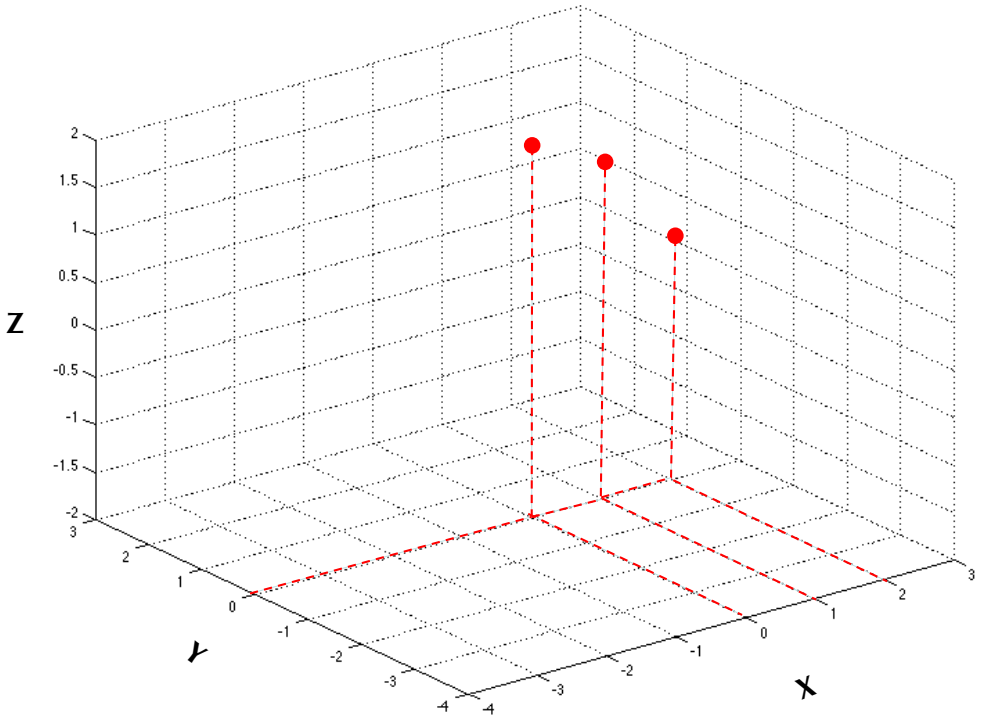
- Une grille XY discrétise l'espace de travail, elle est représentée par 2 matrices X et Y
- La coordonnée Z est calculée pour chaque points XY de la griller

  ex: $Z_{xy} = \cos(x) + \cos(y)$

  $Z_{0,0} = \cos(0) + \cos(0) = 2$

  $Z_{1,0} = \cos(1) + \cos(0) = 1.54$

  $Z_{2,0} = \cos(2) + \cos(0) = 0.58$
- Les points Z sont liées entre eux pour former une surface fillaire. Les points sont lier entre eux en fonction du choix du type de surfaces (pleine, courbes de niveau,etc)
- Les surfaces sont coloriées et éclairées
- La fonction d'affichage dessines les surface en tenant compte des faces cachées



Surface 3D

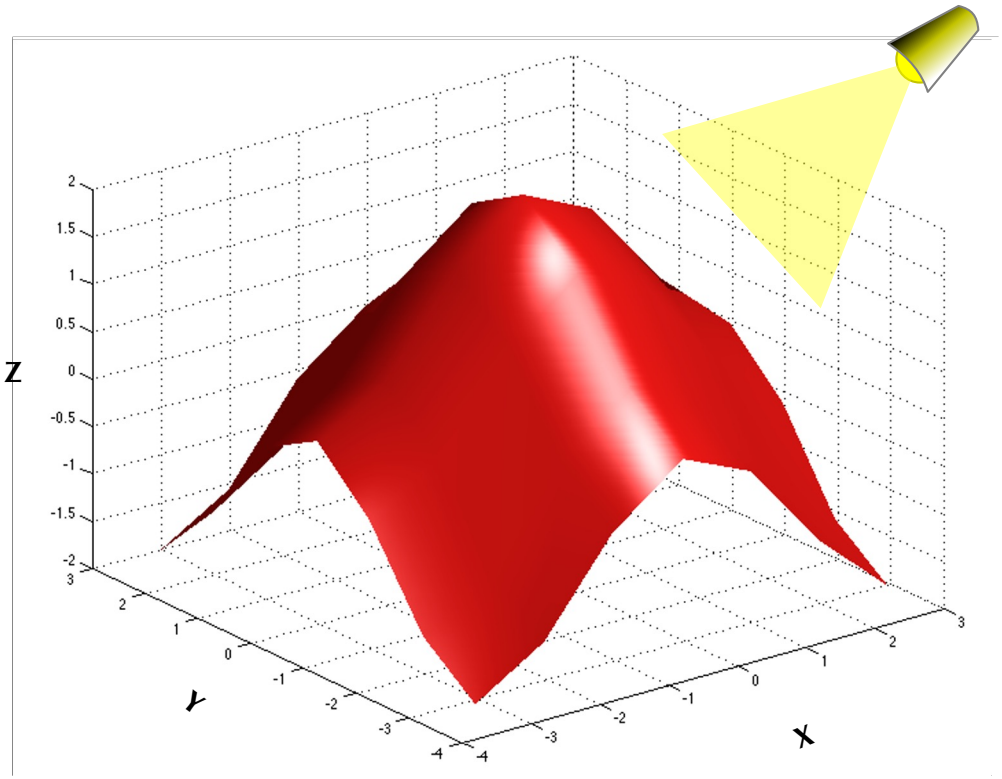
- Une grille XY discrétise l'espace de travail, elle est représentée par 2 matrices X et Y
- La coordonnée Z est calculée pour chaque points XY de la griller

  ex: $Z_{xy} = \cos(x) + \cos(y)$

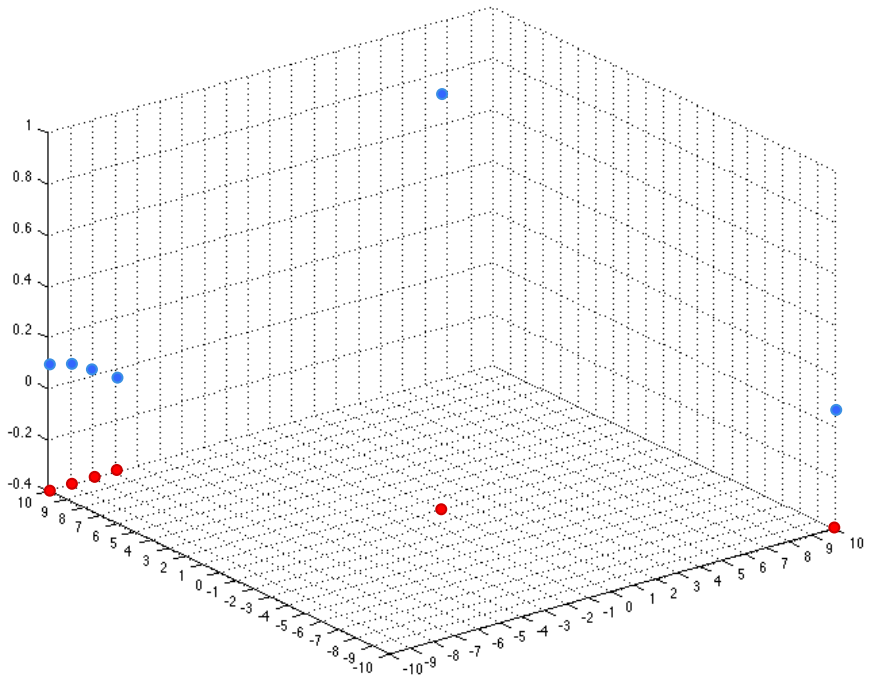
  $Z_{0,0} = \cos(0) + \cos(0) = 2$

  $Z_{1,0} = \cos(1) + \cos(0) = 1.54$

  $Z_{2,0} = \cos(2) + \cos(0) = 0.58$
- Les points Z sont liées entre eux pour former une surface fillaire. Les points sont lier entre eux en fonction du choix du type de surfaces (pleine, courbes de niveau,etc)
- Les surfaces sont coloriées et éclairées
- La fonction d'affichage dessines les surface en tenant compte des faces cachées

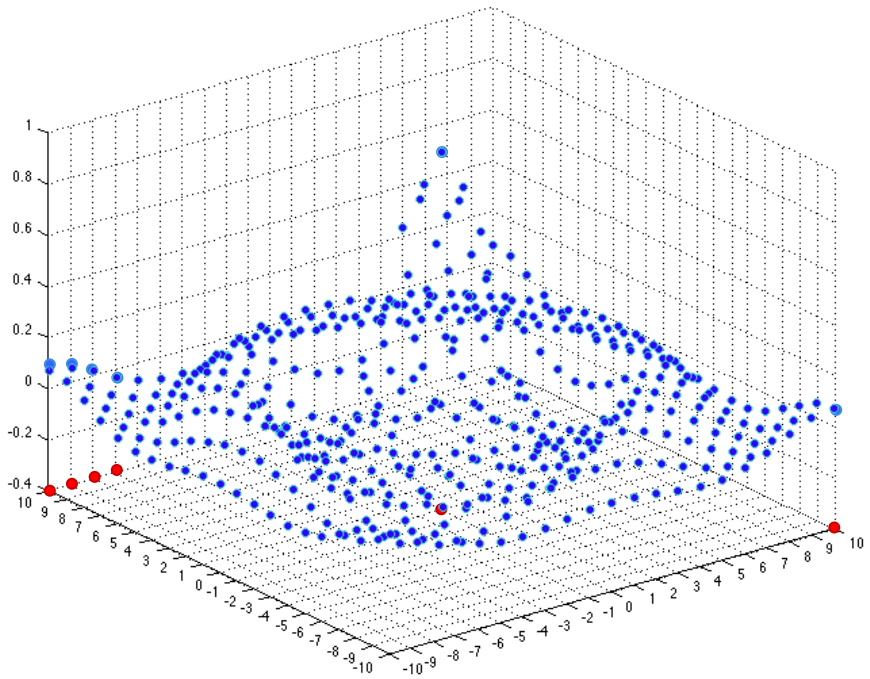


Surface 3D



```
for l=-10:10
  for c=-10:10
      R = sqrt(c^2+l^2)+eps;
      Z = sin(R)/R;
  end
end
```

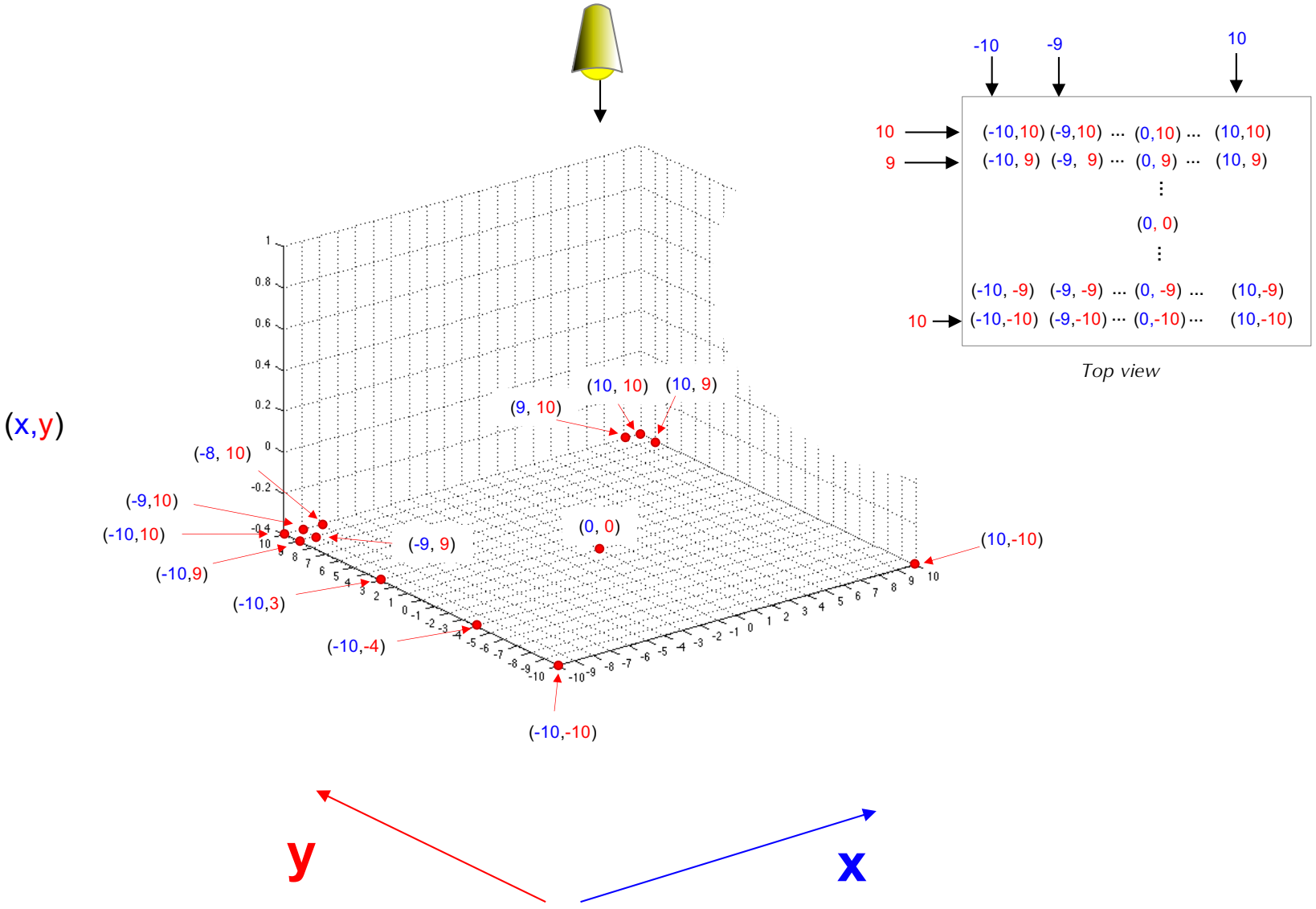
Surface 3D



```
for l=-10:10
  for c=-10:10
      R = sqrt(c^2+l^2)+eps;
      Z = sin(R)/R; % Z scalar
  end
end
```

2.45 [ms]

Surface 3D - vectorization



Surface 3D - vectorization

$$z = f\begin{pmatrix} (-10,10)\ (-9,10)\ \cdots\ (0,10)\ \cdots\ (10,10) \\ (-10,\ 9)\ \ (-9,\ \ 9)\ \cdots\ (0,\ 9)\ \cdots\ (10,\ 9) \\ \vdots \\ (0,\ 0) \\ \vdots \\ (-10,\ -9)\ \ (-9,\ -9)\ \cdots\ (0,\ -9)\ \cdots\ \ (10,-9) \\ (-10,-10)\ \ (-9,-10)\cdots\ (0,-10)\cdots\ \ (10,-10) \end{pmatrix}$$

Top view

Vectorization ☺

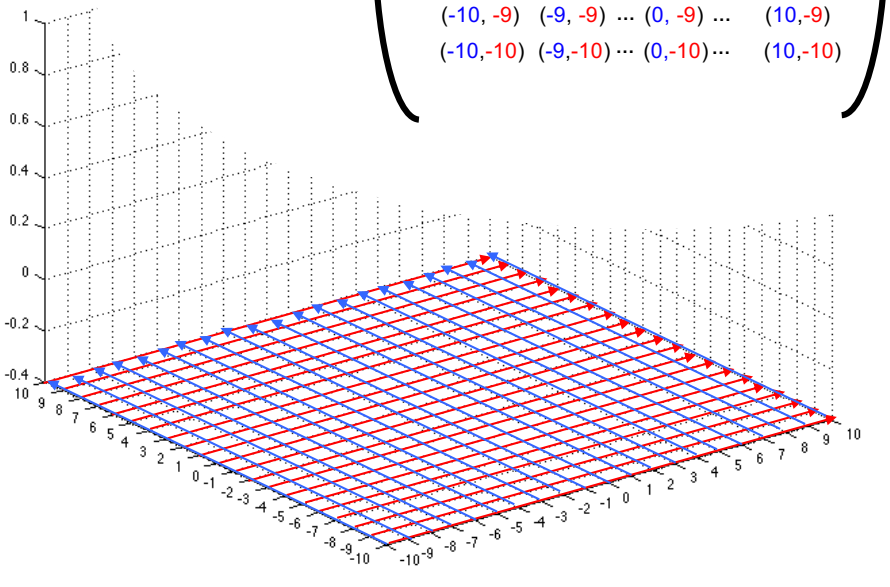
Surface 3D - vectorization

```
X= [
      -10, -9 … 10 ;
      -10, -9 … 10 ;
             …
      -10, -9 … 10 ;
    ]
```

```
Y= [
       10,  10 … 10 ;
        9,   9 …  9 ;
             …
     -10, -10 … -10 ;
    ]
```

$$z = f\begin{pmatrix} (-10,10)\ (-9,10)\ \cdots\ (0,10)\ \cdots\ (10,10) \\ (-10,\ 9)\ \ (-9,\ 9)\ \cdots\ (0,\ 9)\ \cdots\ (10,\ 9) \\ \vdots \\ (0,\ 0) \\ \vdots \\ (-10,-9)\ \ (-9,-9)\ \cdots\ (0,-9)\ \cdots\ (10,-9) \\ (-10,-10)\ (-9,-10)\cdots\ (0,-10)\cdots\ (10,-10) \end{pmatrix}$$

Vectorization ☺



```
[X Y]= meshgrid(-10:10,-10:10)
```

Vectorization 2D with `meshgrid()`

`Meshgrid()` duplique les vecteurs vx et vy pour produire deux tableaux **X** et **Y** qui contiennent les copies des vecteur vx et vy. Le nombre de répétitions des vecteurs est défini par la taille de l'autre vecteur.

`Meshgrid()` fonctionne de la même manière en 3D

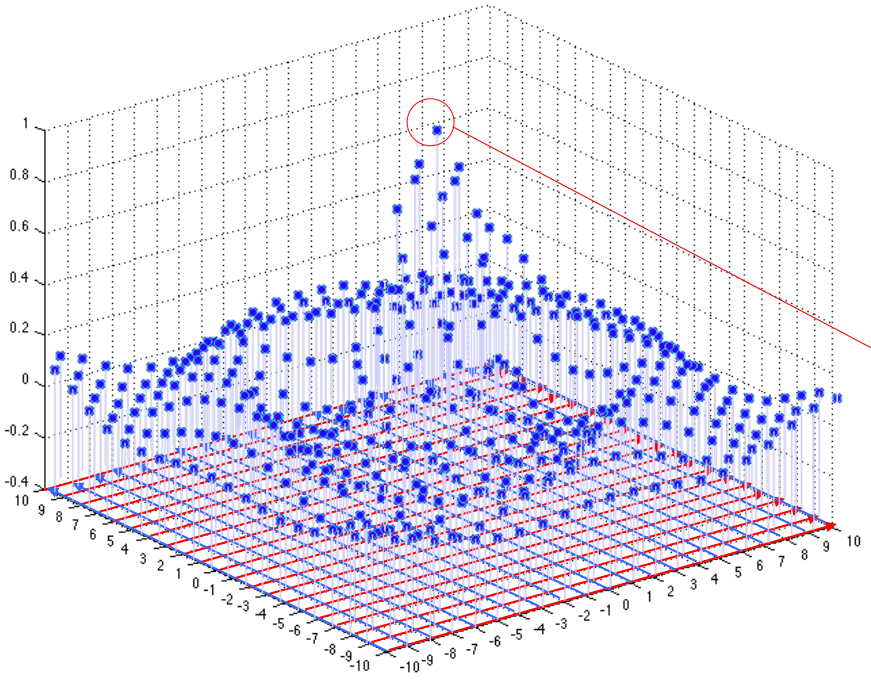
```
vx =  1: 3
vy = 10:14
```

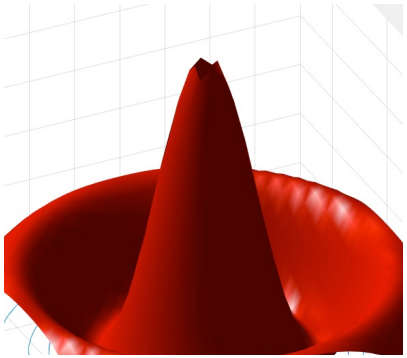
```
% no vectorization ☹
% -> slow & inefficient !

for l=1:length(vx)
   for c=1:length(vy)
       z(l,c) =vx(l)*vy(c);
   end
end
```

```
>>[X,Y] = meshgrid(vx,vy)
X =
     1     2     3
     1     2     3
     1     2     3
     1     2     3
     1     2     3
Y =
    10    10    10
    11    11    11
    12    12    12
    13    13    13
    14    14    14

 Z = X.*Y
```

Vectorization ☺

Surface 3D





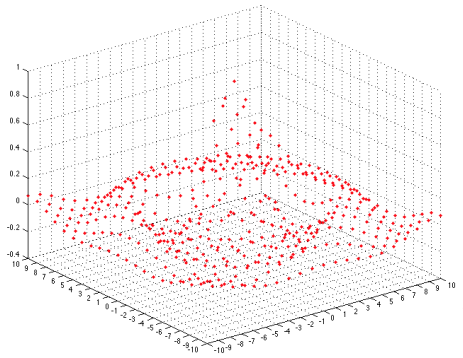
eps *?*

```
[X Y]= meshgrid(-10:10)
R = sqrt(X.^2+Y.^2)+eps;
Z = sin(R)./R;
```

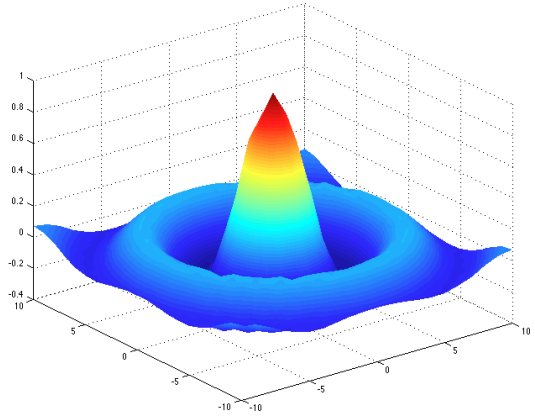
0.043 [ms]

Vectorization ☺

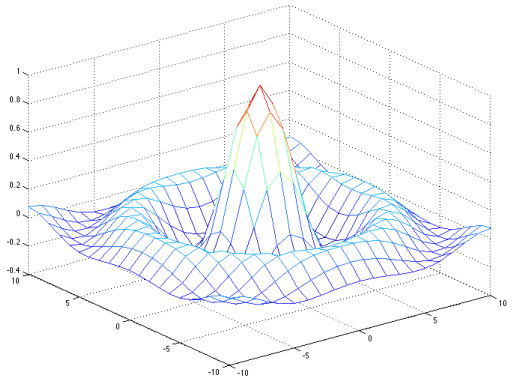
Surface 3D



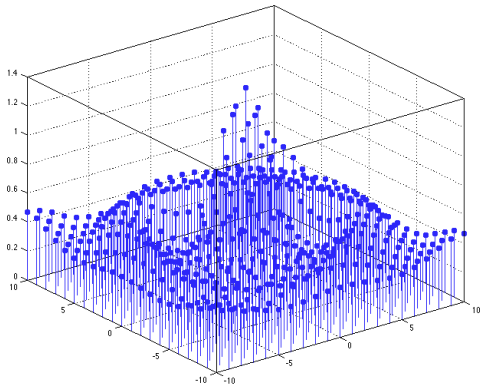
surf(X,Y,Z)



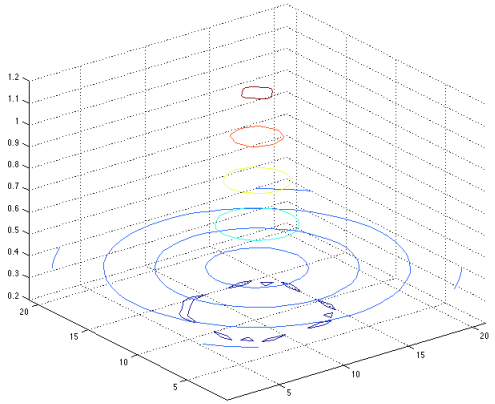
mesh(X,Y,Z)



stem3(X,Y,Z)

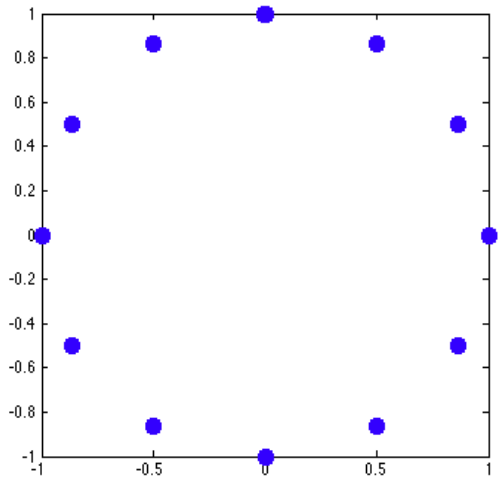


contour(X,Y,Z)



- Demos surfaces 3D

Courbe paramétrique 2D



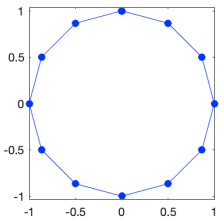
Horloge

Pourquoi 13 points?

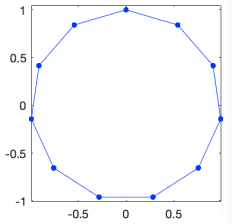
Paramètre: α

```
α = linspace(0,2*pi,13) % 1 tour
x = sin(α);
y = cos(α);
plot(x,y,'.');
```

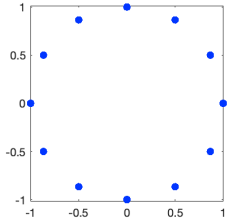
vectorization



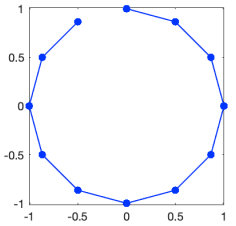
```
linspace(0,2*pi,13)
…
plot(x,y,'.-')
```



```
linspace(0,2*pi,12)
…
plot(x,y,'.-')
```

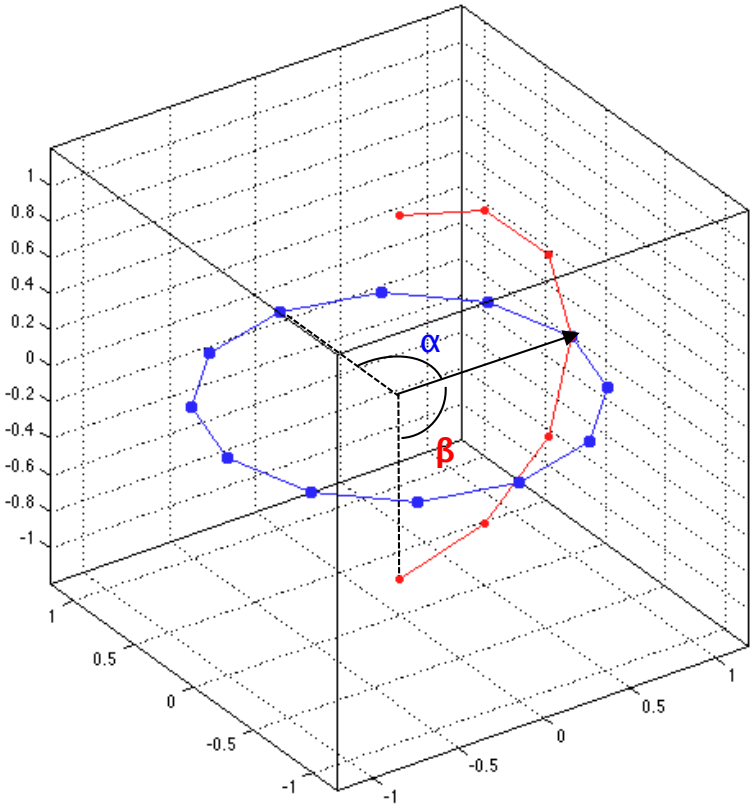


```
linspace(0,2*pi-pi/6,12)
…
plot(x,y,'.')
```



```
linspace(0,2*pi-pi/6,12)
…
plot(x,y,'.-')
```

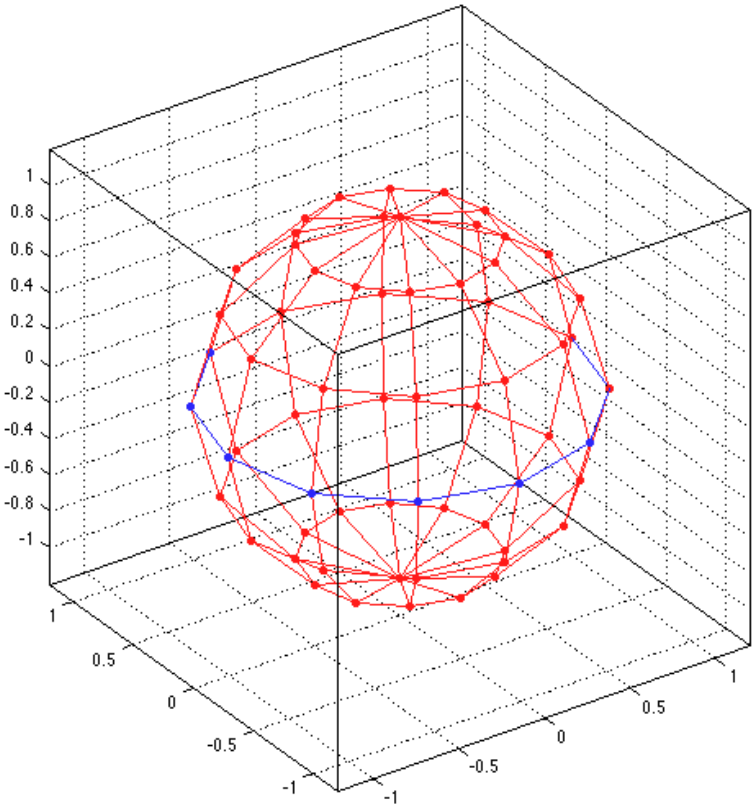
Surface paramétrique 3D



Paramètres: **α**, **β**

```
α = linspace(0,2*pi,13) % 1 tour
β = linspace(0,pi,7) % 1/2 tour
```

Surface paramétrique 3D



Paramètres: α, β

```
αv = linspace(0,2*pi,13) % 1 tour
βv = linspace(0,pi,7) % 1/2 tour

[α, β] = meshgrid(βv, αv);

X = sin(α).*cos(β);
Y = sin(α).*sin(β);
Z = cos(α);

mesh(x,y,z)
```

Vectorization

Surface paramétriques 3D

Dans le cas d'une surface paramétrique, les points de la surface ne sont plus calculés à partir de la grille XY, mais à partir d'autres paramètes.

Ex. une sphère calculée fonction des angles φ et θ.

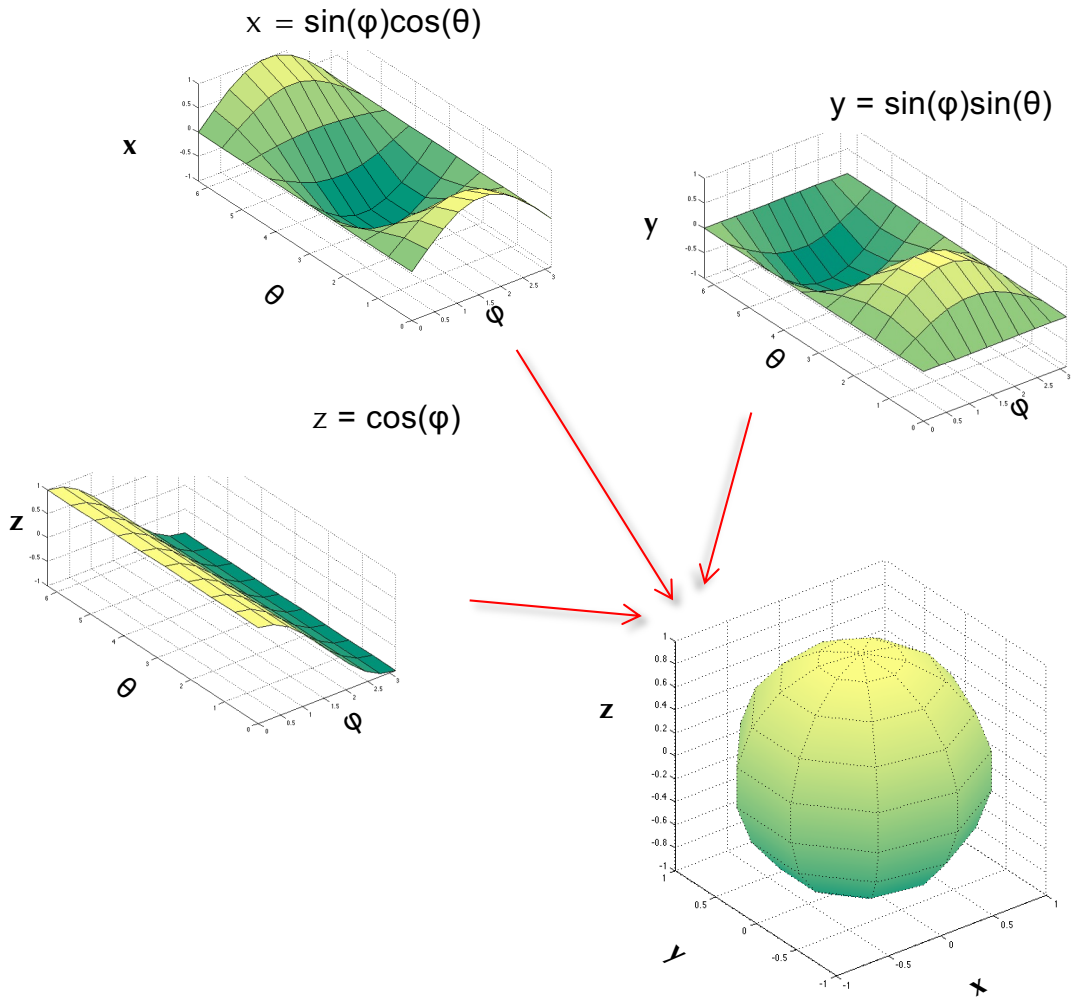
Les coordonnées XYZ des points de la surface paramétrique sont calculés pour chaque point φ-θ de la grille

ex: $x = \sin(\varphi)\cos(\theta)$

$y = \sin(\varphi)\sin(\theta)$

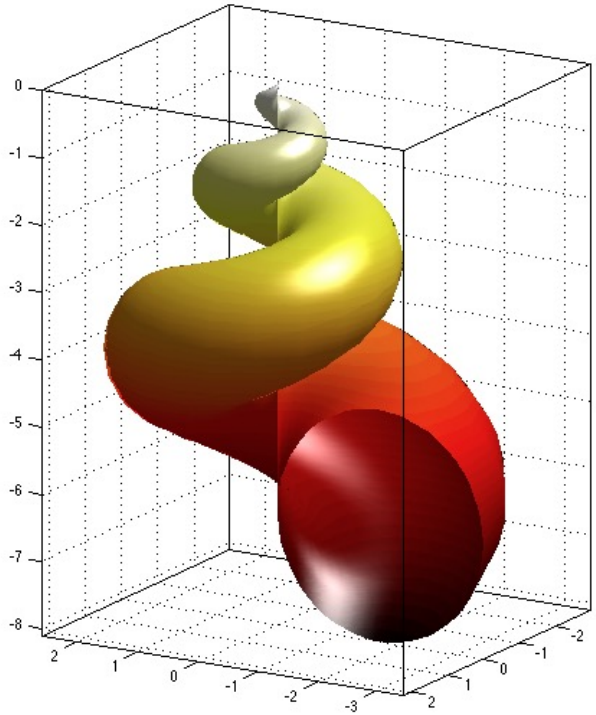
$z = \cos(\varphi)$

Il en résulte 3 matrices XYZ définissant les coordonnées de point de la sphère.



Surface paramétriques 3D

```
figure;                    % new figure
u=linspace(0,6*pi,60);   % uv parametrisation
v=linspace(0,2*pi,60);
[u,v]=meshgrid(u,v);
                           % [X Y Z] = f(u,v)
x=2*(1-exp(u/(6*pi))).*cos(u).*cos(v/2).^2;
y=2*(-1+exp(u/(6*pi))).*sin(u).*cos(v/2).^2;
z=1-exp(u/(3*pi))-sin(v)+exp(u/(6*pi)).*sin(v);
                           % draw surf (X Y Z)
surf(x,y,z, 'FaceColor','interp',...
            'EdgeColor','none',...
            'FaceLighting','phong')
camlight left
colormap hot
view(169,12)
axis equal
box on
figure(gcf)
```



$$x = 2\left[1 - e^{u/(6\pi)}\right]\cos u \cos^2\left(\frac{v}{2}\right)$$

$$y = 2\left[-1 + e^{u/(6\pi)}\right]\sin u \cos^2\left(\frac{v}{2}\right)$$

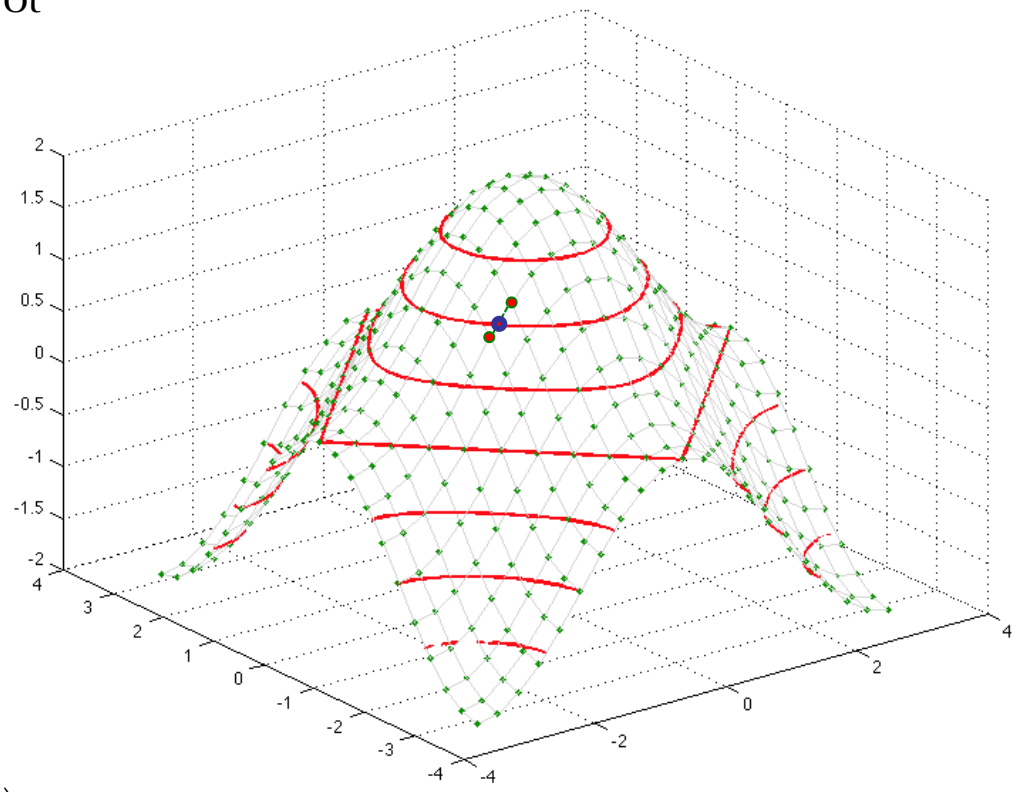
$$z = 1 - e^{u/(3\pi)} - \sin v + e^{u/(6\pi)}\sin v$$

- Démos courbes params

Interpolation

L'interpolation est une opération mathèmatique permettant le calcul de points intermédiaires entre deux points donnés. L'interpolation la plus simple entre deux points est la ligne droite ou interpolation linéaire.

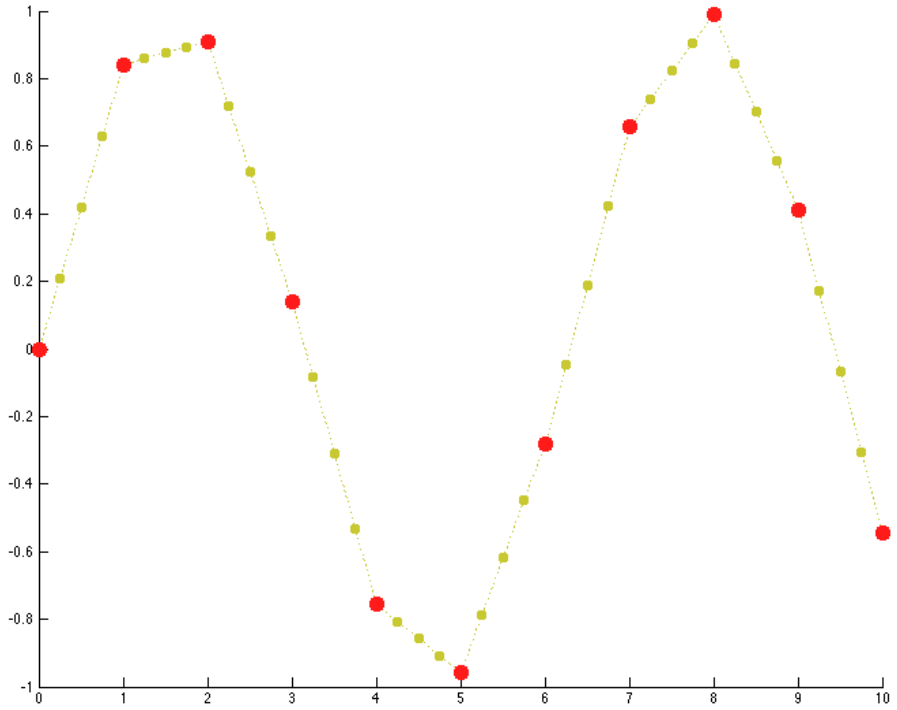
Ex. Contour plot



$z = \cos(x) + \cos(y)$

```
contour(x,y,z)
```

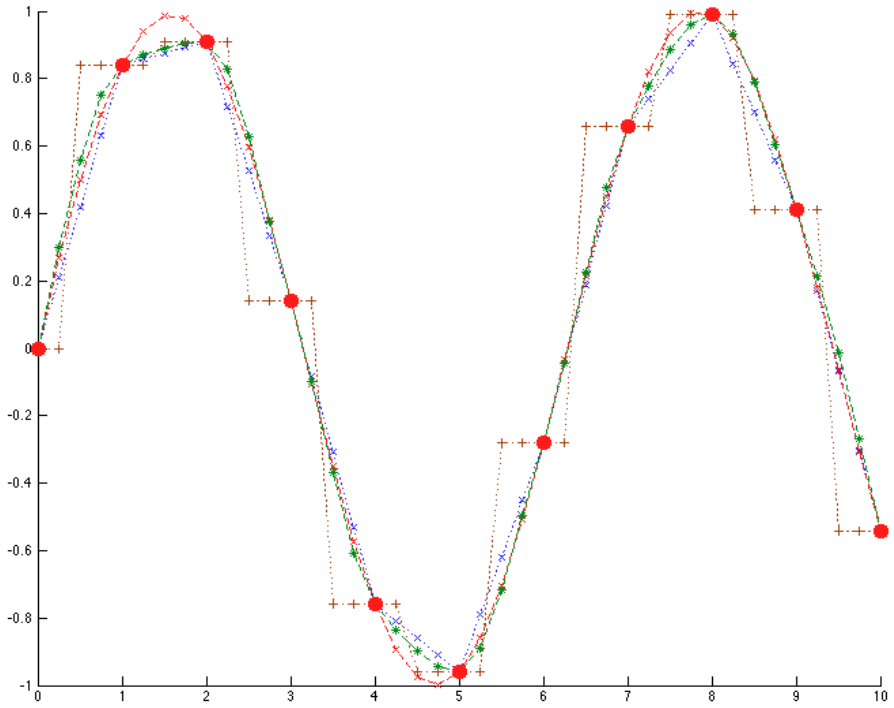
Interpolation




```
x = 0:10;          % 10 pts
y = sin(x);

xi = 0:.25:10;     % 40 pts

yi = interp1(x,y,xi,'linear');

plot(x,y,'or', xi,yi,'-.sy');
```

`Yi = interp1(X,V,Xq,Method)` retourne les valeurs interpolées (Yi) aux points demandés (Xq) en se basant sur la function définie par V=F(X). Method défini comment les points sont interpolés.

Interpolation

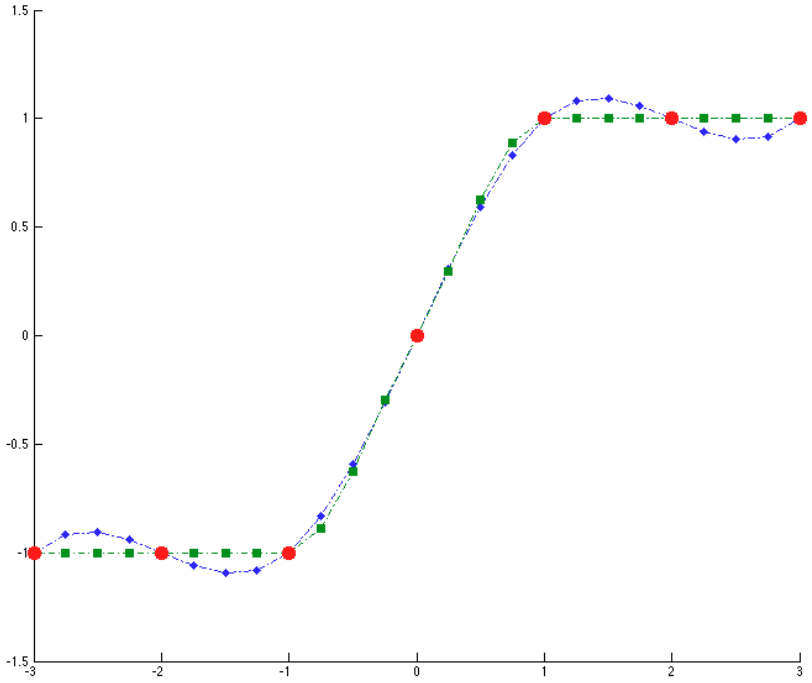


Linear
Spline
Cubic
Nearest

```
x = 0:10;          % 10 pts
y = sin(x);
xi = 0:.25:10;     % 40 pts
yi = interp1(x,y,xi,'cubic');
plot(x,y,'or', xi,yi,'-.xr');
yi = interp1(x,y,xi,'spline');
plot(x,y,'or', xi,yi,'-.*g');
yi = interp1(x,y,xi,'nearest');
plot(x,y,'or', xi,yi,'-.+r');
```

Interpolation

- L'interpolation par une ***spline*** produit un resultat *smooth,* elle est plus précise si les données à interpoler sont *smooth*
- L'interpolation par une ***cubic*** a moins de dépassement *(overshoot)* et moins d'oscillations si les données à interpoler ne sont pas *smooth*

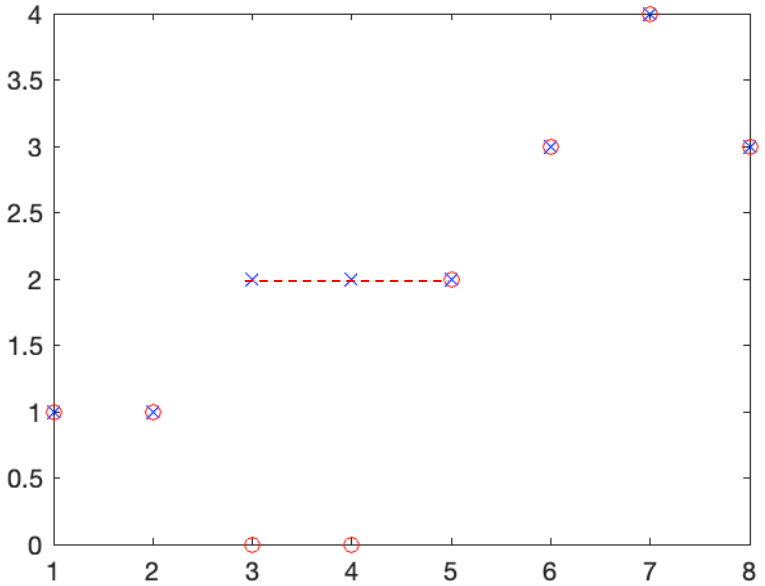



```
x  = -3:3;
y  = [-1 -1 -1 0 1 1 1];
xi = -3:.25:3
ys = interp1(x,y,xi,'spline');
yc = interp1(x,y,xi,'makima');
plot(x,y,'or',  ...
     xi, ys,'-.xb',...
     xi, yc,'-.sg');
```

`makima:`
Modified Akima cubic Hermite interpolation

Interpolation – 'next'

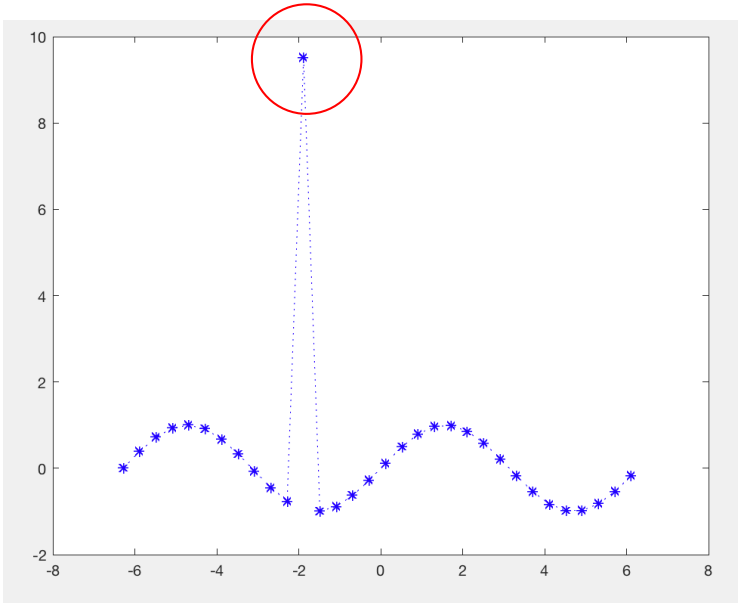
- L'interpolation avec 'next' est similaire à 'nearest'. Les valeurs à interpoler auront la valeur de la prochaine valeur de référence.
- Dans l'exemple ci-dessous les y à '0', soit y(3) et y(4)sont à remplacer par la valeur du prochain y de référence, dans notre cas, y(5) = 2.



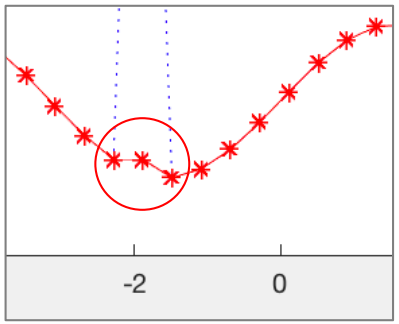

```
% missing values in y are indicated by'0'
x  = [ 1 2 3 4 5 6 7 8 ];
y  = [ 1 1 0 0 2 3 4 3 ];
% xi pos to be interpolated
xi = [     3 4          ];
% xv valid pos
xv = [ 1 2     5 6 7 8 ];

yi= interp1([0 xv],y([1 xv]),xi,'next');

Yn = y;
Yn(xi)=yi;

plot(x,y,'or', x,Yn,'xb');
```

Interpolation

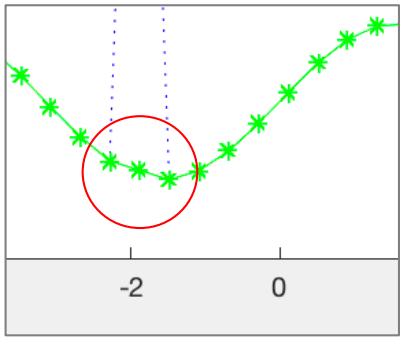
Utilisation de l'interpolation pour replacer un *outlier,* i.e. point aberrant du par exemple à une erreur de mesure.



previous



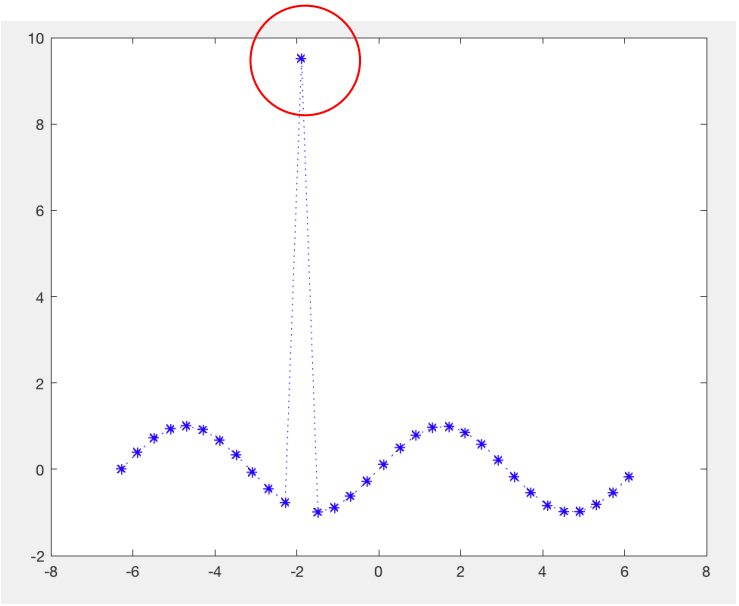
linear

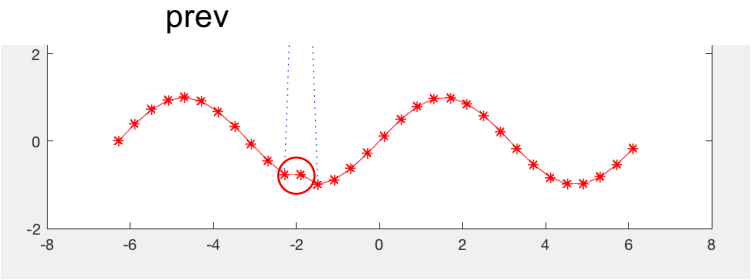


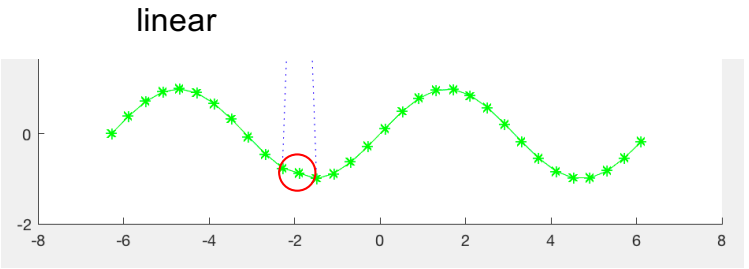
Outliers

Utilisation de l'interpolation pour replacer un *outlier*, i.e. point aberrant du par exemple à une erreur de mesure.

```
X=-2*pi:0.4:2*pi;
Y=sin(X);
Y(12)=10*abs(Y(12)) % create outlier
plot(X,Y,':*b');
% Find outliers pt/segments i.e. segments outside moving
median over 5 values
O=isoutlier(Y,'movmedian',5);
Oid=find(O); % get outliers indexes

%% 1) replace outlier pt with coordinate of previous pts
% X(Oid)=X(Oid-1); % this option will superpose 2 pts
Y1=Y;
Y1(Oid)=Y1(Oid-1);
plot(X,Y1,'-*r');

%% 2) interpolate missing pt
Y2=Y;
idx2= [Oid-1,Oid+1];
idx3= [Oid-1,Oid,Oid+1];
Y2(idx3) = interp1(X(idx2),Y(idx2),X(idx3),'linear');
plot(X,Y2,'-*g');
```



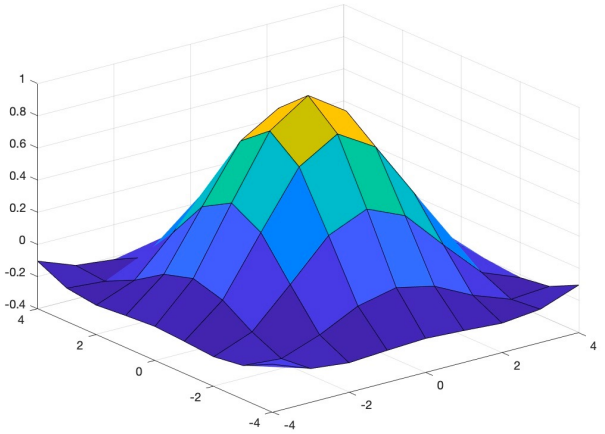




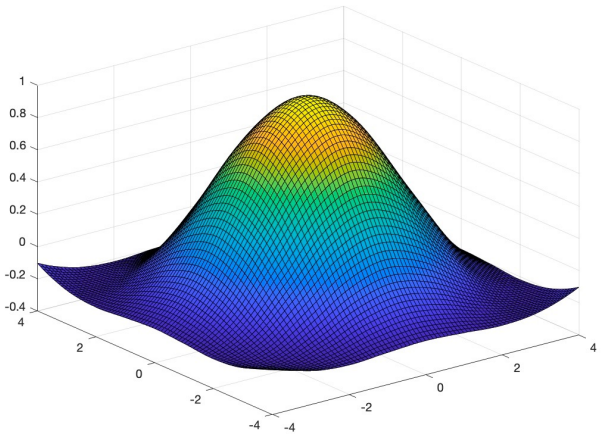
Interpolation 2D

L'interpolation 2D **interp2()** fonctionne de manière similaire à la version 1D.

```
% coarse figure
[X,Y] = meshgrid(-4:1:4);
R = sqrt(X.^2 + Y.^2)+eps;
Z = sin(R)./R ;
surf(X,Y,Z);
```



```
% interpolate coarse figure
[Xq,Yq] = meshgrid(-4:0.1:4);
Vq = interp2(X,Y,Z,Xq,Yq,'cubic',0);
surf(Xq,Yq,Vq)
```



- Demo interpolations

Polynômes

Matlab représente un polynome sous la forme d'un vecteur de coéfficiants, ex.

$$\boldsymbol{a}x^4 + \boldsymbol{b}x^3 + \boldsymbol{c}x^2 + \boldsymbol{d}x + \boldsymbol{e} \quad \Rightarrow \quad \texttt{p = [a b c d e]}$$

Calcul les racines de ***p***: `roots(p)`

Calcul un polynome à partir des racines ***r***: `poly(r)`

Evalue ***p*** aux points défini par ***x*** `polyval(p,x)`

Exemples

```
>> p = [1 -6 -72 -27];
>> r = roots(p)
r =
   12.1229
   -5.7345
   -0.3884

>> poly(r)
ans =
    1   -6  -72  -27
```

```
>> polyval(p,[0 1])
ans =
   -27  -104

>> polyval(p,r)
ans =
1.0e-12 *
      -0.5862
       0.1172
       0.0071
```

Small but not '0', why?

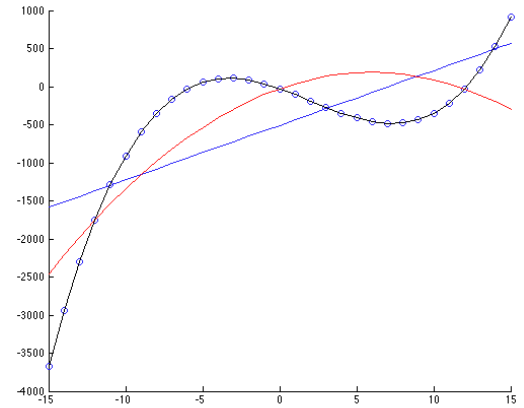
`eps` = 2.2204 e^{-16}

Polynômes et Fit

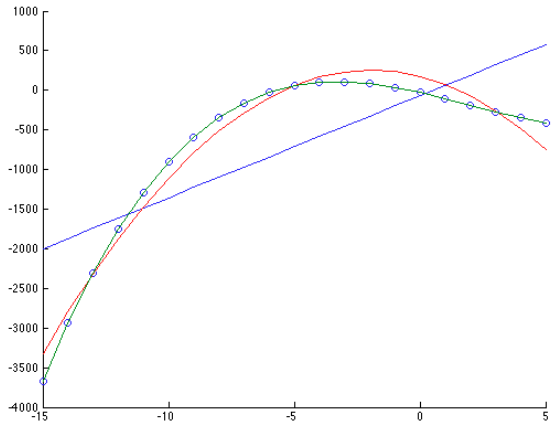
`Polyfit` ajuste un polynomes sur des données en minimisant l'erreur, ex.

```
>> p = [1 -6 -72 -27];
>> x = -15:15 ;
>> y= polyval(p,x);
>> plot(x,y);

>> p1 = polyfit(x,y,1);
>> plot(x,polyval(p1,x),'b');

>> p2 = polyfit(x,y,2);
>> plot(x,polyval(p2,x),'r');

>> p3 = polyfit(x,y,3);
>> plot(x,polyval(p3,x),'g');

>> p4 = polyfit(x,y,4);
>> plot(x,polyval(p4,x),'g');
```

degré



-15 : 15



-15 : 5

Curve Fitting

- L'ajustement de courbe ou *curve fitting,* cherche à faire passer la meilleure courbe possible, i.e. celle qui "suit" le mieux, des données expérimentales, entachées ou non d'erreurs de mesure.
- Plusieurs méthodes existent pour ajuster les paramètres de la courbe afin de choisir le meilleur *fit*: régression linéaire (moindre carré) ou non-linéaire (itérative, Levenberg-Marquardt) ou l'interpolation par des polynômes/*splines*.
- Le *curve fitting* défini une équation qui permet le calcul de n'importe quel point sur cette courbe. Il permet également de lisser (*smooth*) cette courbe.
- La commande `cftool` lance l'outil interactif de *curve/surfave fitting* de Matlab.

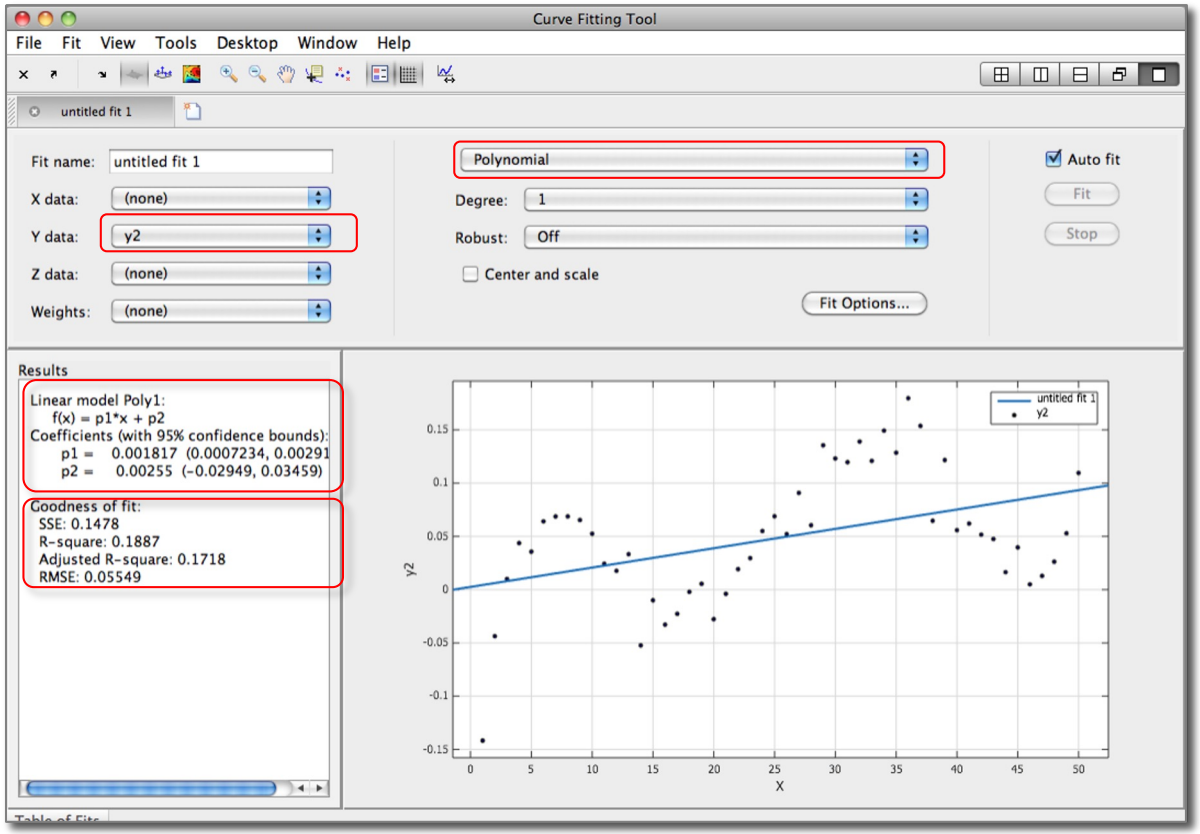
Fit interactif

Choix du type de courbes
et paramètres relatifs

Choix des données

Equation et paramètres

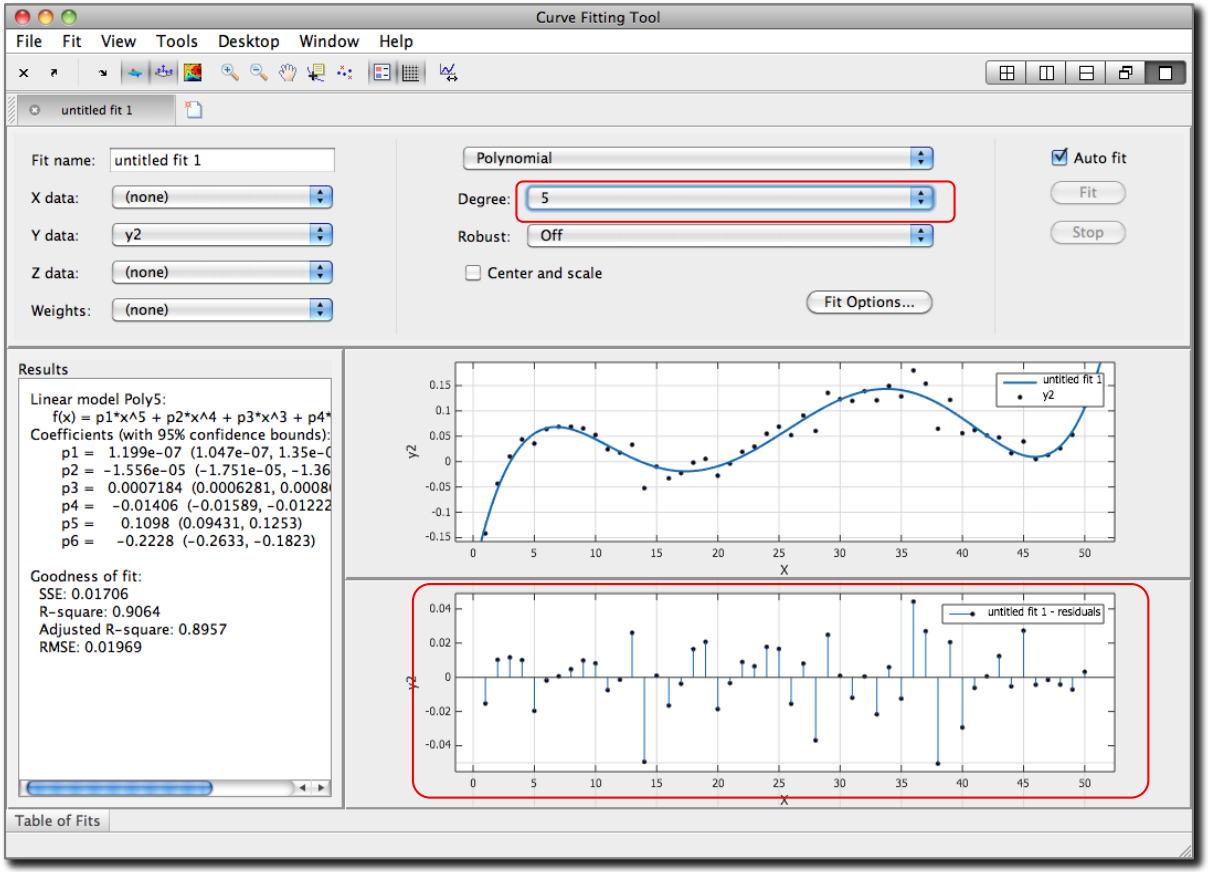
Qualité du fit



SSE: Sum of squares due to error
R2: Coefficient of determination
adjustedR2: Degree-of-freedom adjusted coefficient of determination
stdError: Root mean squared error (standard error)

Fit interactif

Degré du polynôme

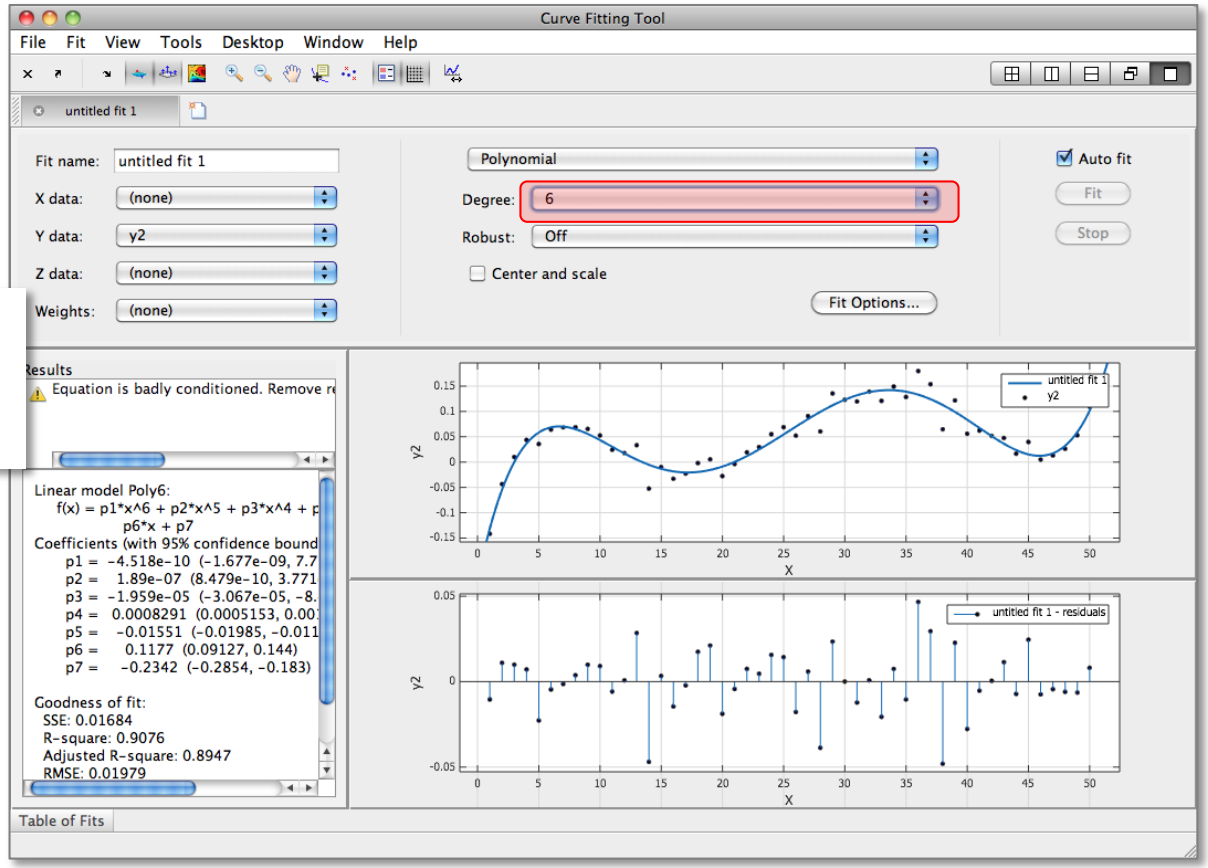


Résidu

Fit interactif

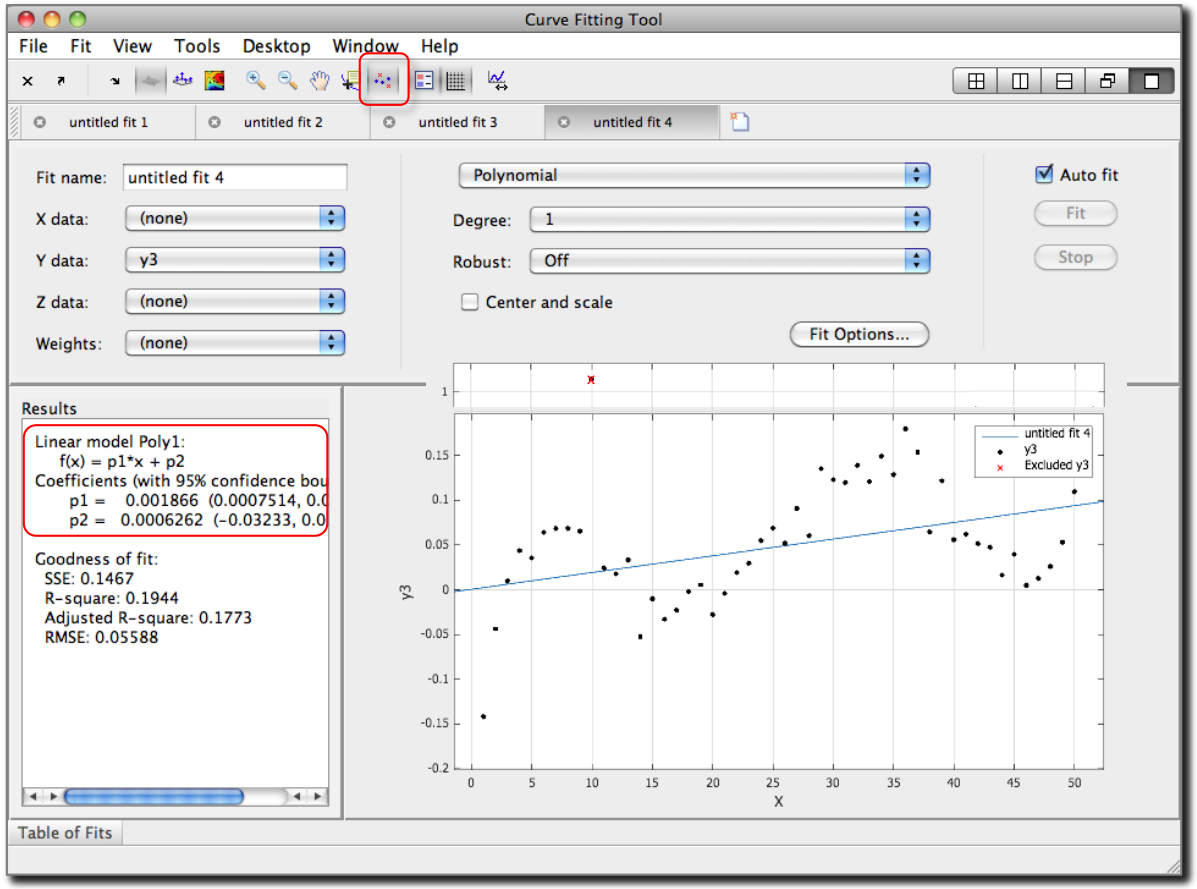
Degré du polynôme

Equation is badly conditioned. Remove repeated data points or try centering and scaling.



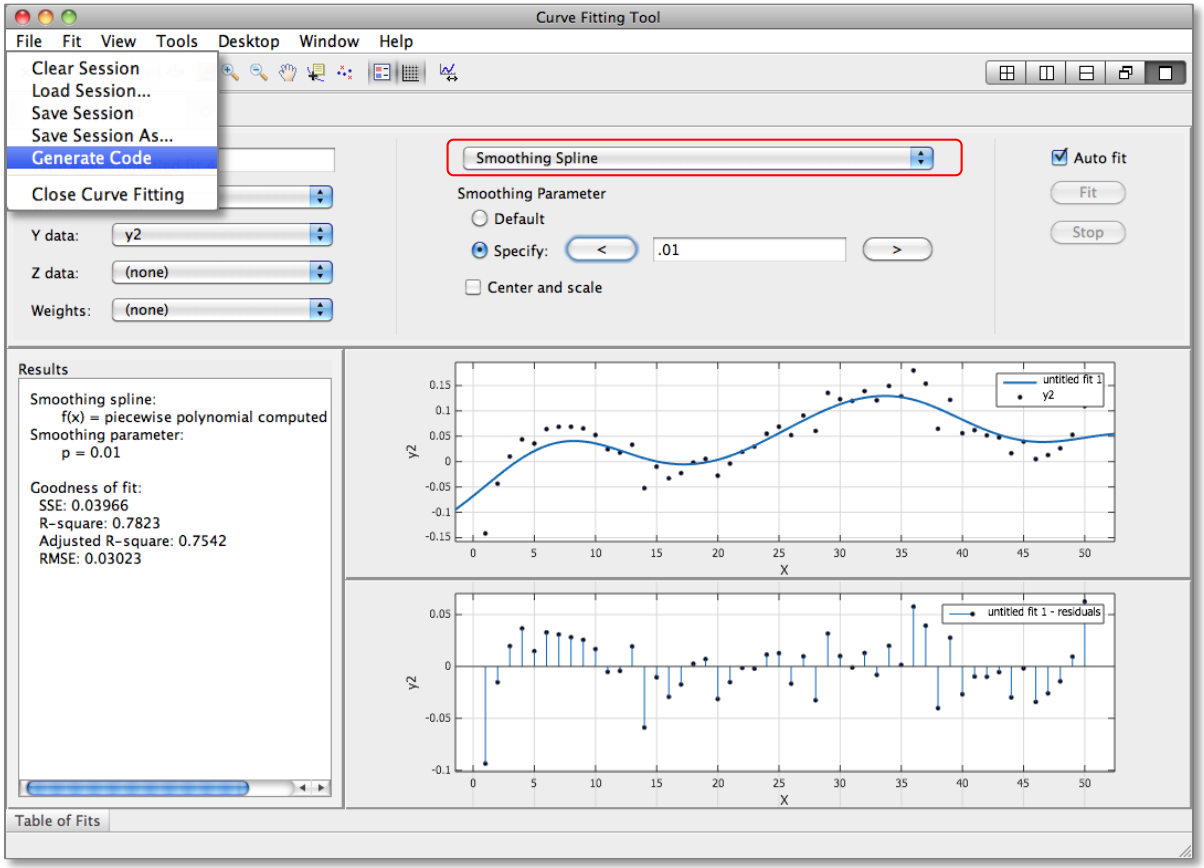
Fit interactif

Remove outliers

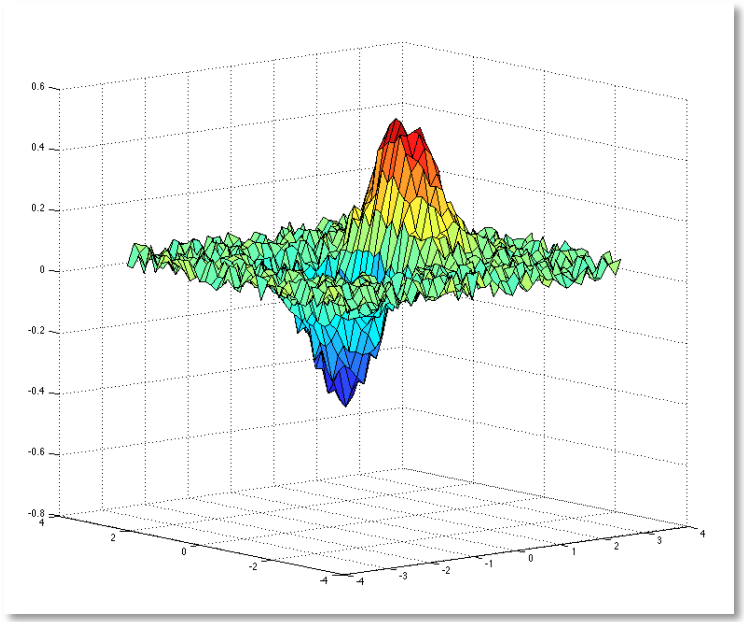


Fit interactif

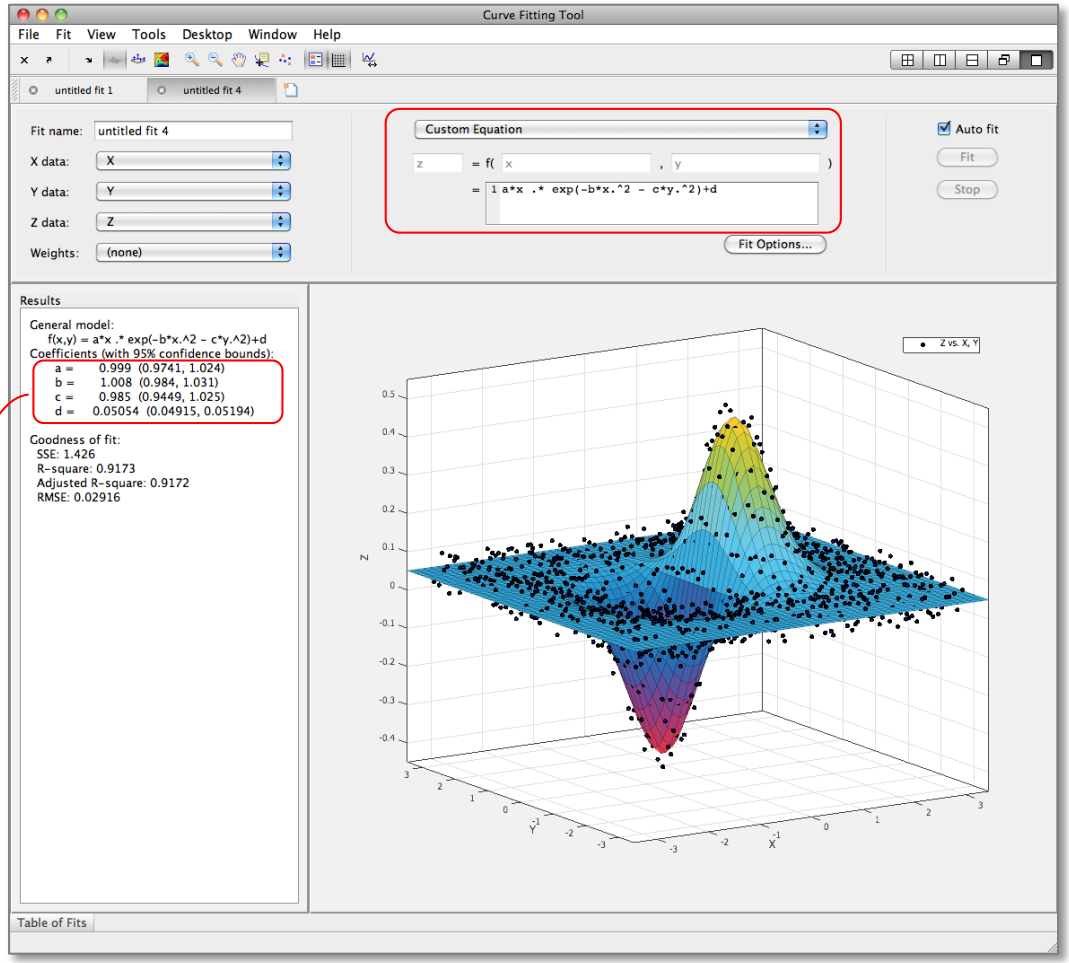
- Sauvegarde des *fits*
- Génération d'une fonction de fit avec les options choisies



Fit interactif - 3D



```
Z = X .* exp(-X.^2 - Y.^2)+0.1*rand(size(X));
```



Fonction `fit()`

De la même manière similaire au fit interactif, la fonction **`fit()`** permet d'ajuster une courbe/surface pour approximer des données expérimentales.

```
[fitObject, gof, output] = fit(x,y,z,fitType, fitOptions)
```

Il existe plusieurs types de fit prédéfinis, par ex.: `poly1, exp1, sin1, …`

Alternativement, il est possible de définir la fonction à ajuster, par ex.:

```
y = a * sin(b * x) + c
```

`fitOptions` permet de spécifier les nombreuses options du fit.

La fonction **`fit()`** retourne:

La courbe/surface ajustée dans **`fitObject`** fitobject

Et une indication de la qualité du fit dans **`gof`** *goodness of fit*

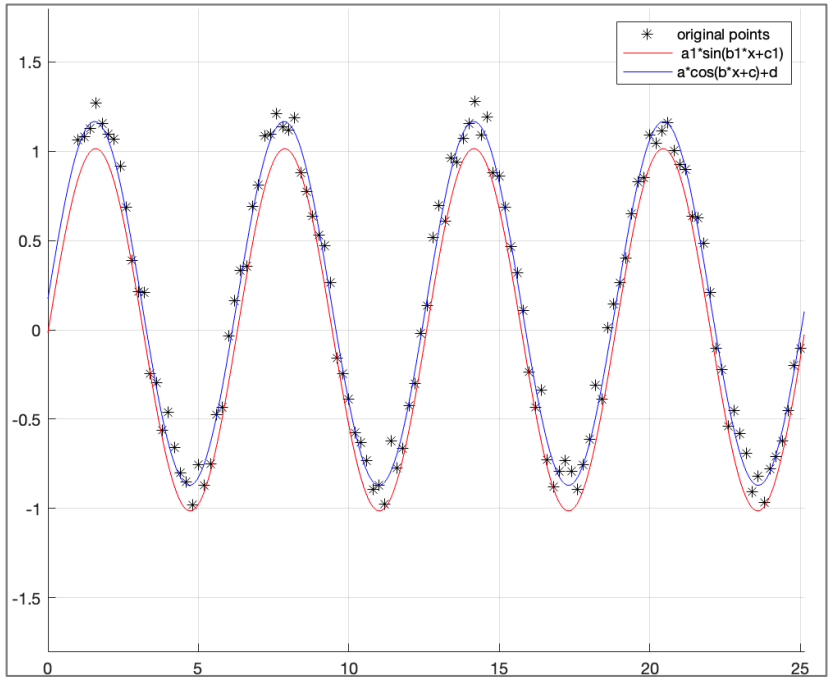
Des informations supplémentaires dans **`output`**

Fonction `fit()`

Exemple:

```
% fit a sinus
ft = fittype('sin1');

opts =
fitoptions('Method','NonlinearLeastSquares');

[fitResult, gof] = fit( x, y, ft, opts )

plot(fitResult);
```

```
% fit a given function
ft2 = fittype( 'a*cos(b*x+c)+d');

opts2 =
fitoptions('Method','NonlinearLeastSquares');

opts2.StartPoint = [0.5 0.9 0.6 1.7];

[fitResult2, gof2] = fit( x, y, ft2, opts2 )

plot(fitResult2, '-b');
```




```
fitResult =  General model Sin1:
     fitResult(x) =  a1*sin(b1*x+c1)
...
gof = sse: 3.4569
...
```

```
fitResult2 = General model:
     fitResult2(x) = a*cos(b*x+cd) + d
...
gof2 = sse: 0.7411
```

Goodness of fit – `gof`

Exemple:

```
[~, gof, output] = fit( x, y, 'poly4' )
```

```
gof =  struct with fields:
           sse: 1.3211e+03
       rsquare: 0.9587
           dfe: 35
     adjrsquare: 0.9540
          rmse: 6.1437
```

Matlab calcul différentes mesures statistiques de la qualité du fit en fonction de l'erreur entre les valeurs estimées via le *fit* et les valeurs réelles.

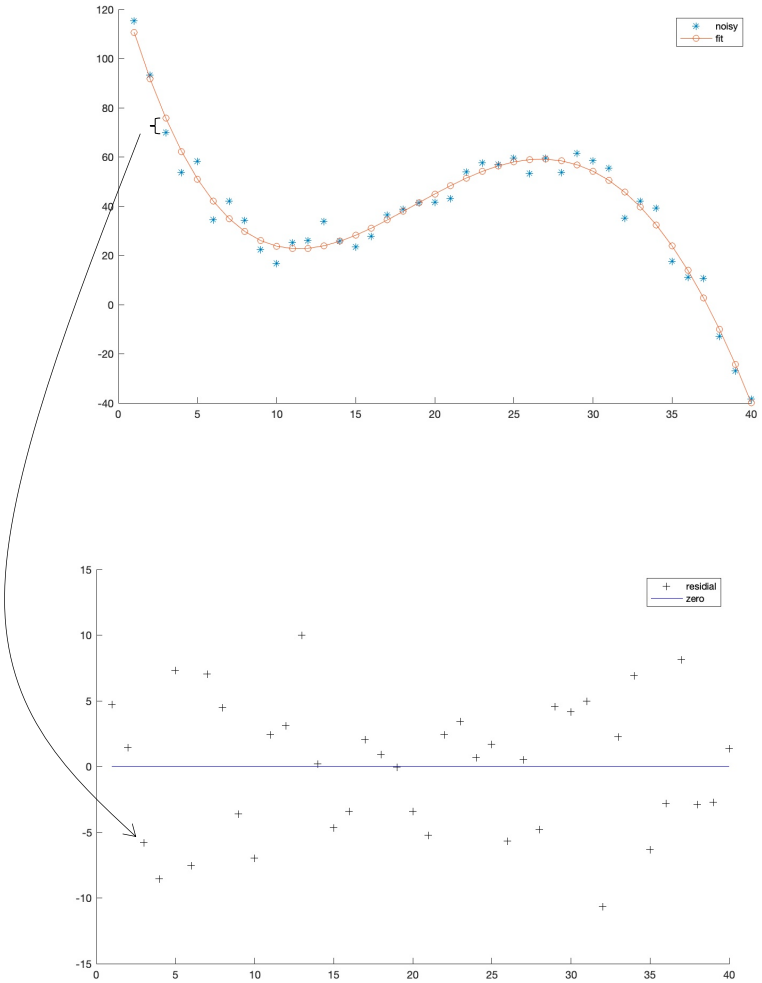
```
output = struct with fields:
        numobs: 40
      numparam: 5
     residuals: [40×1 double]
      Jacobian: [40×5 double]
      exitflag: 1
     algorithm: 'QR factorization and solve'
    iterations: 1
```

Et fourni des informations supplémentaires concernant le *fit*

Goodness of fit – `gof`

Exemple:

```
% initial polynom coef.
P = [4e-7 , -0.0003, 0.06 , -4, 110];
x = 1:5:200;

% evaluate poly at x[] and add some noise
yn = polyval(P,x) + 20*rand(1,length(x)) - 10;

% fit the noisy pts with a poly
[fitModel, gof, output] = fit(x',yn','poly4')

% evaluate fitted poly at x[]
yfit= feval(fitModel,x)';


% -- plot commands omitted -


mResiduals = yn-yfit
mSSE= norm(mResiduals)^2

% or

output.residuals
gof.sse
```

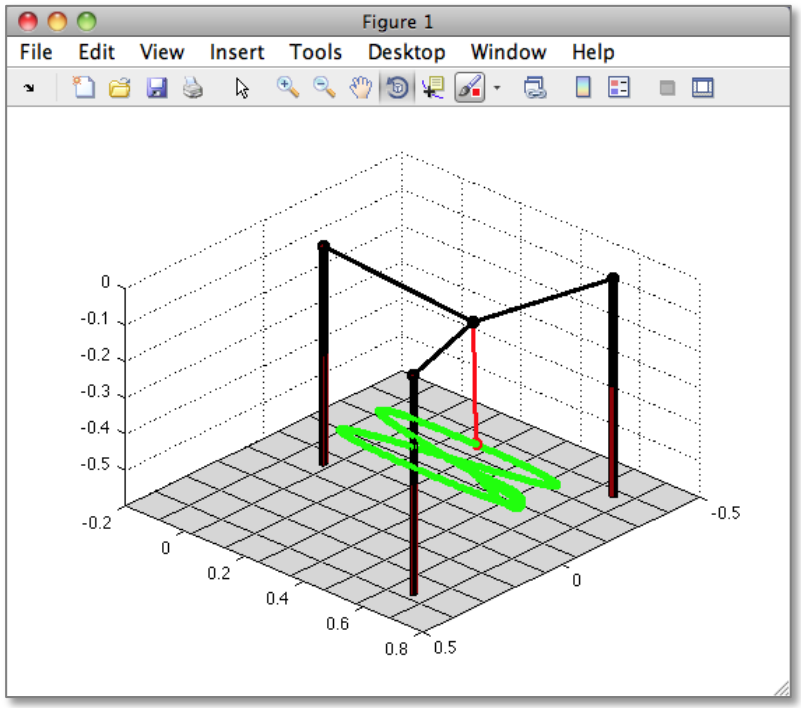


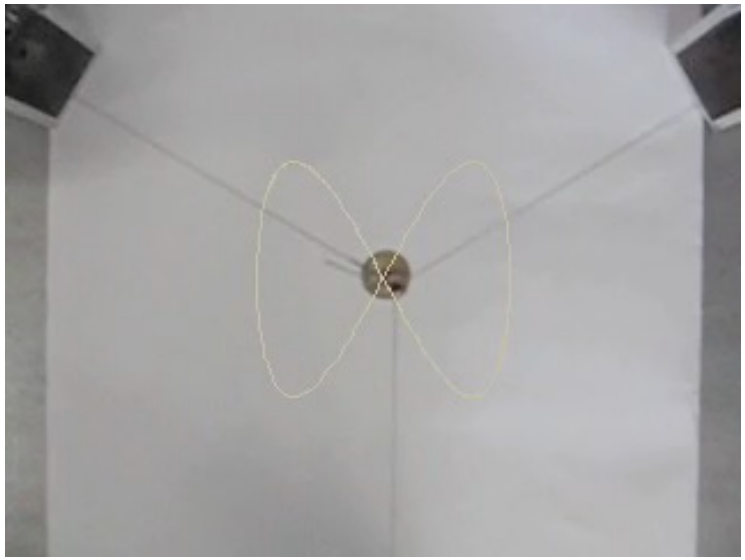
Matlab 3 - recap

- Logical arrays
- Cell arrays
- 3D Surface, meshgrid()
- 2D & 3D parametric plots/surface
- Interpolation
- Fit

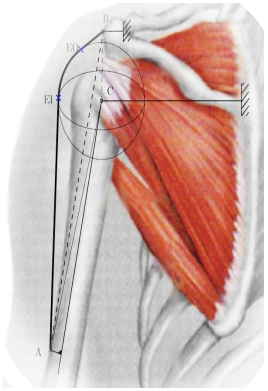
Exemple: spider-crane

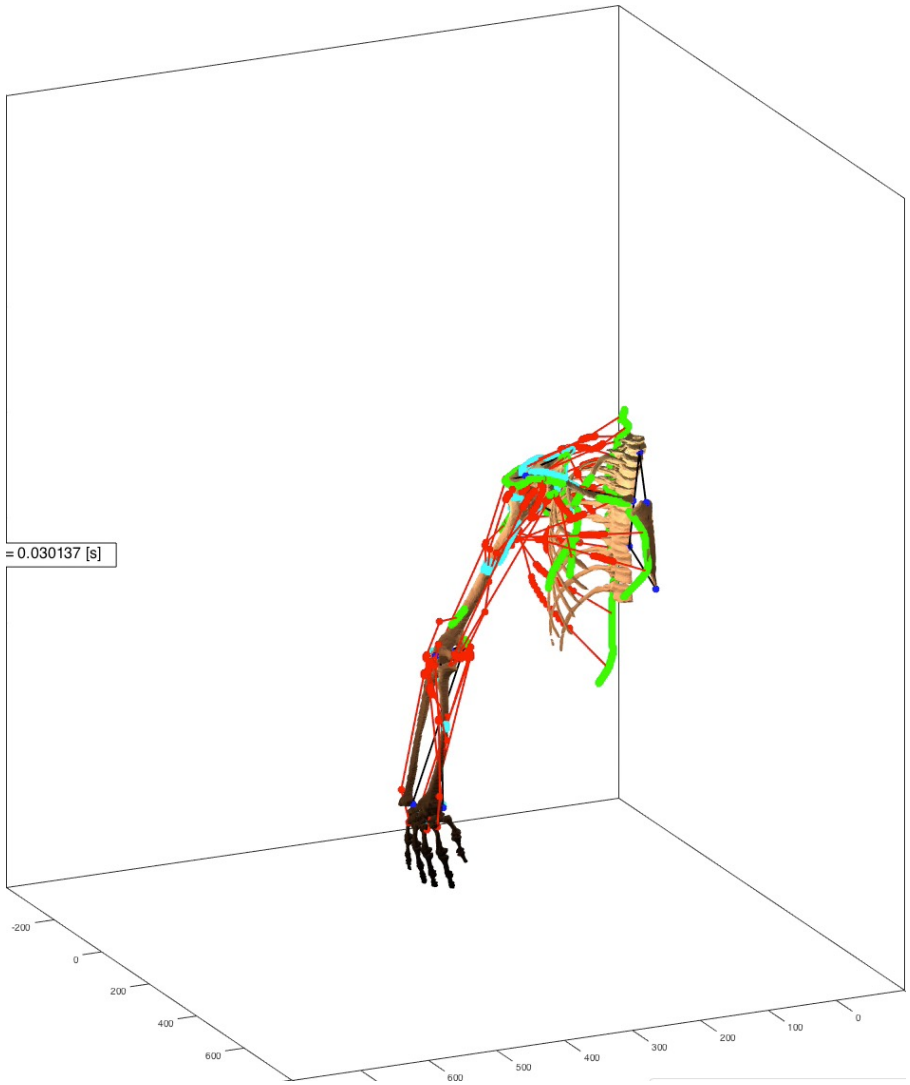
Grue sans partie mobile, permet des déplacements dynamiques rapides.





Exemple: projet épaule





Modélisation des mouvement d'une épaule (skelet + muscles) pour étudier les forces dans les articulations afin d'étudier les causes possible de l'artrose.