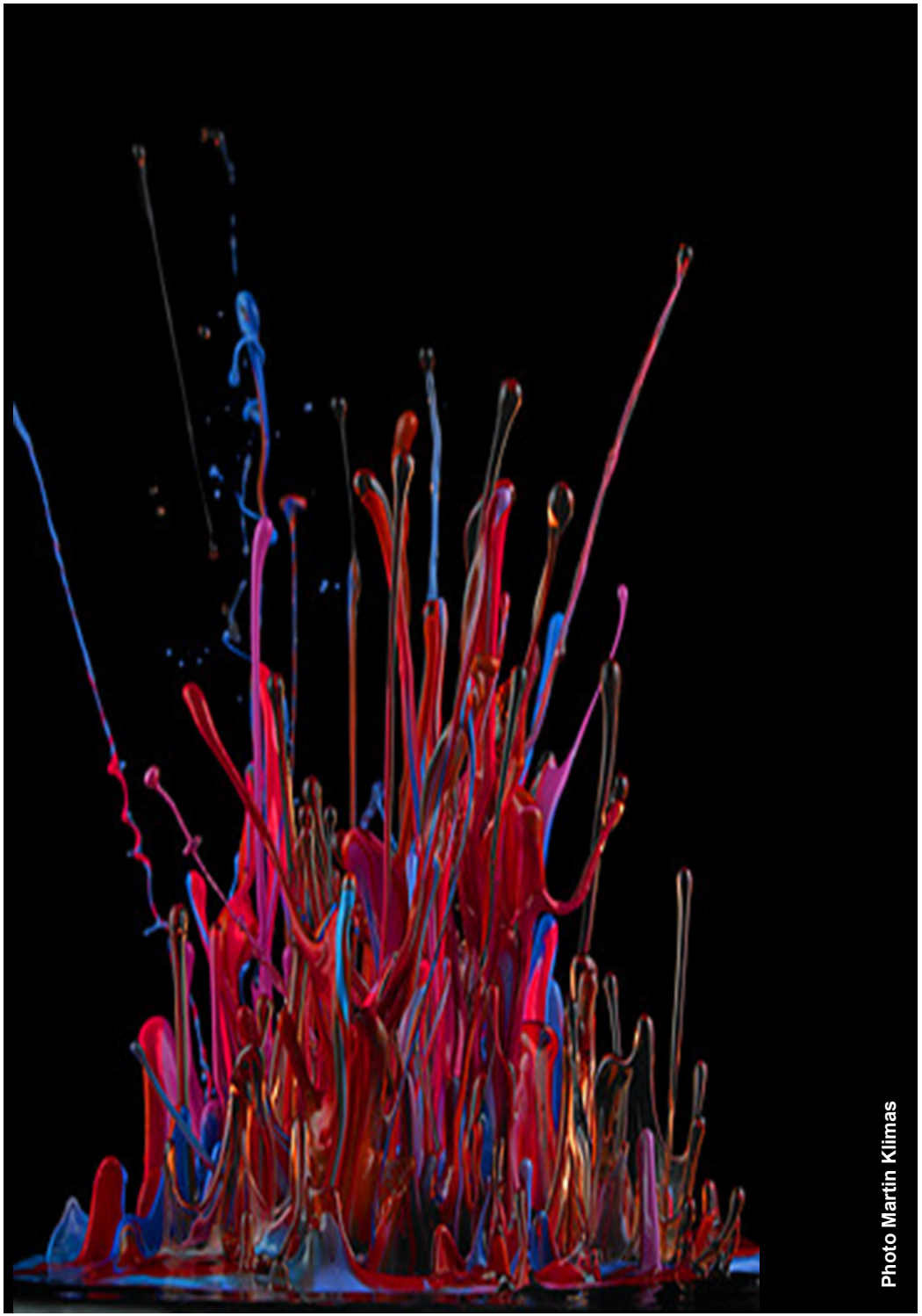


Photo Martin Klimas



# Matlab

*Interpolation & Fit*

*ME 2e semestre*

*rev. 13.7*

Christophe Salzmann

Laboratoire d'Automatique

# Module 10 - recap

- find()  & Logical array `x=R(R>0)`
- Programmation similaire à C/C++
- Plot/ezplot pour un graphique simple
- Dessin d'une surface calculé à partir d'une grille XY
- Surface paramétrique calculée à partir d'une grille formée des paramètres

# Script vs Function

*myScript.m*

```
% demo script
% var are Global!

G_in = 8;
G_c = G_in+1;
G_a = 3;
G_b = G_c + G_a;
```

Workspace

| Name | Value |
|---|---|
| G_a | 3 |
| G_b | 12 |
| G_c | 9 |
| G_in | 8 |

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

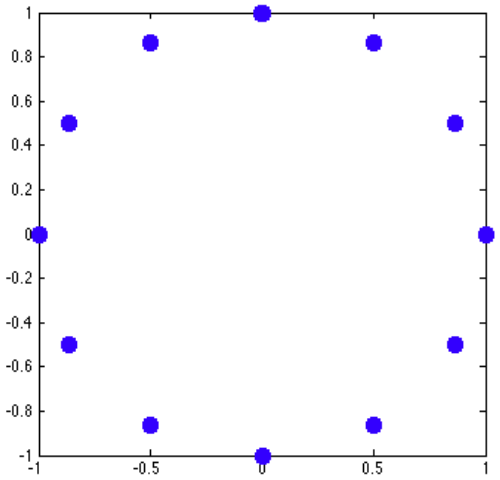
*myF.m*

```
% demo function
% var are local !

Function [a b] = myF(in)
          c = in+1;
          a = 3;
          b = c + a;
end
```

Function *SameAsFileName*
…
end

# courbe paramétrique 2D



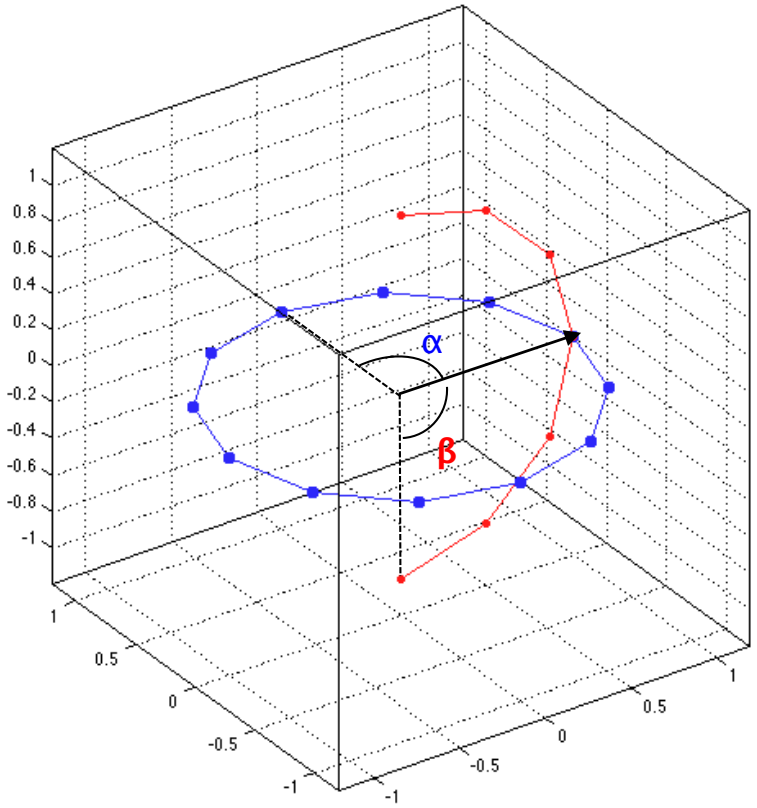

*Pourquoi 13 points?*

Paramètre: α

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

**vectorization**

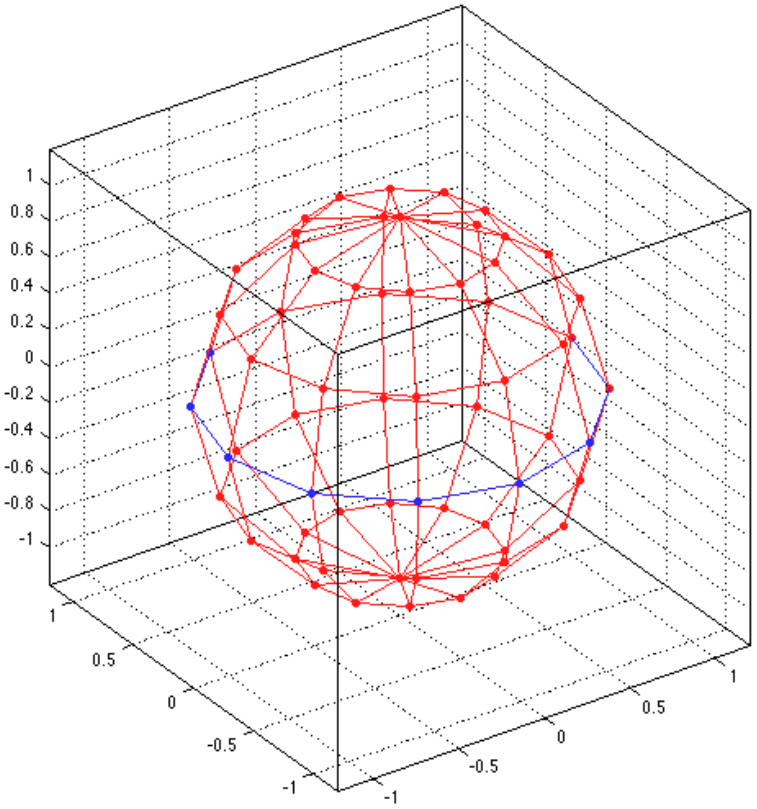
# courbe paramétrique 3D




Paramètres: **α**, **β**

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

# courbe paramétrique 3D




Paramètres: **α**, **β**

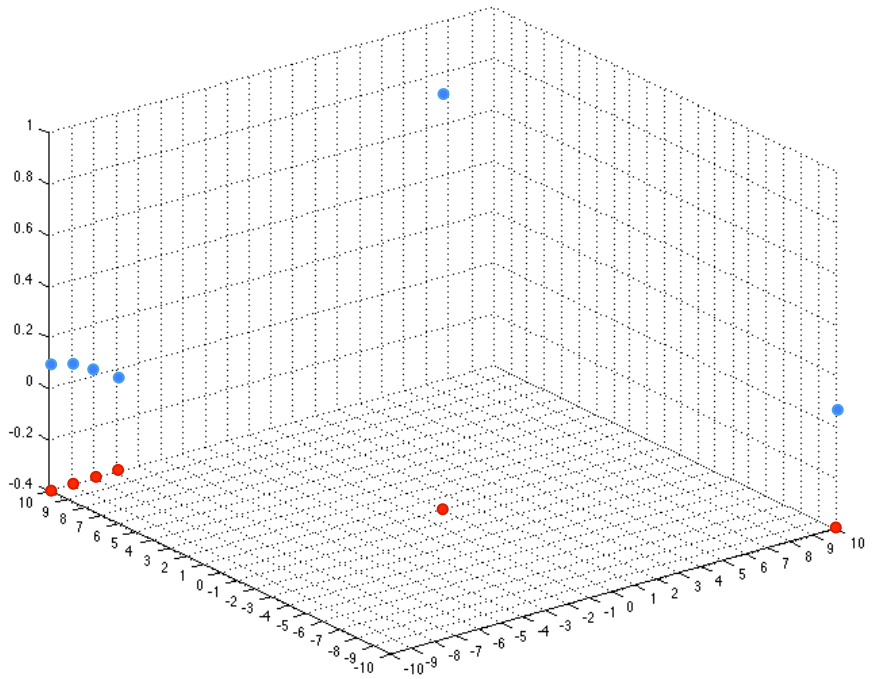
```
α = linspace(0,2*pi,13) % 1 tour
β = linspace(0,pi,7) % 1/2 tour
```

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

mesh(x,y,z)
```

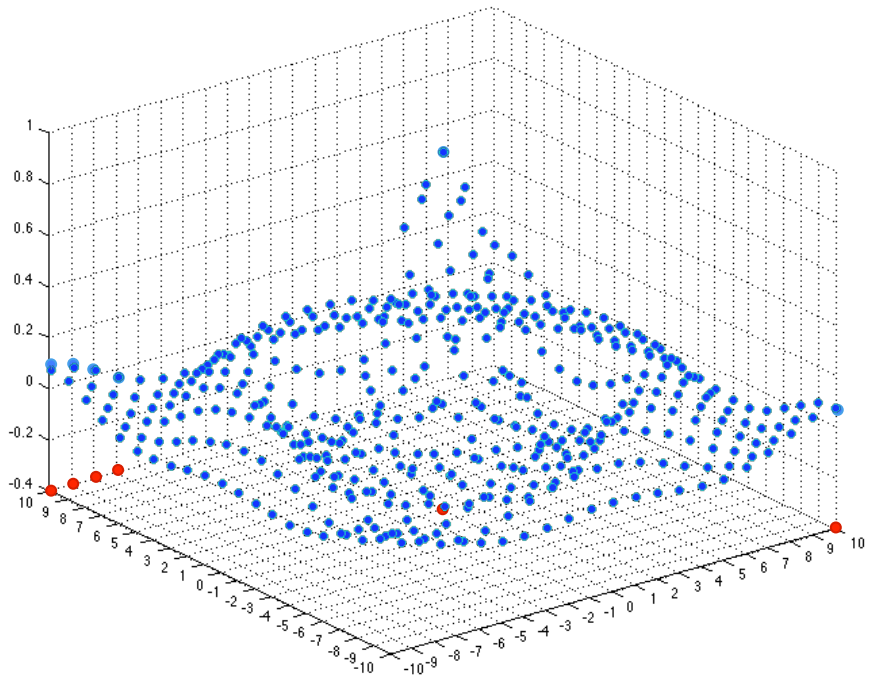
**Vectorization**

# Surface 3D




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

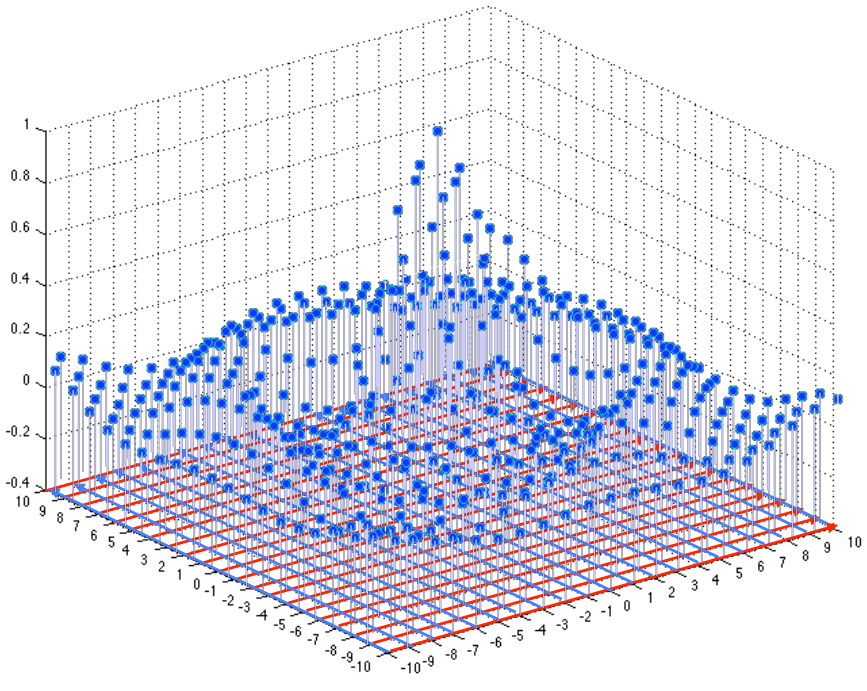
# Surface 3D



**2.45 [ms]**

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

# Surface 3D

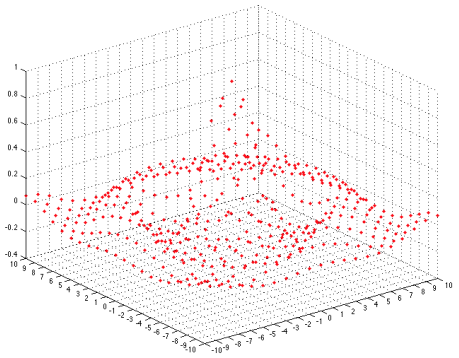


**0.043 [ms]**

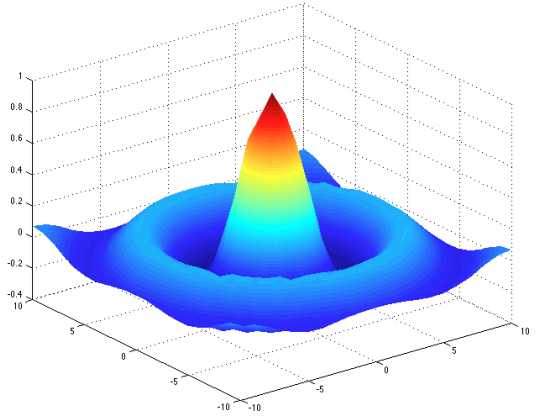
**Vectorization**

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

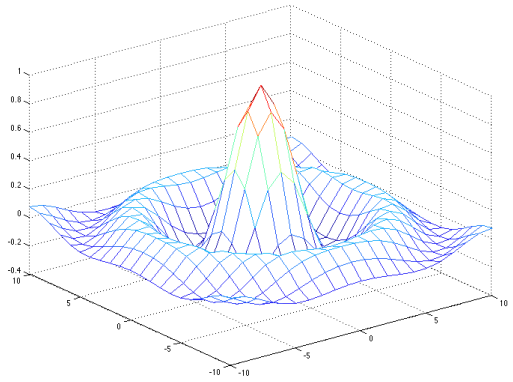
# Surface 3D



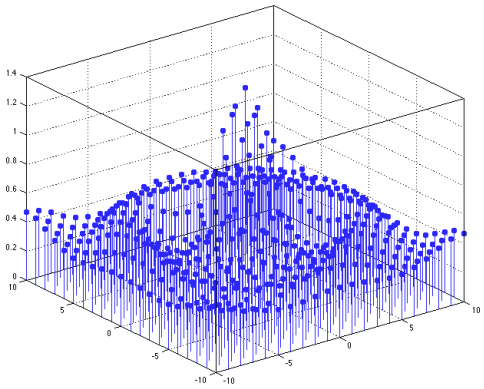
surf(X,Y,Z)



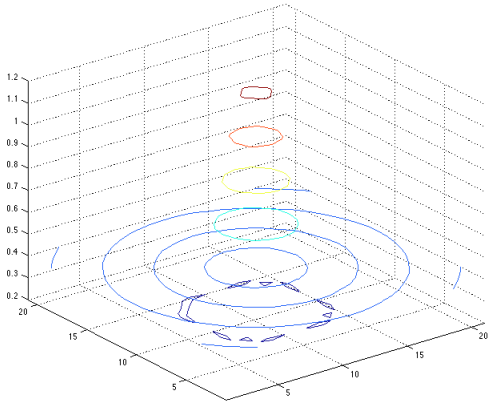
mesh(X,Y,Z)



stem3(X,Y,Z)



contour(X,Y,Z)

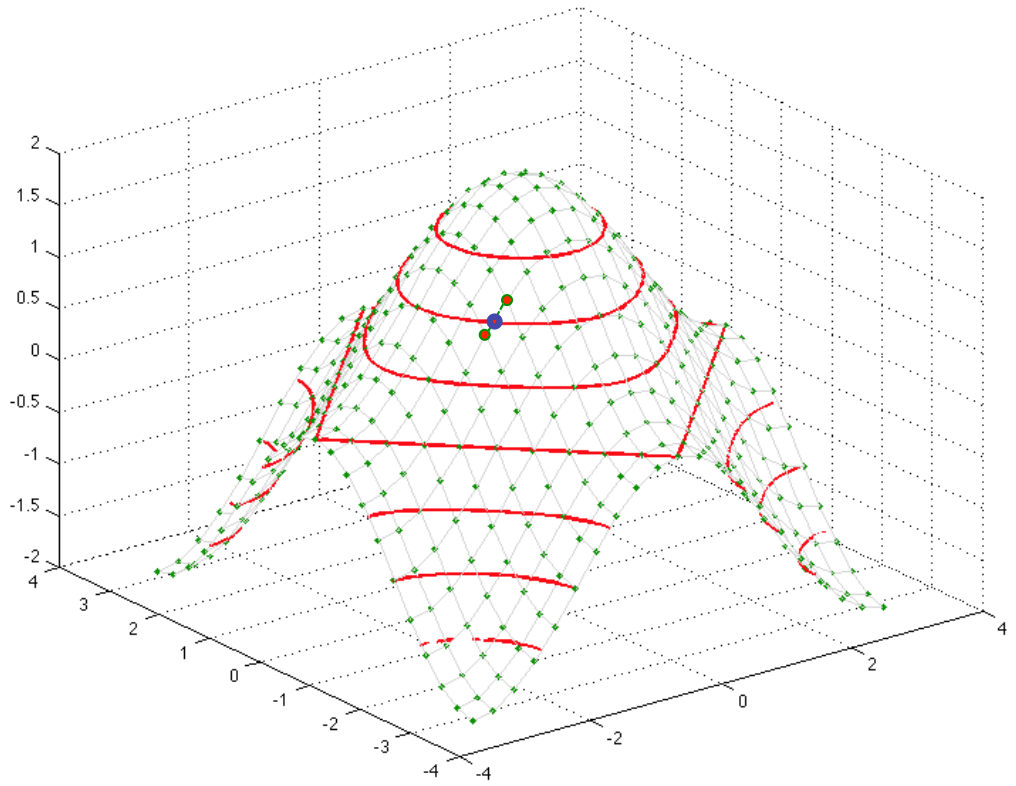


# Module 11

- Interpolation
- Fit
- Differentiation
- Intégration
- Simulation
- Représentation des nombres

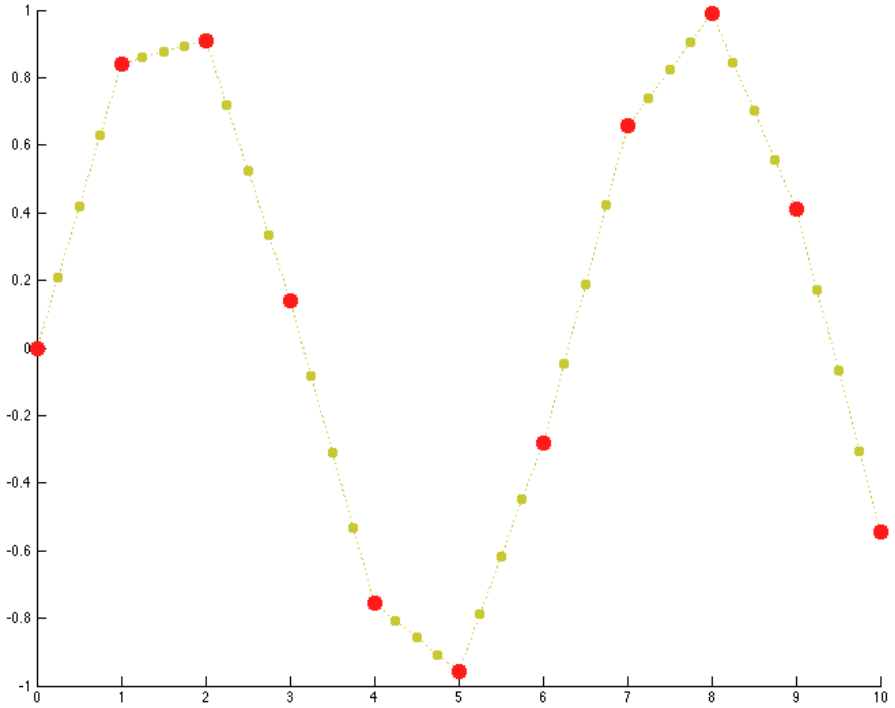
# Interpolation

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




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

# Interpolation



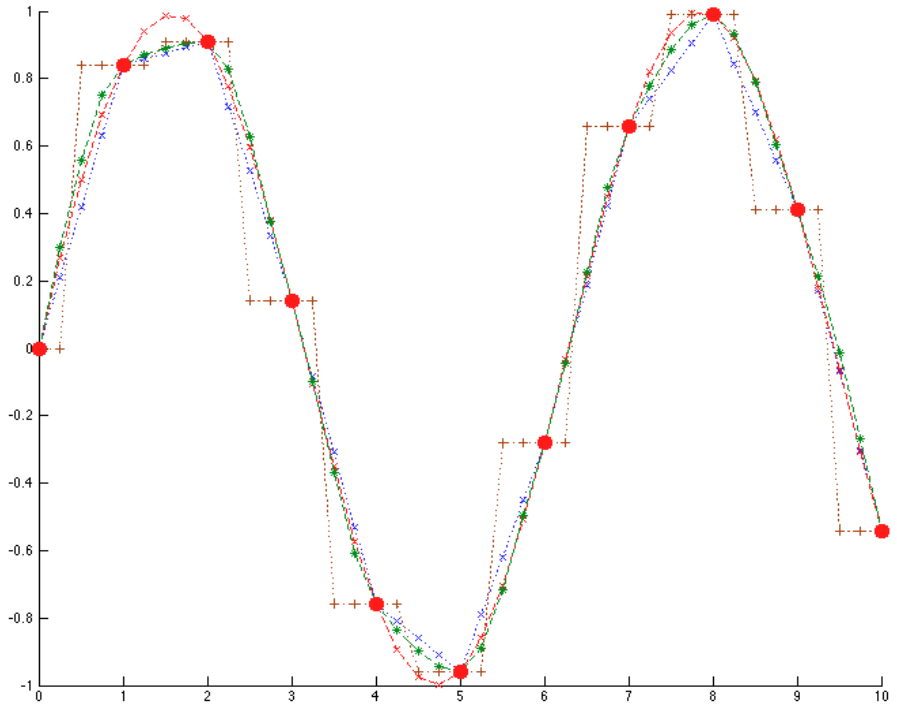

```
x = 0:10;          % 10 pts
y = sin(x);

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

yi = interp1(x,y,x2,'linear');

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

# Interpolation

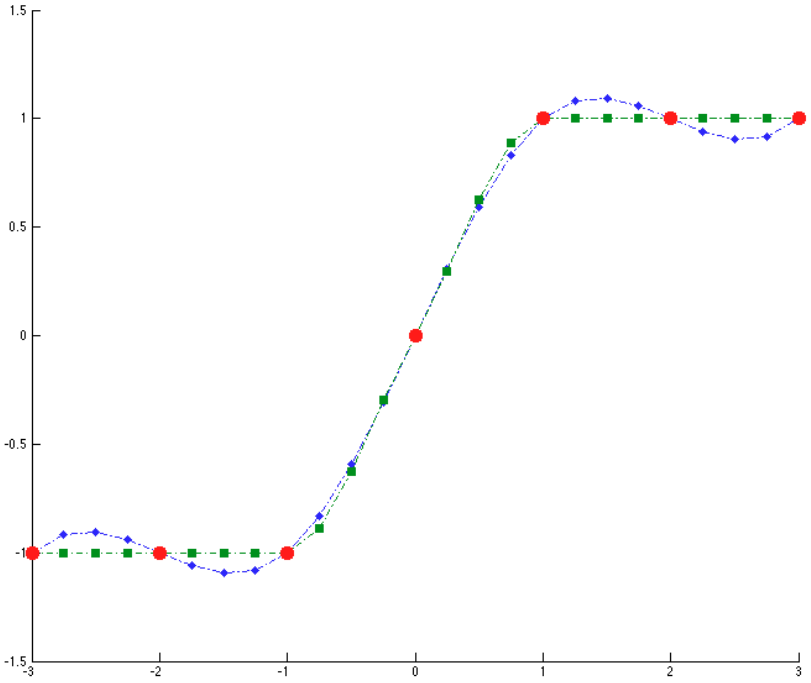



Linear
Spline
Cubic
Nearest

```
x = 0:10;          % 10 pts
y = sin(x);
xi = 0:.25:10;     % 40 pts
yi = interp1(x,y,x2,'cubic');
plot(x,y,'or', xi,yi,'-.xr');
yi = interp1(x,y,x2,'spline');
plot(x,y,'or', xi,yi,'-.*g');
yi = interp1(x,y,x2,'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*** n'a pas 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,'cubic');
plot(x,y,'or',  ...
     xi, ys,'-.xb',...
     xi, yc,'-.sg');
```

# 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.

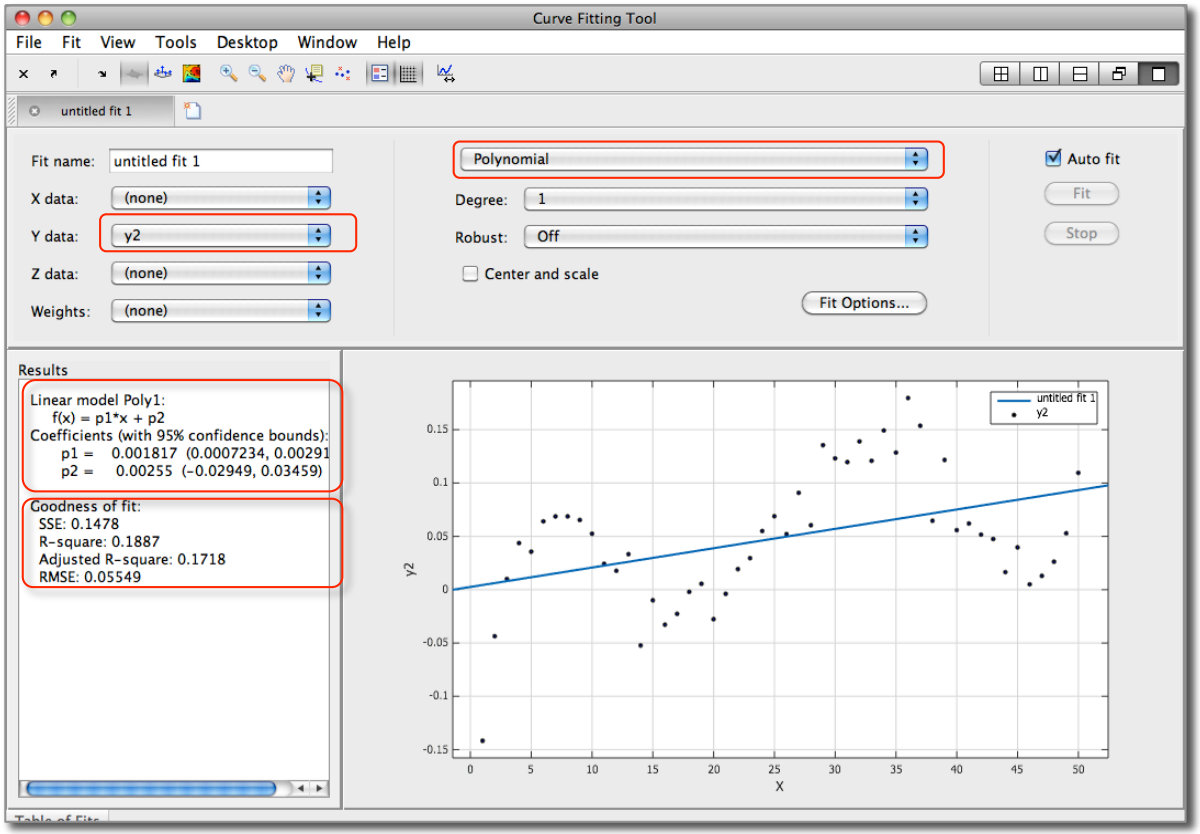
# Fit interactif

Choix du type de courbes
et paramètres relatifs

Choix des données

Equation et paramètres

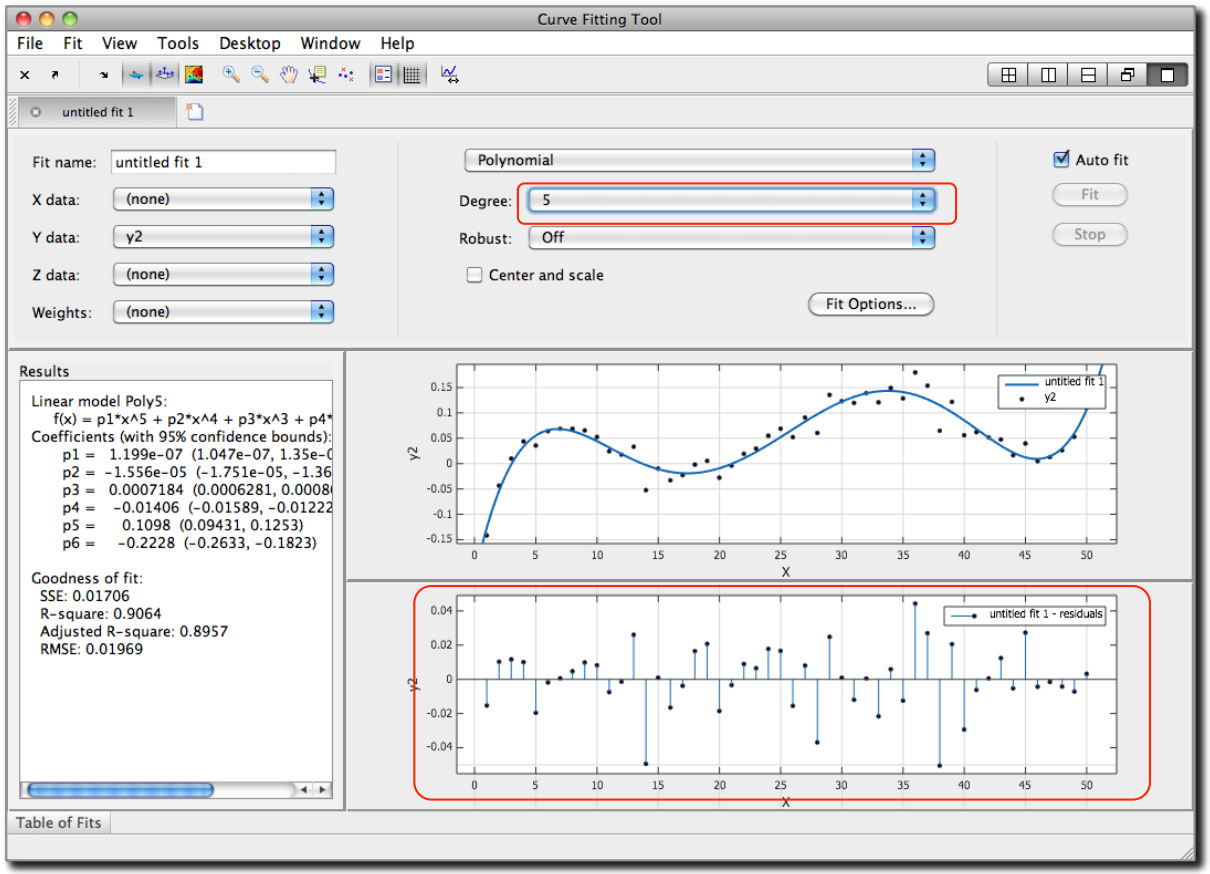
*Qualité* du fit




**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

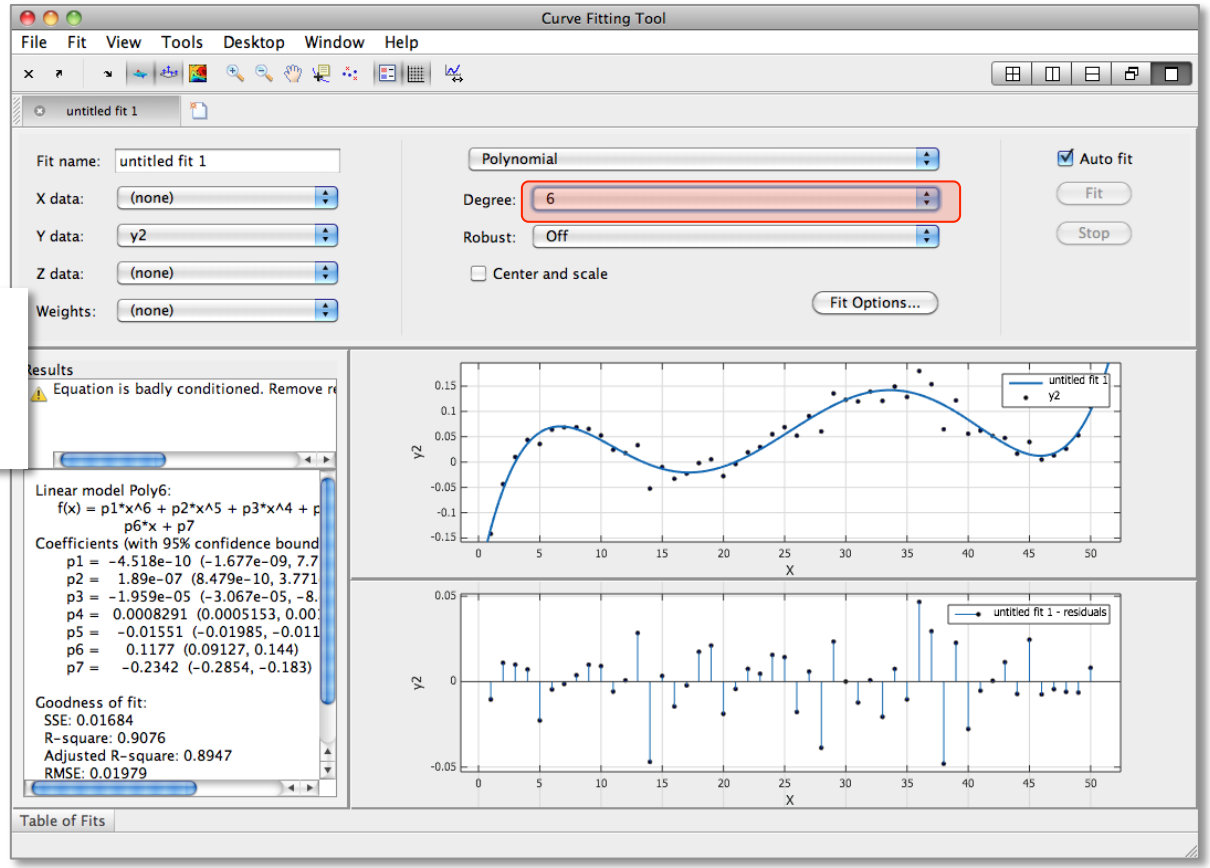



Résidu

# Fit interactif

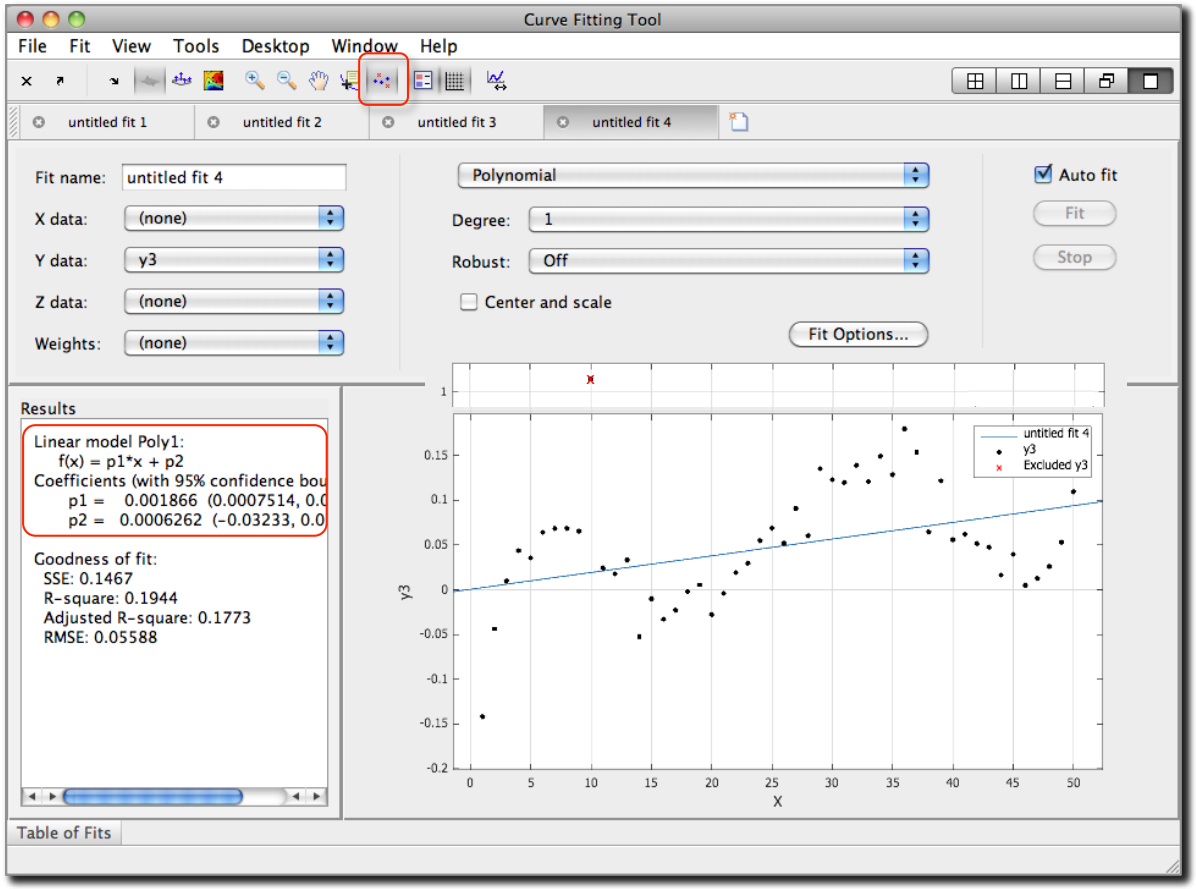
Degré du polynôme

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



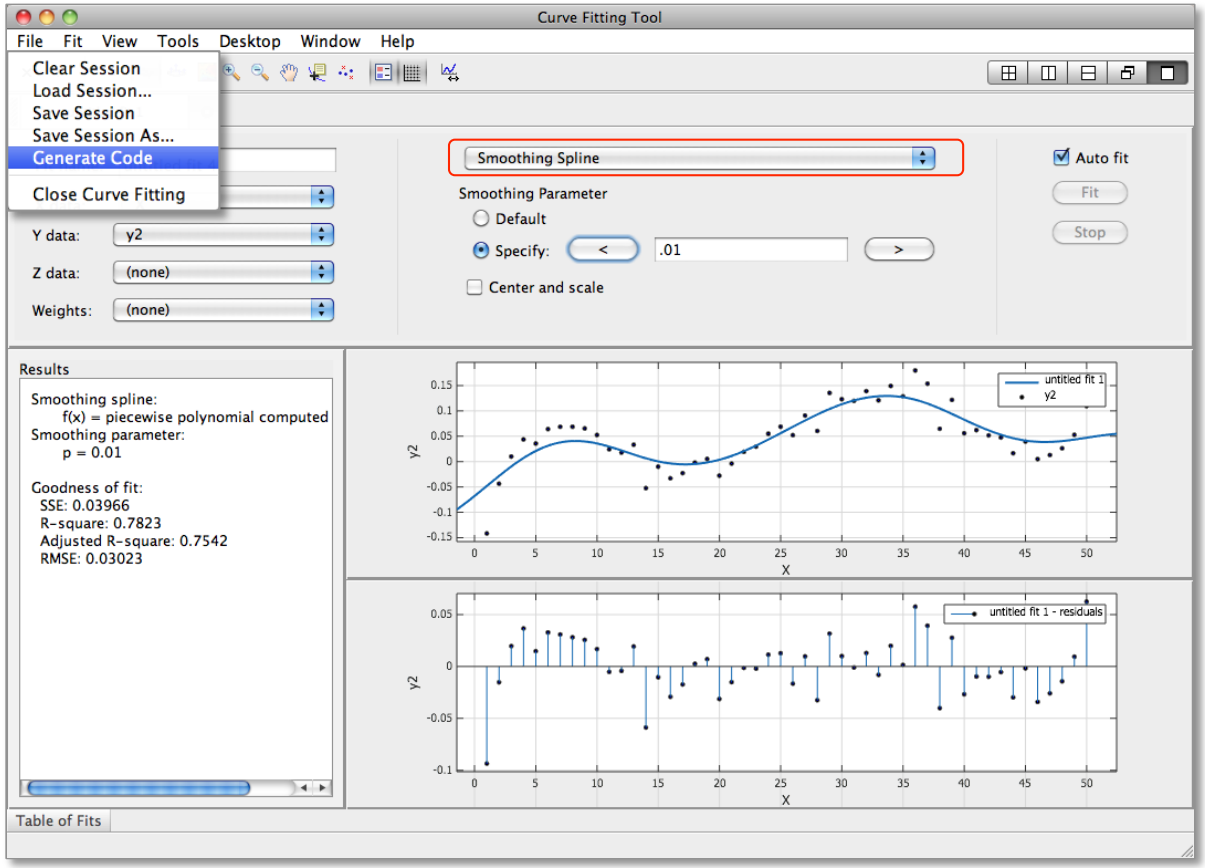
# Fit interactif

*Remove outliers*

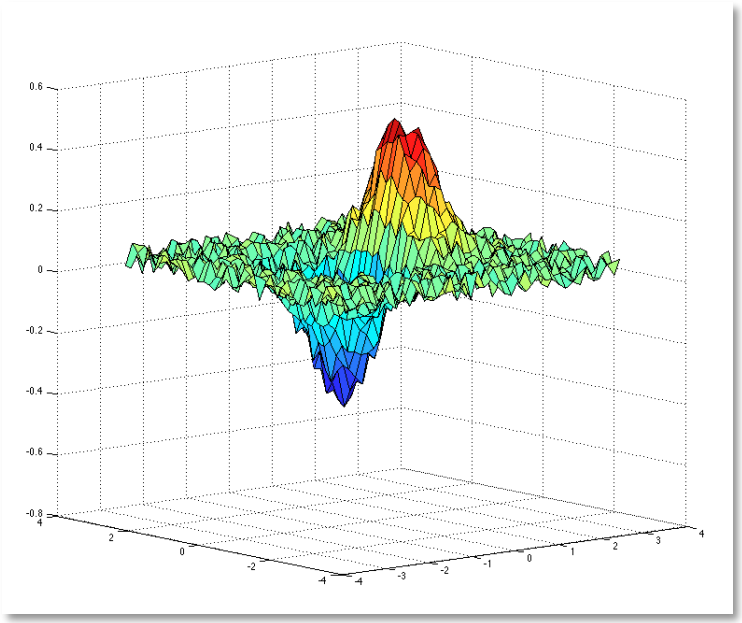


# Fit interactif

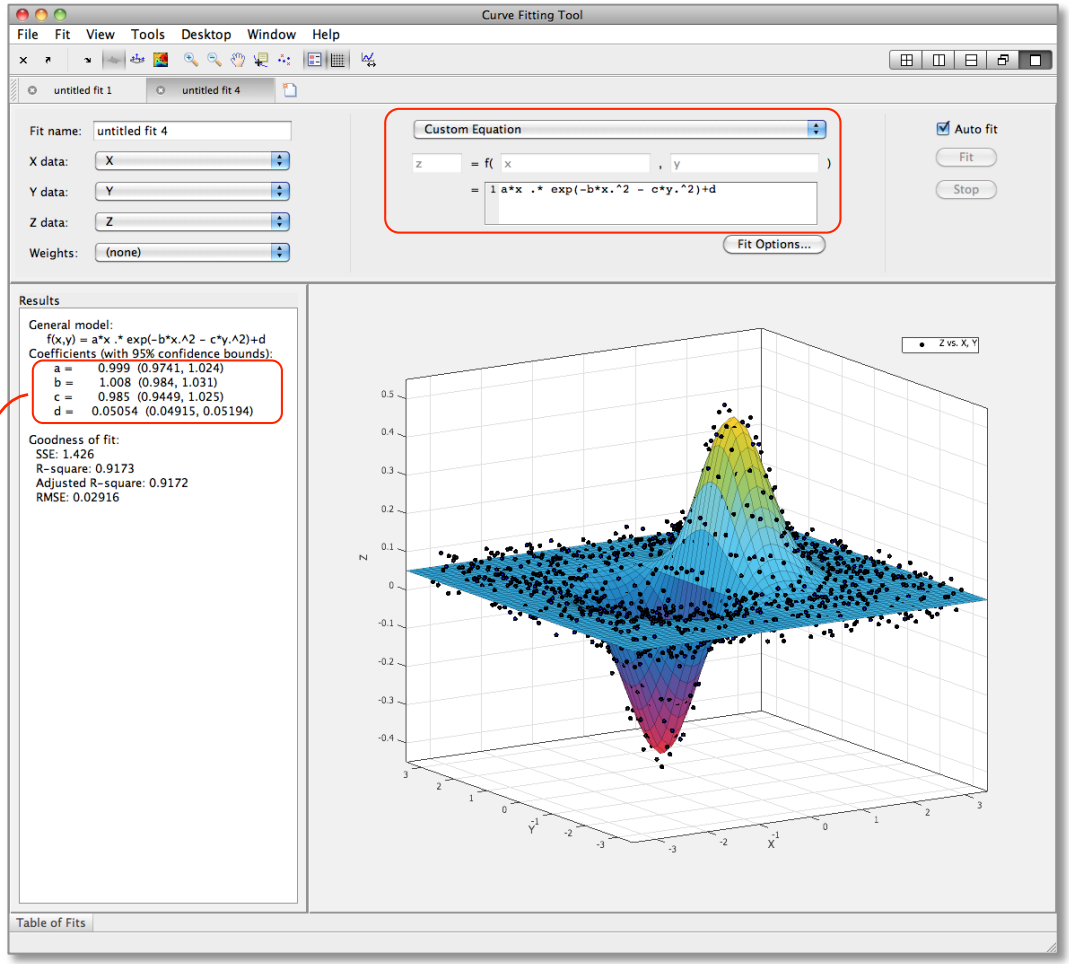
- Sauvegarde des *fits*
- Génération d'une fonction de fit avec les options choisies



# Fit interactif - 3D



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



# Polynômes

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

$$ax^4 + bx^3 + cx^2 + dx + e \quad \Rightarrow \quad p = [a\ b\ c\ d\ e]$$

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
```

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

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

Small but not '0', why?

eps = 2.2204 e$^{-16}$

# Polynomes et Fit

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

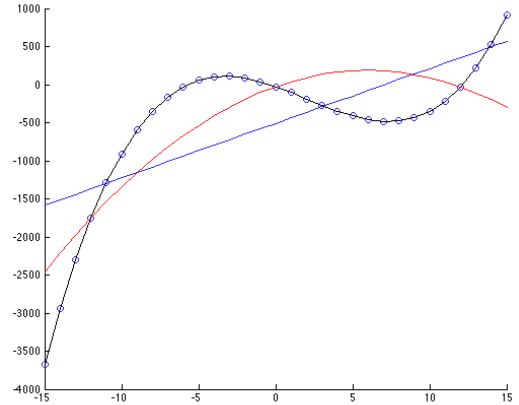
```
>> p = [1 -6 -72 -27];
>> x = -15:15 ;
>> y= polyval(p,x);
>> plot(x,y);

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

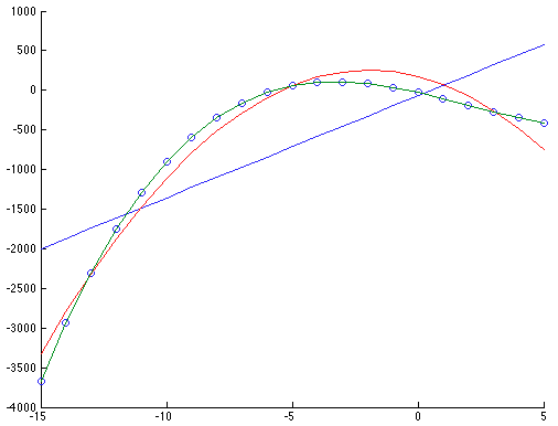
>> p2 = polyfit(x,y,2);
>> plot(x,polyval(p2,x),'r');

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

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




**-15 : 15**




**-15 : 5**

# Algèbre linéaire

```
>> A = [8  1  6;...
        3  5  7;...
        4  9  2];

>> rank(A)
ans =
     3

>> det(A)
ans =
     360  % det ≠ 0 => A can be inverted

>> inv(A)
ans =
    0.1472   -0.1444    0.0639
   -0.0611    0.0222    0.1056
   -0.0194    0.1889   -0.1028
```

# Algèbre linéaire

Soit le système d'équations suivant

$$\begin{aligned} 10a + 7b + 8c + 7d &= 32 \\ 7a + 5b + 6c + 5d &= 23 \\ 8a + 6b + 10c + 9d &= 33 \\ 7a + 5b + 9c + 10d &= 31 \end{aligned}$$

Il peut être décrit sous la forme $Ax = b$,

avec

```
>> A = [10  7   8  7;...
        7  5   6  5;...
        8  6  10  9;...
        7  5   9 10];

>> B = [32;23;33;31];

>> det(A)
ans =
     1  % det ≠0 => A can be inverted
```

Il peut être résolut:

```
>> x = inv(A)*b
x =
    1.0000      a
    1.0000      b
    1.0000      c
    1.0000      d
```

# Algèbre linéaire

Alternativement, il peut être résolut avec l'opérateur '\' *left matrix divide*

```
>> x = A\b
x =
    1.0000
    1.0000
    1.0000
    1.0000
```

Si la matrice A n'est pas carrée, '\' fourni une solution approximée par des moindres carrés, avec une erreur = A*x-b

```
>> A = [2 -2;...
       -1  1;...
        3  4];

>> rank(A)
ans =
        2

>> b = [4;3;2];
```

```
>> x = A\b;
x =
    0.8571
   -0.1429

>> err = A*x -b
err =
   -2.0000
   -4.0000
         0
```

# Différentiation

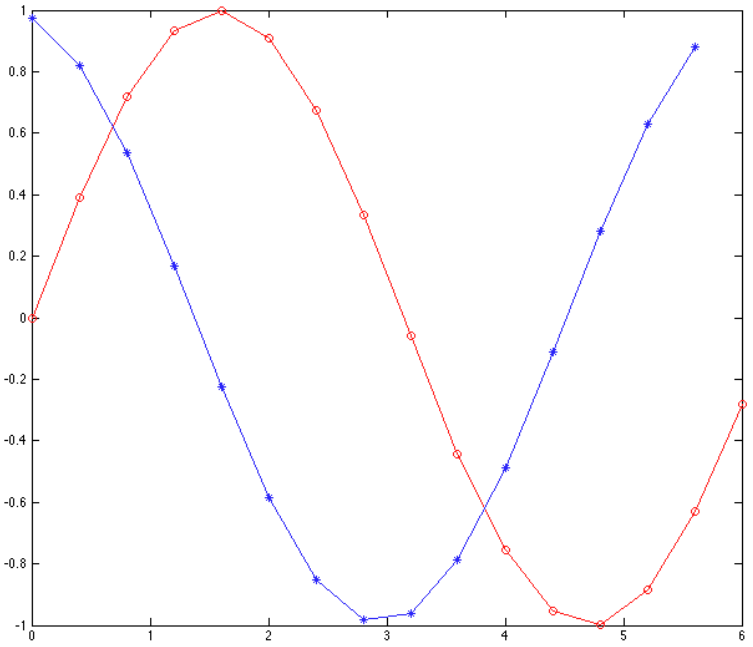
La fonction **diff()** calcul la différence entre 2 points adjacents d'un vecteur. Le vecteur retourné à un élément de moins.
Pour une matrice la fonction **diff()**est appliquée ligne par ligne.

```
diff(X)=[X(2)-X(1) X(3)-X(2) ... X(n)-X(n-1)]
```

```
pas = 0.4;
x   = 0:pas:2*pi;
Y   = sin(x);
dY  = diff(C);
dX  = diff(X);
plot(xx,C,'r-o',...
     xx(1:end-1),dY/dX,'-*');
```

Ex.

```
dY(1)=(Y(2)-Y(1))/(X(2) - X(1))
     = (0.4-0.0)/0.4   = 1
```



# Différentiation symbolique

La fonction `diff()` permet également de calculer une dérivée de manière symbolique.

```
syms x
diff(sin(x))

 cos(x)

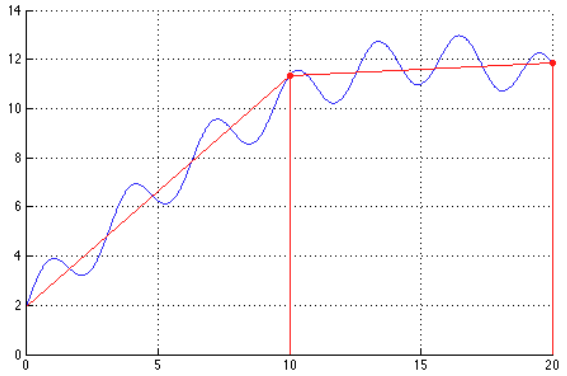

diff('cos(y)',sym('y'))

 -sin(x)
```

# Intégration - méthode des trapèzes

Intégration numérique par la méthode des trapères avec la fonction `trapz(X,Y)`.

Note: matlab implémente d'autres méthodes d'intégration, par ex. Simson à l'aide de `quad()`.

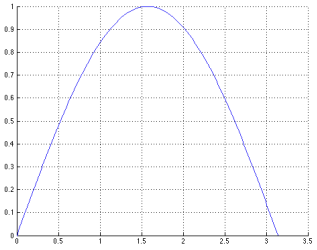


*Ex. Sin(x) avec la méthode des trapèzes*

$$\int_0^{\pi} \sin(x)\,dx = 2$$

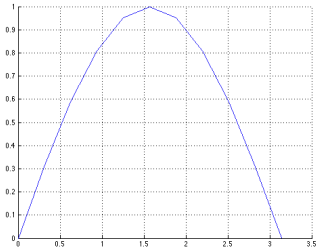
```
X = 0:pi/n:pi;
Y = sin(X);
Z = trapz(X,Y)
```

`n = 100 pts`



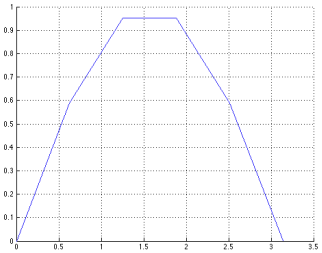
`z = 1.9998`

`n = 10 pts`



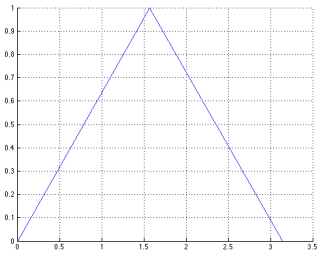
`z = 1.9835`

`n = 6 pts`



`z = 1.9338`

`n = 3 pts`



`z = 1.5708`

# Intégration symbolique

Intégration symbolique avec la fonction `int(X,Y)`.

```
syms x
z = int(sin(x))
```

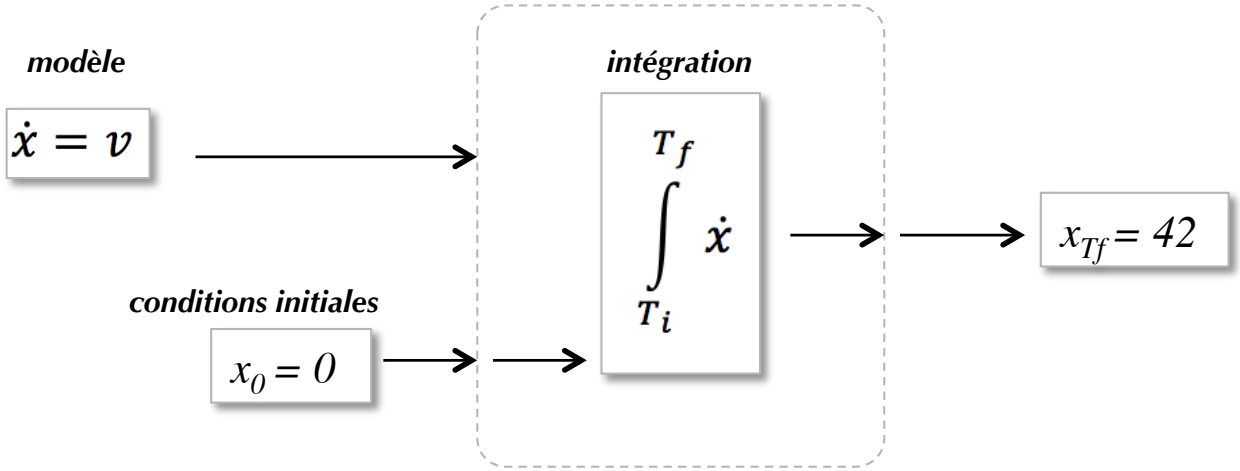
```
 z = -cos(x)
```

```
z = int(sin(x),x,0,pi)
```

```
 z = 2
```

# Intégration ODE - simulation

- Un système physique (ou économique, etc.) peut être décrit à l'aide d'équations mathématiques définissant un modèle plus ou moins précis de la réalité.
- Le modèle décrivant le système physique est employé par un logiciel de simulation (ex. *toolbox* Simulink) pour représenter l'évolution du modèle au cours du temps.
- L'évolution temporelle du modèle est calculée de manière itérative en intégrant numériquement les équations sur un petit pas de temps.
- Plusieurs méthodes sont à disposition du moteur d'intégration ou *Solver* pour intégrer les équations. La plus simple est la méthode des trapèzes et la plus courante est Runge-Kutta.
- La manière de définir mathématique un modèle ainsi que les problèmes liés à l'intégration numérique (condition initiales, convergence, etc) seront abordés dans d'autres cours.



# Simulation - balle

Exemple:

*Balle lâchée d'une hauteur de 10m avec une vitesse initiale nulle*

- La vitesse de la balle va accélérer de manière constante, défini par *g*
- Le modèle a 2 états, la position et la vitesse de la balle

$$\frac{dx}{dt} = v \qquad \frac{dv}{dt} = -g$$

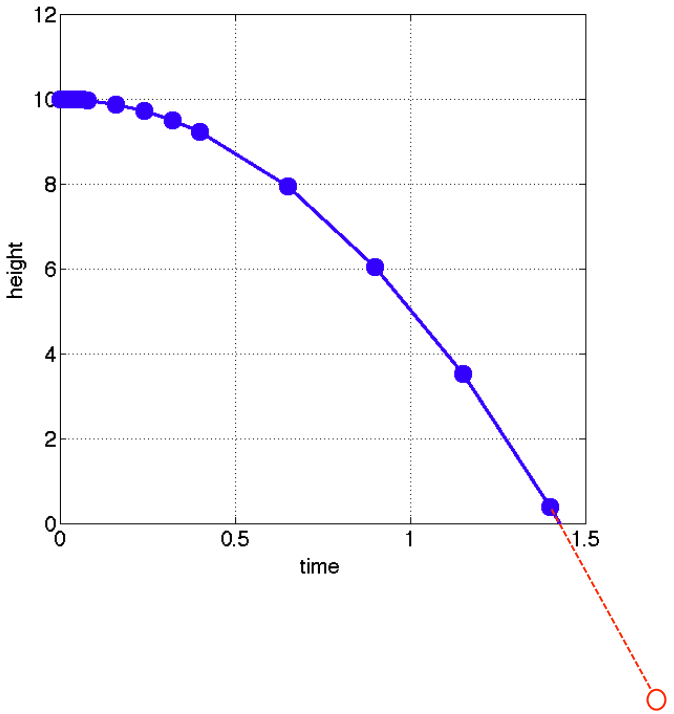
- Le *solver* ODE intègre les équations dynamiques du modèle à partir des conditions initiales, dans un lapse de temps donné en utilisant la méthode Runge-Kutha.

# Simulation - balle

- Le *solver* ODE intègre la fonction `ODEball` à partir des conditions initiales `y0`, dans un lapse de temps `[0 2]` sec en utilisant la méthode Runge-Kutha `ode45`.
- Le modèle à intégrer a 2 états: la position *x* `y(1)` et la vitesse *v* `y(2)` de la balle

```
% modele a integrer
function dydt = ODEball(t,y)
   dydt = [y(2); -9.8];
end

% cond. init. h=10, v=0
y0 = [10 0];
ts = 0;  tf = 2;
[t y]= ode45(@ODEball,[ts tf],y0);
plot(t,y (:,1),'-bo');
```



# Simulation – balle & rebonds

- La collision avec le sol (x=0) est détectée avec un événement **`events`** qui stoppe le *solver*.
- La simulation est ensuite relancée avec de nouvelles conditions initiales: nouvelle vitesse = - 0.85 * ancienne vitesse; hauteur = 0

```
function [val,stop, dir] = events(t,y)
    val  = y(1);    % position
    stop = 1;       % stop simulation
    dir  = -1;
 end


 function dydt = ODEball(t,y)
    dydt = [y(2); -9.8];
 end

opt = odeset('Events',@events);


ts = 0; tf = 30; % cond. init.
y0 = [10; 0];

tout = ts;  yout = y0.';
```

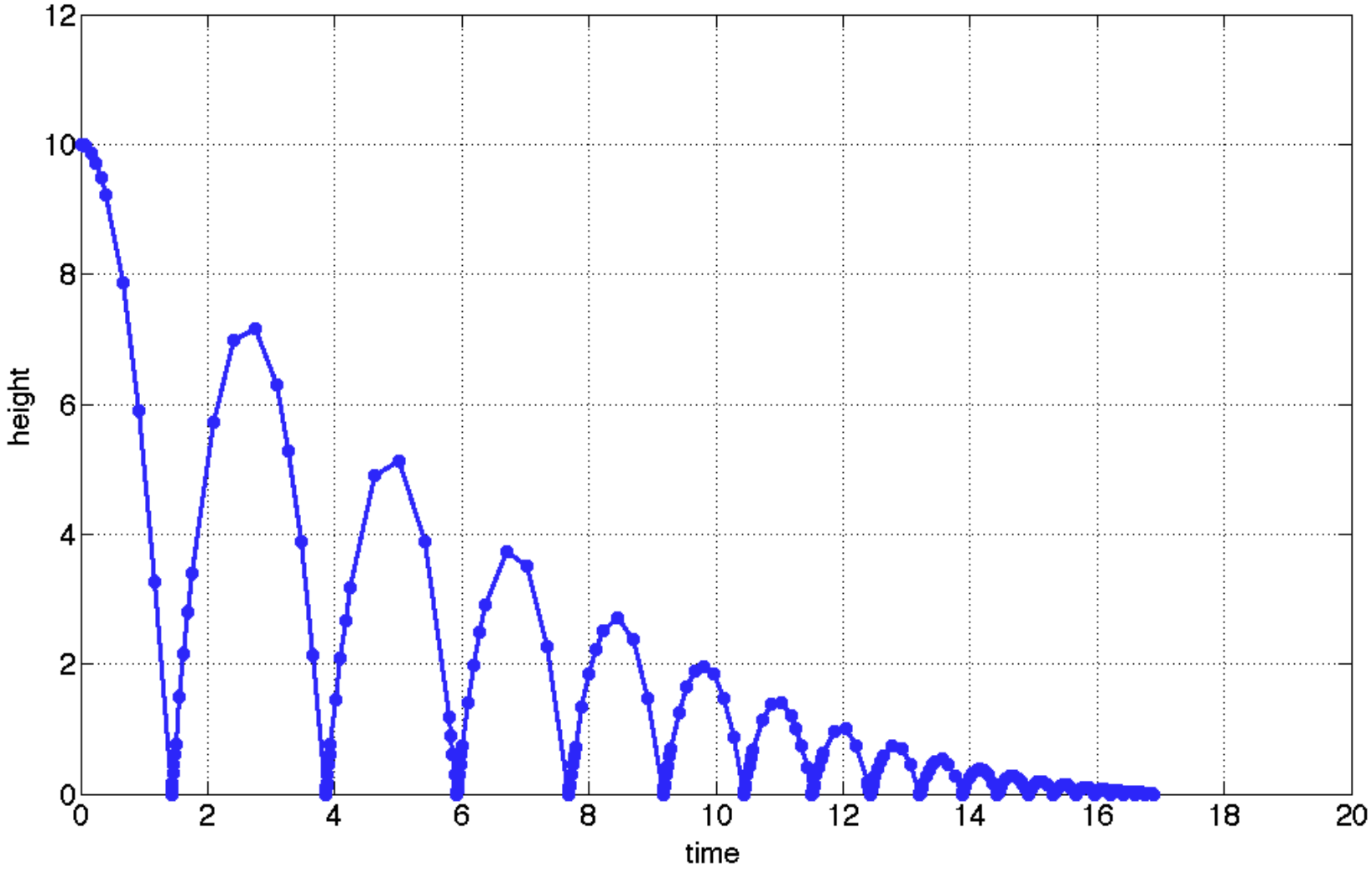
```
% 20 rebonds
for i = 1:20
          % simule 1 rebond
    [t,y] = ode45(@f,[ts tf],y0,opt);

          % accumule les pts
     nt = length(t);
     tout = [tout; t(2:nt)];
     yout = [yout; y(2:nt,:)];

          % nouvelles cond. init.
     y0 = [0, -0.85*y(nt,2)];
     ts = t(nt);
end

 plot(t,y (:,1),'-bo');
```

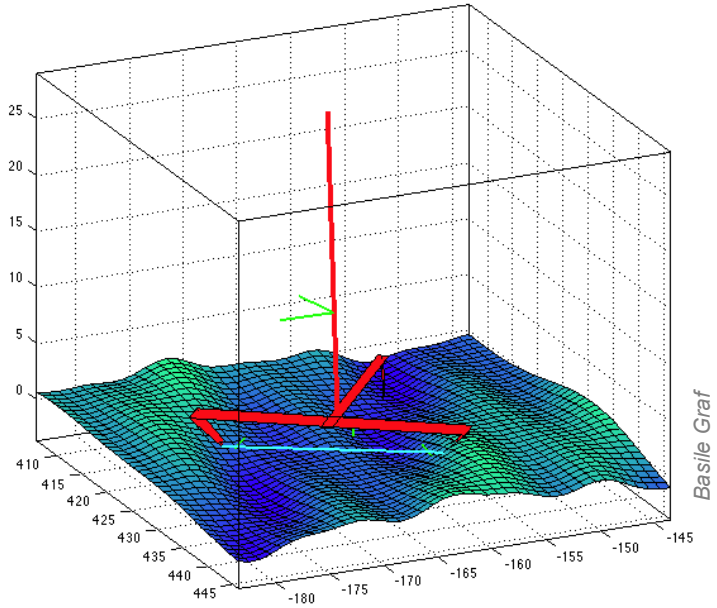
# Simulation – balle & rebonds



# Simulation – Hydroptère

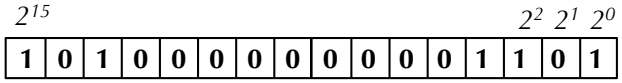



http://hydroptere.epfl.ch/



# Représentation des nombres

- Un ordinateur/CPU comprend et travaille avec des 1 et 0 (choix)
- Pour pouvoir êtres employés par le CPU, les nombres sont donc convertis de base 10 en base 2
- Le CPU gère des nombres binaires de taille donnée, ex: 32bit, 64bit
- Les entiers (base 10) représentés en base 2 ont donc une taille fixée par le nombre de bits à disposition, le nombre de valeurs possibles est également fixé par le nbr. de bits.




$1 \times 2^{15} + 0 \times 2^{14} + 1 \times 2^{13} + 2^3 + 2^2 + 2^0 = 32768 + 40960 + 8 + 4 + 1 = 73741$

- Les compilateurs définissent des constantes pour ces nombres maximums, ils peuvent également êtres calculé si l'on connaît la taille des nombre.

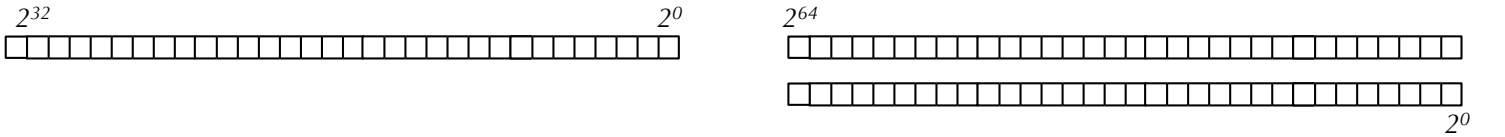
Ex: entier **non signé**, représenté sur 16 bit => 1111111111111111 => 0 .. 65535 = $2^{16}$-1
pour les entier **signés** sur 16 bit => 1111111111111111 => -32768 .. 32767

C -> `UINT_MAX` (`#include <limits.h>`)
C++ -> `numeric_limits<int>::max()` (`#include <climits>`)

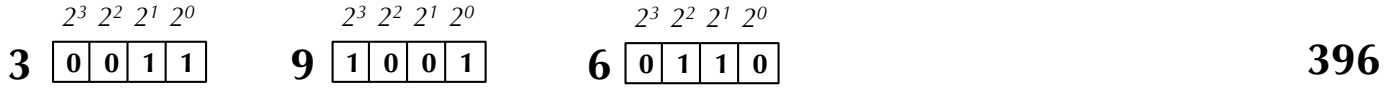
*Comment faire pour représenter les nombres entre 0 et 1 ou des nombres plus grand qu'`INT_MAX` ?*

# Représentation des nombres

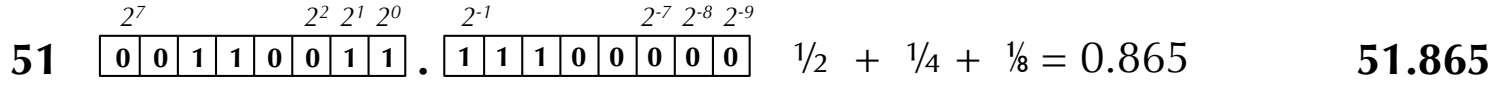
- Augmenter le nombre de bits employés pour représenter un nombre (32 -> 64 )




- Encodage ***BCD***, chaque chiffre (0 à 9) est codé sur 4 bit (0000 à 1001), nécessite des traitements spécifiques




- Format ***fixed point*** définissant une partie entière et une partie décimale encodé sur un nombre fixe de bit, -> rapidité de calcul similaire aux entiers combinée avec la possibilité de représenter des nombres entre 0 et 1.

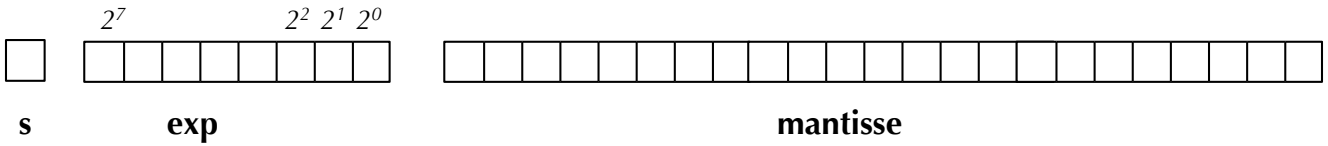


# Représentation des nombres

- Format ***floating point*** définissant un nombre réel à l'aide d'un signe, d'une mantisse et d'un exposant. Ce format permet de placer la virgule n'importe ou dans le nombre, d'ou son nom. Le format *floating point* a été normalisé (IEEE 754)

Ex. réel simple précision *(single*-32 bits):

1 bit de signe (**s**), 8 bits d'exposant (**exp**), 23 bits de **mantisse**, avec **'1.'** implicite -> 1.xxxx)




$valeur = (-1)^s, 1.\text{mantisse} \times 2^{(\text{exp} - \text{décalage})} = 1.\text{mantisse x } 2^{(\text{exp } -127)}$

*déclage*: $2^{\text{nbr de bit de l'exposant}}$, $2^{8-1}-1 = 2^{8-1}-1 = 127$ => décalage de -126 à 127

```
0  00000000  00000000000000000000001 =>    2^0-127 x 2^-23 x = 1/2^127 x = 1.4e^-45
0  11111111  00000000000000000000000 =>   NaN
```

# Représentation des nombres

- De part ça construction, un nombre réel ne peut pas forcément être représenté dans le format *floating point,* il est alors **approximé** en utilisant à la place l'**arrondi** le plus proche.
- L'opération d'arrondir un nombre est donc **forcément entachée d'erreur!**
- Il y a un également le risque d'un **effet cumulatif** des erreurs à prendre en compte. Il est dépendant de l'algorithme utilisé (itérations)

Ex.

```
double x= 0.1;   //  ->0.10000000 1490116119
```

- Par construction, les nombres en virgule flottante ne sont **pas espacés de façon identique** dans l'ensemble des réels : plus l'on est proche du plus petit nombre représentable, plus ils sont denses.

# Représentation des nombres

- En raison du risque d'erreur d'arrondi, **ne jamais faire de test d'égalité stricte** '==' ou '!=' sur des nombres représentés en virgule flottante.

ex.

```
single x;
for(x=0.0; x != 10; x += 0.1)   // boucle infinie
        cout<< x <<endl;
```

- La **précision** des nombres représentés en virgule flottante **étant finie**, l'addition d'un nombre très petit à un nombre très grand (dans les bornes des nombres valides) peut n'avoir aucun effet et retourner le nombre très grand.
- De même l'ordre des opérations peut jouer un rôle.

ex.

```
single a=10000; b=0.000001
a+b = 10000 = a !!!
même si b >> epsilon!
```

# Module 10 - recap

- Interpolation
  - Pour trouver des valeurs intermédiaires
- Fit
  - Pour faire passer une courbe au plus près de points donnés
- Intégration et différentiation
  - Trapz et Diff
- Simulation
  - Intégration d'ODEs
- Représentation des nombres
  - Un nombre réel ne peut pas forcément être représenté dans le format floating point, il est alors **approximé** ce qui introduit une **erreur**!