Algorithmes, Programmation, IA

Cours 7

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/algo-prog-ia-25

Plan

- représentation des nombres réels
- norme flottant IEEE 754
- recherche de racines et d'extrema
- méthode par descente du gradient
- régression linéaire

dès maintenant: télécharger Python 3 en `http://www.python.org`

Quelques rappels

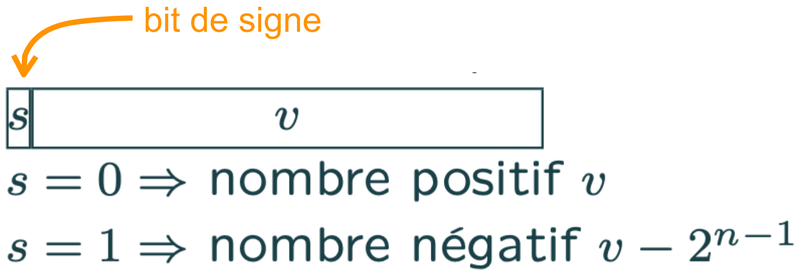
- le cours utilise le langage Python et l'environnement Visual Studio Code. (`vscode`)
- et pour la partie IA, la bibliothèque Pytorch

Représentation des nombres entiers

• entiers: en complément à 2 sur *n* bits (*n* = 8, 16, 32, 64)

$$-2^{n-1} \leq x \leq 2^{n-1} - 1$$

10000000 01111111 en binaire



n = 8 $-128 \leq x \leq 127$

n = 16 $-32768 \leq x \leq 32767$

n = 32 $-2147483648 \leq x \leq 2147483647$

n = 64 $-9223372036854775808 \leq x \leq 9223372036854775807$ [9 milliards de milliards]

• entiers: grand nombres en précision arbitraire (GMP, Maple, Ocaml, Python3)
[ne dépend que de la taille de la mémoire]

Représentation des nombres réels

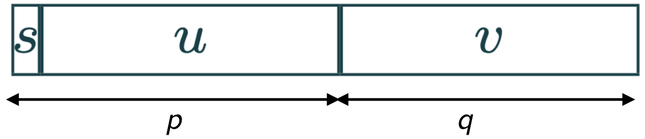
• en virgule fixe : par exemple p bits avant la virgule et q bits après

[utilisé en graphique interactif, par exemple Macintosh 1ères versions 1984]

$$-2^{p-1} \leq x \leq 2^{p-1} - 1/2^q$$

$$-32768 \leq x \leq 32767.9999847412109375$$

$$2^{15} = 32768 \quad (p = q = 16)$$



[en fait, ce sont les entiers sur 32 bits divisés par 2^{16}]

• réels exacts : impossible puisque l'égalité est indécidable

[comment décider si 1 == 0.99999999999999999…. ?]

• réels exacts par intervalles : compliqué et peu efficace

Représentation des nombres réels

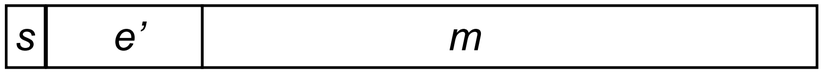
- les réels flottants sont des approximations des nombres réels

```
>>> 0.1 + 0.2 == 0.3
False
```

```
>>> 0.1 + 0.2
0.30000000000000004
```

- en virgule flottante : *IEEE 754 standard for floating points* [W. Kahan 1985]

  bit de signe *s*, exposant *e'* sur 8 bits,
  mantisse *m* sur 23 bits



$s \in \{0, 1\} \qquad e = e' - 127$

$0 \leq e' < 255, \quad -2^{23} \leq m \leq 2^{23} \implies x = \pm\, m\, 2^e$

$e' = 255, \; m = 0. \implies x = \pm\infty$

$e' = 255, \; m \neq 0. \implies x = \text{NaN}$

Représentation des nombres réels

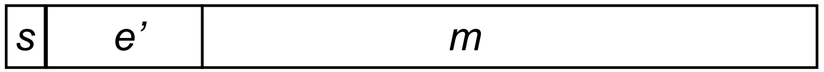
- les réels flottants sont des approximations des nombres réels

```
>>> 0.1 + 0.2 == 0.3
False
```

```
>>> 0.1 + 0.2
0.30000000000000004
```

- en virgule flottante : *IEEE 754 standard for floating points* [W. Kahan 1985]

  bit de signe *s*, exposant *e'* sur 11 bits,
  mantisse *m* sur 52 bits



$s \in \{0, 1\} \quad e = e' - 1023$

$0 \leq e' < 2047, \quad -2^{52} \leq m \leq 2^{52} \implies x = \pm\, m\, 2^e$

$e' = 2047, \; m = 0. \implies x = \pm\infty$

$e' = 2047, \; m \neq 0. \implies x = \text{NaN}$

Représentation des réels flottants

- les réels flottants en notation scientifique (en décimal)

```
>>> 1.234e3
1234
```

1.234×10^{3}

```
>>> 1.234e-13
.0000000000001234
```

1.234×10^{-13}

- les réels flottants sont représentés en binaire dans les processeurs

```
>>> 1.234e1
01100.01010111000010100011110101110000101000111 1
```

```
>>> 1.234e-3
0.0000000001010000110111110001010110100100101
```

```
>>> 0.5
000000000.1
```

```
>>> 0.75
00000000.11
```

[en puissances de 2 : $\cdots, 2^{3}, 2^{2}, 2^{1}, 2^{0} \cdot 2^{-1}, 2^{-2}, 2^{-3}, \cdots$]

Représentation des réels flottants

- remarque: certains nombres décimaux finis ne peuvent être représentés en flottant

```
>>> 0.1
0.0001100110011001100110011001100110011001100110011
```

```
>>> -0.2
1.1100110011001100110011001100110011001100110011111
```

- un réel peut avoir 2 représentations différentes selon la place du point

$$0.000214 \times 10^{2} = 0.214 \times 10^{-1}$$

- **représentation normalisée** (avec "01" ou "10" le plus à gauche dans la représentation binaire)

```
>>> 0.1
(0.1100110011001100110011001100110011001100110011000, -3)
```

```
>>> -0.2
(1.0011001100110011001100110011001100110011001100111100, -2)
```

[le bit de signe différencie les 2 cas]

- cas spécial: zéro est représenté par tous les bits signe, exposant, mantisse valant zéro

Erreur en calcul flottant

- erreur absolue d'arrondi par rapport au nombre réel exact (*p* chiffres significatifs après le point)

$$1/2 \times 2^{-p} \times 2^{e}$$

- erreur relative ou encore précision ϵ de la machine

$$\epsilon = 1/2 \times 2^{-p}$$

```
def precision () :
    a = 1.0
    while ((a + 1.0) - a) - 1.0 == 0.0 :
        a = 2 * a
    return a

epsilon = precision()
print (epsilon)
print (math.log2 (epsilon))
```

```
9007199254740992.0
53.0
```

- les opérations flottantes produisent des erreurs

- l'opération la plus redoutable est la soustraction

Erreur en calcul flottant

- pour minimiser l'erreur, on change les formules par exemple quand $4ac \ll b^2$

$$\frac{-b+\sqrt{b^2-4ac}}{2a} \qquad \frac{2c}{-b-\sqrt{b^2-4ac}}$$

$$\frac{-b-\sqrt{b^2-4ac}}{2a} \qquad \frac{2c}{-b+\sqrt{b^2-4ac}}$$

- **Règle**: un test entre flottants n'a aucun sens si l'erreur de $|a-b|$ n'est pas estimée

```
if a > b :
    ...
else :
    ...
```

- série de livres Numerical recipes [1986 – 2007]

- bugs célèbres dus au calcul flottant: missile **Patriot** (1991), plate-forme **Sleipner A** (1991), **FDIV Intel** (1994), **Ariane 501** (1996), etc.

Exercice L'addition est-elle commutative ou associative en calcul flottant ?

Bibliothèque numpy

- analyse numérique en Python
- vrais tableaux multi-dimensionnels
- beaucoup de fonctions utiles en calculs numériques
- efficace et fonctionne sur les architectures parallèles
- *open source*
- tutoriel simple en `https://www.w3schools.com/python/numpy/default.asp`

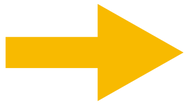
Bibliothèque numpy

- analyse numérique en Python

```
import numpy as np
```

- vrais tableaux

```
a = [1, 2, 3, 4, 5]
b = np.array ([1, 2, 3, 4, 5])
print (a, b)
print (type(a), type(b))
```

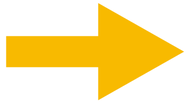


```
[1, 2, 3, 4, 5] [1 2 3 4 5]
<class 'list'> <class 'numpy.ndarray'>
```

N-dimensional array

- vrais tableaux multidimensionnels

```
c = np.array ([[1, 2, 3, 4], [5, 6, 7, 8]])
print (c)
print (c[1, 2])
print (type(c))
print (b.dtype, c.dtype)
print (b.ndim, c.ndim)
print (b.shape, c.shape)
```



```
[[1 2 3 4]
 [5 6 7 8]]
7
<class 'numpy.ndarray'>
int64 int64
1 2
(5,) (2, 4)
```

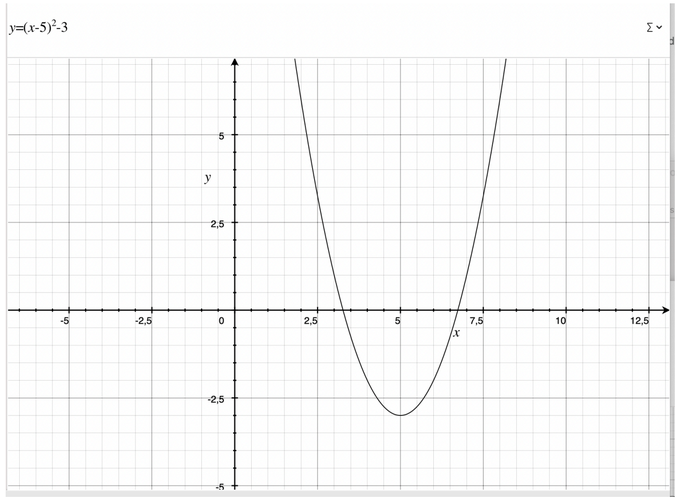
- les tableaux occupent une place contigüe en mémoire, et sont homogènes (tous leurs éléments ont un même type)

- attention aux ***alias*** ! on peut copier un tableau avec la méthode copy()

```
d = c
e = c.copy()
```

Descente du gradient

• trouver le maximum ou minimum d'une fonction *f (x)*




```
def f(x) :
    return (x-5)**2 - 3

def df (x) :
    return 2*(x-5)

alpha = 0.1

def min (f, x0) :
    x = x0
    for i in range (30) :
        x -= alpha * df(x)
    return x

print (min (f, 3))
```

• on part d'un point sur la courbe `f(x)` et on descent vers le minimum
en fonction de la pente de la courbe à ce point

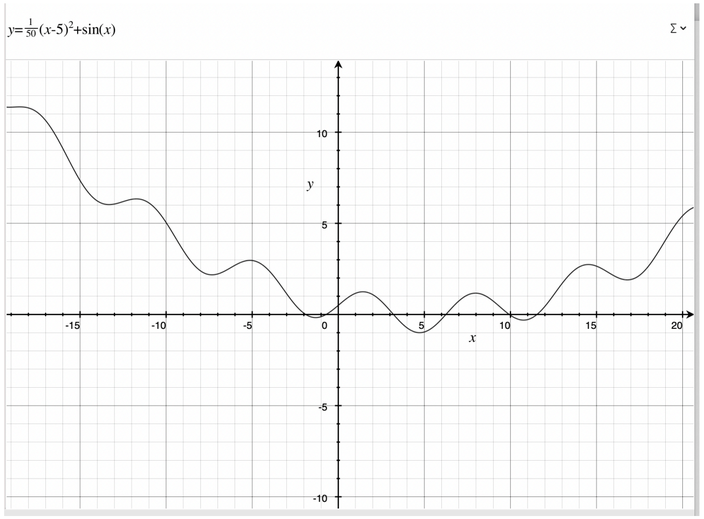
• la pente se calcule avec la dérivée `df(x)` de la fonction en ce point

• on avance progressivement en choisissant un bon taux de progression `alpha`

• ce taux `alpha` ne doit être ni trop petit, ni trop grand

Descente du gradient

- la descente du gradient peut ne trouver qu'un maximum ou minimum local



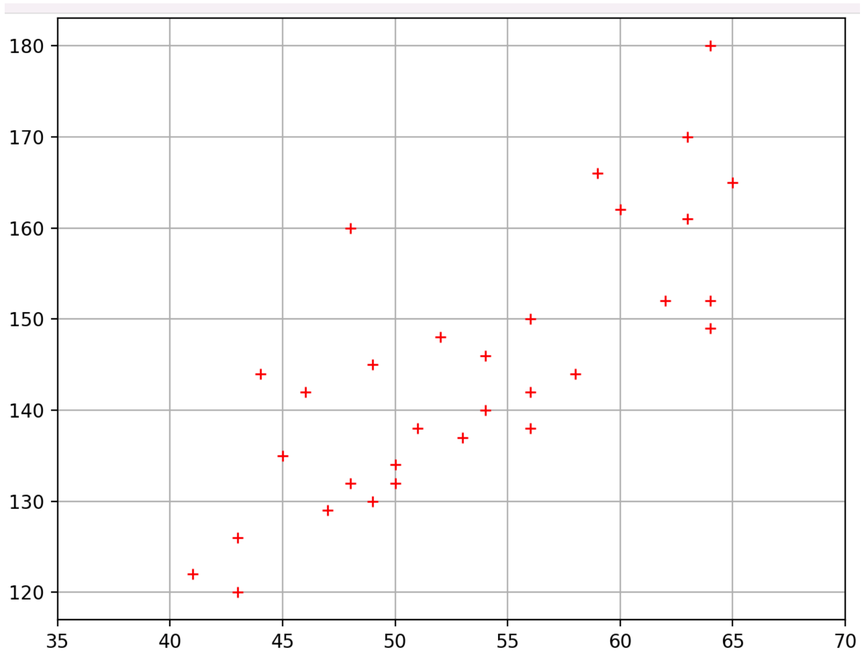

```python
def f(x) :
    return 0.02 * (x-5)**2 + math.sin(x)

def df (x) :
    return 2*(x-5)

alpha = 0.1

def min (f, x0) :
    x = x0
    for i in range (30) :
        x -= alpha * df(x)
    return x

print (min (f, 10))
```

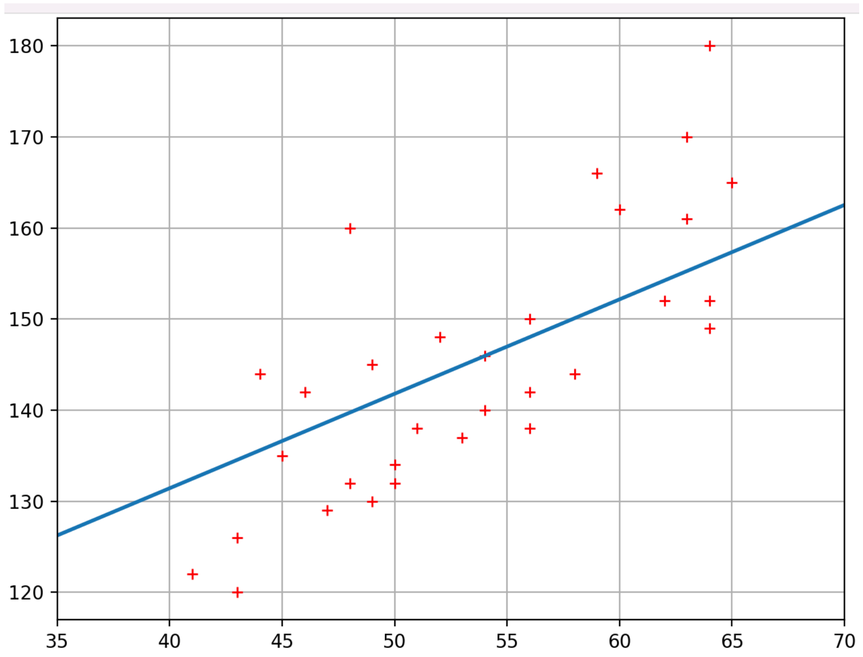
- ce minimum est global dans le cas d'une fonction parabolique

$$y = (x - a)^2$$

Régression linéaire

- on se donne un jeu de données (ici la pression artérielle en fonction de l'âge)
- on cherche une approximation linéaire de ces données

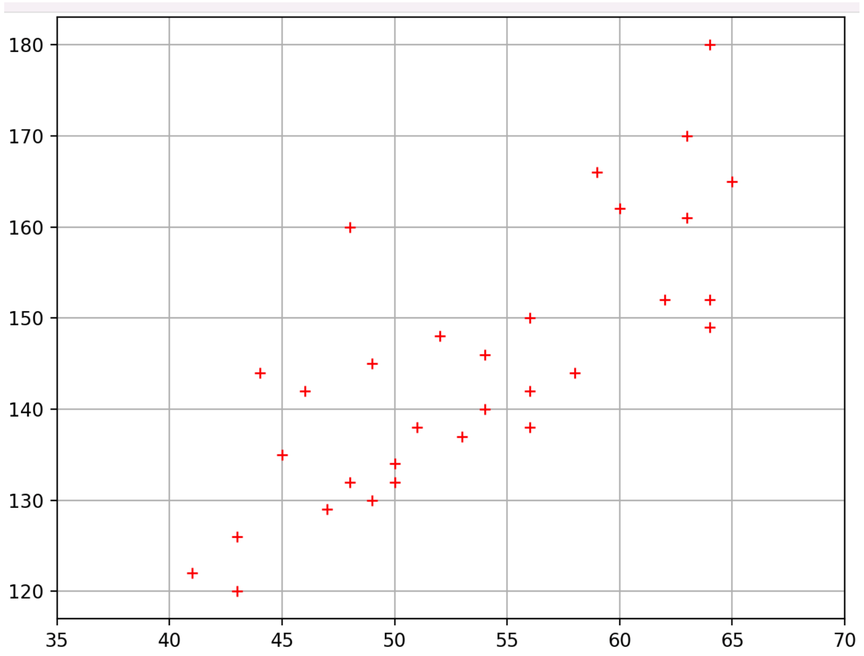




Régression linéaire

• le jeu de *n* données $(x^{(i)}, y^{(i)})$

```
dataset = [ (41, 122), (43, 120), (43, 126), (44, 144), (45, 135),
 (46, 142), (47, 129), (48, 132), (48, 160), (49, 130), (49, 145),
 (50, 132), (50, 134), (51, 138), (52, 148), (53, 137), (54, 140),
 (54, 146), (64, 149), (56, 138), (56, 142), (56, 150), (58, 144),
 (59, 166), (60, 162), (62, 152), (63, 161), (63, 170), (64, 152),
 (64, 180), (65, 165) ]
```



$x^{(0)} = 41,\ y^{(0)} = 122,\ x^{(1)} = 43,\ y^{(1)} = 120,\ x^{(2)} = 43,\ y^{(2)} = 126, \cdots$

• on cherche une fonction $f(x) = \theta_0 + \theta_1 x$
telle que la valeur absolue de la différence $|f(x^{(i)}) - y^{(i)}|$
soit la plus petite possible pour tous les $(x^{(i)}, y^{(i)})$

```
def f (x, theta0, theta1) :
    return theta0 + theta1 * x
```

• la différence à minimiser est la somme :

$$J = \sum_{i=0,n-1} \frac{1}{2} (f(x^{(i)}) - y^{(i)})^2$$

[calcul des moindres carrés]

```
def J(dataset) :
    global theta0, theta1
    s = 0
    for (x, y) in dataset :
        s += 0.5 * (f (x, theta0, theta1) - y)**2
    return s
```

• *J* est aussi appelé le **coût** de l'approximation linéaire,
il dépend du jeu de données et des 2 **paramètres** θ_0 et θ_1

Régression linéaire

- on calcule les dérivées (partielles) du coût *J* par rapport à θ_0 et θ_1

$$\frac{\partial J}{\partial \theta_0} = \sum_{i=0,n-1} f(x^{(i)}) - y^{(i)}$$

$$\frac{\partial J}{\partial \theta_1} = \sum_{i=0,n-1} x^{(i)} \left(f(x^{(i)}) - y^{(i)}\right)$$

```
def derJ0 (dataset) :
    global theta0, theta1
    s = 0
    for (x, y) in dataset :
        s += (f (x, theta0, theta1) - y)
    return s
```

```
def derJ1 (dataset) :
    global theta0, theta1
    s = 0
    for (x, y) in dataset :
        s += x * (f(x, theta0, theta1) - y)
    return s
```

- on minimise le coût *J* par une descente du gradient en faisant varier les 2 paramètres θ_0 et θ_1

```
def descent (dataset) :
    global theta0, theta1, alpha
    for i in range (100) :
        theta0 -= alpha * derJ0 (dataset)
        theta1 -= alpha * derJ1 (dataset)
```

- on choisit un bon *alpha* et 2 valeurs initiales arbitraires pour θ_0 et θ_1

Régression linéaire

- bibliothèque matplotlib
- régression linéaire avec plus que 2 paramètres
- solutions exactes
- apprentissage
- réseau de neurones