

Module : Informatique

Classes: 1^{ères} Années MP – PC –T

Chapitre IX: Simulation numérique

Enseignant : Dr. Khemaies GHALI (ghali_khemaies@yahoo.fr)

Page Facebook: <https://www.facebook.com/groups/ipein.info/>

Année universitaire 2017/2018

Plan

1. Utilisation des classes « array » et « ndarray » de la bibliothèque « numpy »
2. Simulation d'expériences et traçage de courbes « matplotlib »
3. Résolution d'équations différentielles du premier ordre
4. Méthodes de recherche du zéro d'une fonction
5. Méthodes d'intégrations

Partie 1: Utilisation des classes « array » et « ndarray » de la bibliothèque « numpy »

Modules NumPy et SciPy (1/2)

- ❑ La **bibliothèque Numpy** contient des fonctions essentielles pour traiter:
 - les tableaux, les matrices et les opérations de type algèbre linéaire avec Python.
- ❑ La **bibliothèque SciPy** contient des fonctions supplémentaires pour l'optimisation de calculs, des fonctions spéciales, etc.
- ❑ Les deux bibliothèques adoptent une **implémentation proche du hardware**, et donc beaucoup plus efficace pour les calculs:
 - Prévu pour le **calcul scientifique**,
 - Comprennent des **modules écrits en C et en Fortran** de manière à les rendre aussi rapides que possible.

Modules NumPy et SciPy (2/2)

- ❑ L'un des points communs fondamentaux de **NumPy** et de **SciPy** est leur aptitude à gérer les tableaux et les matrices.
 - La possibilité de **créer des tableaux à partir de listes** Python (*list* ou *[]*) grâce à la fonction objet **array()**:
- ❑ NumPy définit de **nouveaux types**, en particulier le type **ndarray**.
- ❑ Le type **array** est un **tableau multidimensionnel** dont tous les **éléments sont de même type** (en général ce sont des nombres):
 - Les éléments sont indicés par un **T-uple** d'entiers positifs,
 - Dans NumPy les **dimensions s'appellent « axes »** et le **nombre de dimensions « rank »**.

Module NumPy: Exemples (1/5)

```
1. >>> import numpy as np
2. # création à partir d'une liste
3. >>> A = np.array([[1,2,3],[4,5,6]])
4. >>> A
5. array([[1, 2, 3],
6.        [4, 5, 6]])
7. >>> A.ndim # tableau a 2 dimensions
8. 2
9. >>> A.shape # 2 lignes et 3 colonnes
10. (2, 3)
11. >>> A.dtype # tableau d'entiers
12. dtype('int64')
13. # tableau de flottants car une case est flottante
14. >>> A = np.array([[1, 2, .3], [4, 5, 6]])
15. >>> A.dtype # valeur est de type float
16. dtype('float64')
17. >>>
```

Module NumPy: Exemples (2/5)

- ❑ Les types de bases sont plus complets. On peut par exemple **avoir des entiers codés sur 8, 16 ou 32 bits au lieu de 64 bits** (voir annexe 1).
- ```
1. # tableau d'entiers 8 bits
2. >>> A = np.array([1,1], dtype = np.int8)
3. >>> A[0] = 127 # 127 = 01111111 en base 2 (en comp. à 2)
4. >>> A
5. array([127, 1], dtype = int8)
6. >>> A.dtype
7. dtype('int8')
8. >>> A[0] = A[0] + 1
9. >>> A
10. array([-128, 1], dtype = int8)
11. >>> A = np.array([1,1], dtype=int) # tableau d'entiers
12. >>> A.dtype #int32 ou int64 ça dépend de la machine!
13. dtype('int64')
```

## Module NumPy: Exemples (3/5)

```
1. >>> A[0]=9223372036854775807 #valeur maximum sur 64 bits
2. >>> A
3. array([9223372036854775807, 1])
4. >>> A[0] = A[0] + 1
5. __main__:1: RuntimeWarning: overflow encountered in
 long_scalars

6. >>> A = np.array([9223372036854775808,1])
7. >>> A.dtype # pas un "int64" c'est un "objet"!
8. dtype('object')
9. >>> A[0]
10. 9223372036854775808
```

## Module NumPy: Exemples (4/5)

□ En pratique, on crée rarement des tableaux à la main:

```
1. >>> a = np.arange (10) # tab. 1D: 0, 1, ... n-1
2. >>> a
3. array ([0 , 1, 2, 3, 4, 5, 6, 7, 8, 9])
4. >>> type(a)
5. <class 'numpy.ndarray'>
6. >>> np.arange (1, 9, 2)
7. array ([1 , 3, 5, 7])
8. >>> np.linspace(0, 1, 6) # start, end, nbr. des points
9. array ([0. , 0.2 , 0.4 , 0.6 , 0.8 , 1.])
10. >>> np.linspace (0,1,5, endpoint = False) # start, end,
 nombre des points
11.array ([0., 0.2, 0.4, 0.6, 0.8])
12.#table avec des réels aléatoires dans [0,1[
13. >>> np.random.rand(2,3)
14.array([[0.35797599, 0.03598783, 0.96642928],
15. [0.9348623 , 0.70968864, 0.3997335]])
```

## Module NumPy: Exemples (5/5)

```
1. >>> np.ones ((2,3)) # float par défaut
2. array ([[1., 1., 1.] ,
3. [1., 1., 1.]])
4. >>> np.zeros ((3 ,2)) # float par défaut
5. array ([[0., 0.] ,
6. [0., 0.] ,
7. [0., 0.]])
8. >>> I = np.eye (3); I # I: Matrice identité, float
9. array ([[1., 0., 0.] ,
10. [0., 1., 0.] ,
11. [0., 0., 1.]])
12. >>> d = np.diag (np.array([1 , 2, 3, 4])); d
13.array ([[1, 0, 0, 0],
14. [0, 2, 0, 0],
15. [0, 0, 3, 0],
16. [0, 0, 0, 4]])
17. >>> e = np.diag (d); e #retourne le diagonal de mat. d
18.array ([1 , 2, 3, 4])
```

## NumPy: Quelques opérations simples

| Opération                       | Description                     |
|---------------------------------|---------------------------------|
| +, -, *, /, **                  | Opérations termes à termes      |
| numpy.dot(a,b)                  | Multiplication matricielle      |
| numpy.cos(a), numpy.exp(a), ... | Opérations termes à termes      |
| a.T ou a.transpose()            | Transposition                   |
| a.trace() ou np.trace(a)        | Trace de a (somme sur le diag.) |
| a.min(), a.sum, ...:            | Minimum, somme des élém. de a   |

```
1. >>> a = np.array ([[1 , 2, 3], [4, 5, 6]]); a
2. array ([[1 , 2, 3],
3. [4, 5, 6]])
4. >>> a > 2 # c'est toujours termes à termes
5. array ([[False , False , True],
6. [True , True , True]], dtype = bool)
7. >>>
```

## NumPy: Indices, extraction de sous matrice (1/6)

```
1. >>> a = np.arange(10,20); a
2. >>> array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
3. >>> b = a[[2, 4, 2]]; b # [2, 4, 2] est une liste python
4. array ([12, 14, 12])
5. >>> a[[9 , 2]] = -10; a # init. de a[9] et a[2] à -10
6. array([10, 11, -10, 13, 14, 15, 16, 17, 18, -10])
7. >>> a = np.arange(12).reshape ((4,3)); a
8. array ([[0, 1, 2],
9. [3, 4, 5],
10. [6, 7, 8],
11. [9, 10, 11]])
12. >>> np.size(a), np.shape(a)[0], np.shape(a)[1]
13. (12, 4, 3) #nbr éléments, nbr lignes et nbr colonnes
14. >>> b = a.T; b
15. array ([[0, 3, 6, 9],
16. [1, 4, 7, 10],
17. [2, 5, 8, 11]])
```

## NumPy: Indices, extraction de sous matrice (2/6)

```
1.>>> c = a[: :2, :]; c #ligne avec pas de 2 + tt
 col.
2.array ([[0 , 1, 2],
3. [6, 7, 8]])
4.>>> c[0,1] = 10; c # c[i,j] équi. c[i][j]
5.array ([[0, 10, 2],
6. [6, 7, 8]])
7.>>> a # c'est toujours du python !
8.array ([[0, 10, 2],
9. [3, 4, 5],
10. [6, 7, 8],
11. [9, 10, 11]])
12.>>> b #déjà b = a.T
13.array ([[0, 3, 6, 9],
14. [10, 4, 7, 10] ,
15. [2, 5, 8, 11]])
```

## NumPy: Indices, extraction de sous matrice (3/6)

```
1. # Pour vraiment copier, il faut utiliser la méthode copy
2.>>> a = np.arange (10)
3.>>> b = a[: :2].copy ()
4.>>> b[0] = 12
5.>>> b
6.array ([12 , 2, 4, 6, 8])
7.>>> a
8.array ([0 , 1, 2, 3, 4, 5, 6, 7, 8, 9])
9. #utilisation de la fonction nonzero()
10.>>> a = np.array([[2 , 3, 4],[4, 5, 6]])
11.>>> i = np.nonzero (a > 3); i # i est un tuple qui
 contient les indices des valeurs recherchées
12.(array ([0 , 1, 1, 1]), array ([2 , 0, 1, 2]))
13.>>> b = a[i]; b
14.array ([4 , 4, 5, 6])
15.>>> a[i] = 0; a
16.array ([[2 , 3, 0],
17. [0, 0, 0]])
```

## NumPy: Indices, extraction de sous matrice (4/6)

```
1. # concatination des tableau
2.>>> a = np.array ([[2 , 3, 0],[0, 0, 0]])
3.>>> np.concatenate ((a, np.eye(3))) #sur les lignes
4.array ([[2., 3., 0.] ,
5. [0., 0., 0.] ,
6. [1., 0., 0.] ,
7. [0., 1., 0.] ,
8. [0., 0., 1.]])
9.>>> np.concatenate ((a, np.eye(2)), 1) #sur les colonnes
10.array ([[2., 3., 0., 1., 0.] ,
11. [0., 0., 0., 0., 1.]])
12. # des opérations de diffusions "Broadcasting"
13. >>> x = np.arange (4); x
14. array ([0 , 1, 2, 3])
15. >>> x.ndim # affiche dimension du tableau
16. 1
17. >>> x.shape # affiche nbr. éléments par dimension
18. (4 ,)
```

## NumPy: Indices, extraction de sous matrice (5/6)

```
1. >>> a = np.array([[1,2,3], [4,5,6]])
2. >>> np.reshape(a, 6) #ou a.reshape(6)
3. array([1, 2, 3, 4, 5, 6])
4. >>> np.reshape(a, (2,3)) # ou a.reshape(2,3)
5. array([[1, 2, 3],
6. [4, 5, 6]])
7. >>> np.reshape(a, (3,-1)) # "-1" est remplacé par 2
8. array([[1, 2],
9. [3, 4],
10. [5, 6]])
11.>>> b = np.ones(5); b
12. array([1., 1., 1., 1., 1.])
13.>>> c = np.ones((3,4)); c
14. array([[1., 1., 1., 1.],
15. [1., 1., 1., 1.],
16. [1., 1., 1., 1.]])
```

## NumPy: Indices, extraction de sous matrice (6/6)

```
1. >>> b + c
2. ValueError: operands could not be broadcast together
 with shapes (5) (3,4)

3. >>> x = np.arange(9).reshape((3,3)); x
4. array([[0, 1, 2],
5. [3, 4, 5],
6. [6, 7, 8]])
7. >>> y = x[:,1]; y # extraire une colonne
8. array([1, 4, 7])
9. >>> y.ndim; y.shape
10. 1
11. (3,)
12. >>> z = x[:,1:2]; z
13. array([[1],
14. [4],
15. [7]])
16. >>> y - z
17. array([[0, 3, 6],
18. [-3, 0, 3],
19. [-6, -3, 0]], dtype=int32)
```

## NumPy: Type Matrix

```
1. # Il existe aussi un type matrice qui a toujours 2 dimensions
2. >>> A = np.matrix ([np.arange(4), np.arange(4,8)]); A
3. matrix ([[0 , 1, 2, 3],
4. [4, 5, 6, 7]])
5. >>> b = np.matrix ([1, 1, 1, 1])
6. >>> b.ndim; A.ndim #une matrice a toujours 2 dimensions!
7. 2
8. 2
9. >>> A.shape; b.shape
10. (2, 4)
11. (1, 4)
12. >>> A*b # * est la mult. matricielle sur les matrices
13. Traceback (most recent call last) :
14. ...
15. ValueError : objects are not aligned
16. >>> A*b.T
17. matrix ([[6] ,
18. [22]])
```

## Partie 2: Simulation d'expériences et traçage de courbes (Module Matplotlib)

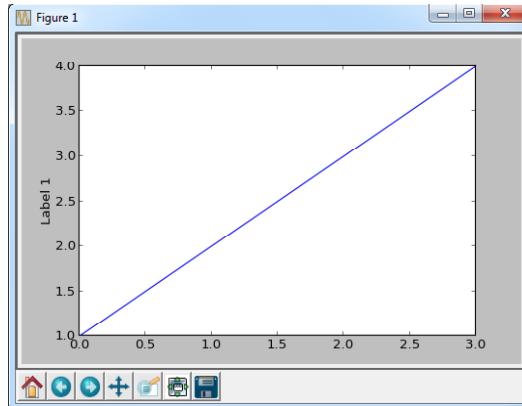
## Introduction

❑ **Matplotlib** est une bibliothèque python qui dessine des graphiques.

- Nul besoin de connaissances en interfaces graphiques pour créer un graphique dynamique avec possibilité de zoom et de sauvegarde par l'utilisateur.
- Il est d'ailleurs possible de sauvegarder les graphiques en format matriciels comme le PNG, JPEG, etc. et vectoriels comme le PDF et le SVG.

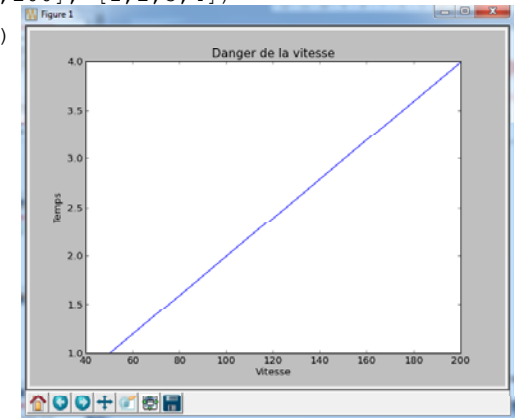
## Matplotlib: Exemple1

```
1. import matplotlib.pyplot as plt
2. plt.plot([1,2,3,4])
3. plt.ylabel('Label 1')
4. plt.show()
```



## Matplotlib: Exemple2

```
1. import matplotlib.pyplot as plt
2. plt.title("Danger de la vitesse")
3. plt.plot([50,100,150,200], [1,2,3,4])
4. plt.xlabel('Vitesse')
5. plt.ylabel('Temps')
6. plt.show()
```

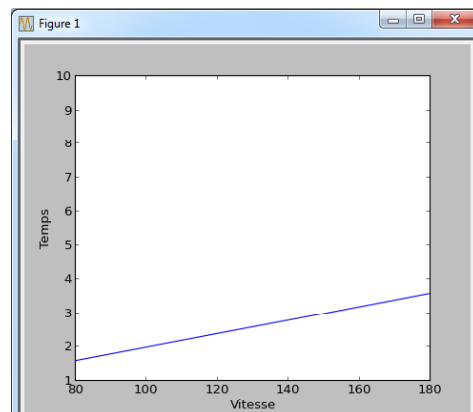


## Matplotlib: Exemple3

Il est possible d'encadrer son graphique à l'aide de la méthode

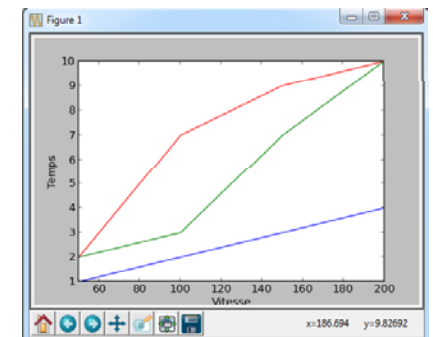
**`axis(xmin, xmax, ymin, ymax)`**

```
1. ...
2. plt.ylabel('Temps')
3. plt.axis([80, 180, 1, 10])
4. plt.show()
```



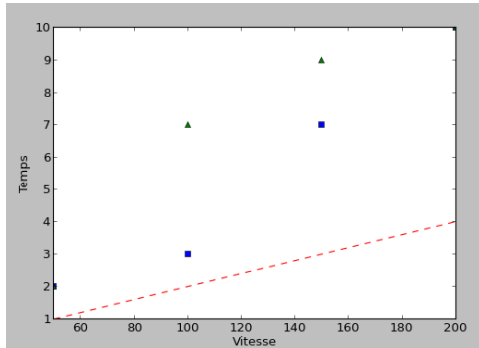
## Matplotlib: Plusieurs courbes

```
1. # Il est possible de superposer les courbes
2. plt.plot([50,100,150,200], [1,2,3,4])
3. plt.plot([50,100,150,200], [2,3,7,10])
4. plt.plot([50,100,150,200], [2,7,9,10])
5. ...
6. #on peut utiliser aussi la
7. # syntaxe suivante:
8. plt.plot([50,100,150,200],
9. [1,2,3,4],
10. [50,100,150,200],
11. [2,3,7,10],
12. [50,100,150,200],
13. [2,7,9,10])
```



## Matplotlib: Les visuels des courbes

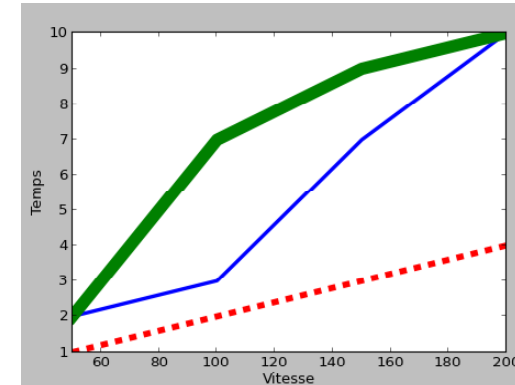
1. # Il est possible de changer la couleur et la forme des courbes:
2. # Courbes possibles: -, --, -., :
3. # Couleurs possibles: b g r c m y k w
4. plt.plot([50,100,150,200], [1,2,3,4], "r--")
5. plt.plot([50,100,150,200], [2,3,7,10], "bs")
6. plt.plot([50,100,150,200], [2,7,9,10], "g^")



| character | color   |
|-----------|---------|
| 'b'       | blue    |
| 'g'       | green   |
| 'r'       | red     |
| 'c'       | cyan    |
| 'm'       | magenta |
| 'y'       | yellow  |
| 'k'       | black   |
| 'w'       | white   |

## Matplotlib: Largeur des courbes

1. # On peut changer la largeur des courbes comme ceci:
2. plt.plot([50,100,150,200], [1,2,3,4], "r--", linewidth=5)
3. plt.plot([50,100,150,200], [2,3,7,10], "b", linewidth=3)
4. plt.plot([50,100,150,200], [2,7,9,10], "g", linewidth=10)



IPEIN 2017/2018

Informatique - Chapitre IX

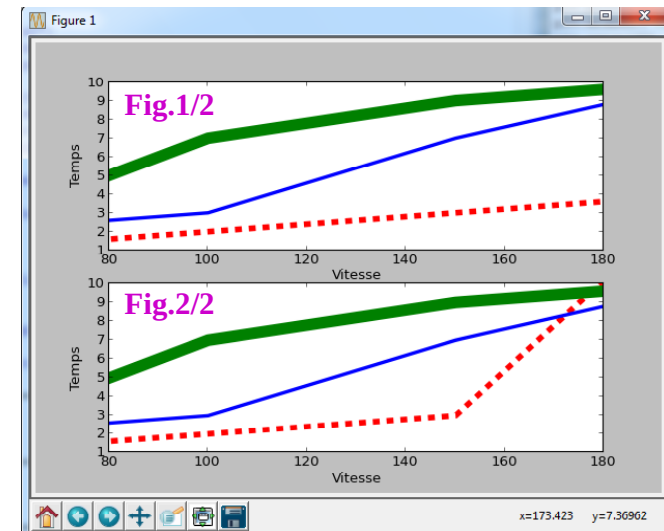


26/92

## Matplotlib: Plusieurs graphiques (1/2)

1. # Il est possible de mettre plusieurs graphiques
2. # sur une même image:
3. plt.subplot(211) # Fig.1/2 - avec 2 lignes et une colonne
4. plt.plot([50,100,150,200], [1,2,3,4], "r--", linewidth=5)
5. plt.plot([50,100,150,200], [2,3,7,10], "b", linewidth=3)
6. plt.plot([50,100,150,200], [2,7,9,10], "g", linewidth=10)
7. plt.xlabel('Vitesse'); plt.ylabel('Temps')
8. plt.axis([80, 180, 1, 10])
9. plt.subplot(212) # Fig.2/2 - 2 rows, 1 column
10. plt.plot([50,100,150,200], [1,2,3,15], "r--", linewidth=5)
11. plt.plot([50,100,150,200], [2,3,7,10], "b", linewidth=3)
12. plt.plot([50,100,150,200], [2,7,9,10], "g", linewidth=10)
13. plt.xlabel('Vitesse'); plt.ylabel('Temps')
14. plt.axis([80, 180, 1, 10])
15. plt.show()

## Matplotlib: Plusieurs graphiques (2/2)



IPEIN 2017/2018

Informatique - Chapitre IX



28/92

IPEIN 2017/2018

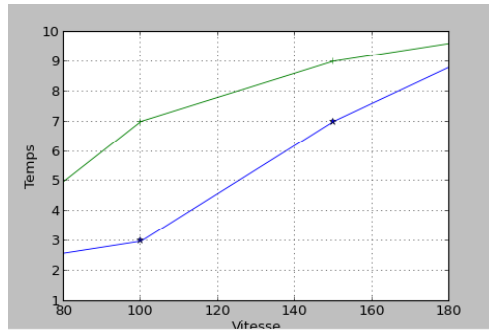
Informatique - Chapitre IX



27/92

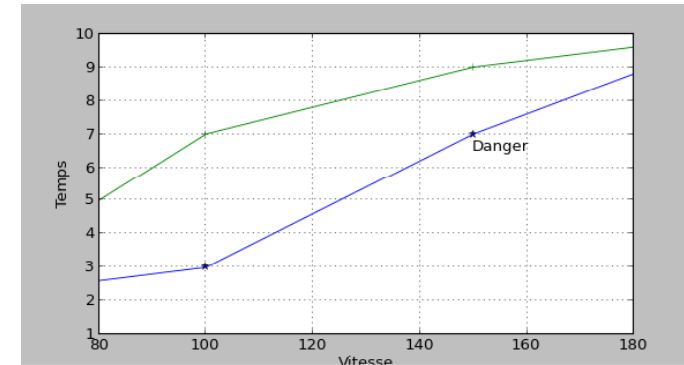
## Matplotlib: Grille

1. # on peut ajouter une grille (et des marqueurs):
2. # avec marqueurs possibles : o, +, ., x, \* et ^
3. plt.grid(True)
4. plt.plot([50,100,150,200], [2,3,7,10], "b", linewidth=0.8, marker="\*")
5. plt.plot([50,100,150,200], [2,7,9,10], "g", linewidth=0.8, marker="+")



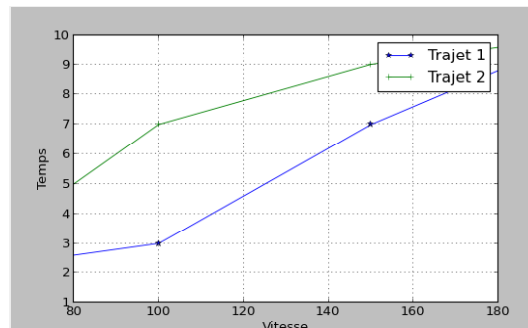
## Matplotlib: Ajout du texte

1. # Il est possible d'ajouter du texte en indiquant
2. # les coordonnées:
3. . . .
4. plt.text(150, 6.5, 'Danger')
5. . . .



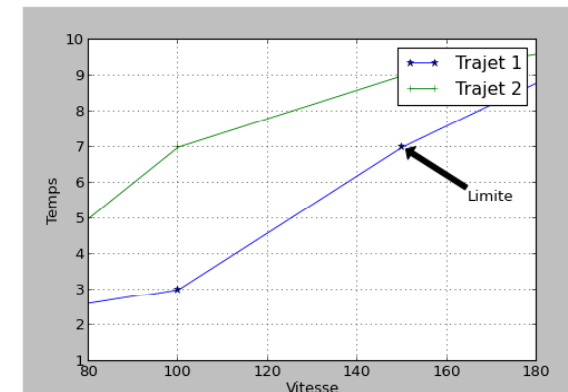
## Matplotlib: Légende

1. # Pour ajouter une légende pour les courbes
2. plt.grid(True)
3. plt.plot([50,100,150,200], [2,3,7,10], "b", linewidth=0.8, marker="\*", label="Trajet 1")
4. plt.plot([50,100,150,200], [2,7,9,10], "g", linewidth=0.8, marker="+", label="Trajet 2")
5. plt.axis([80, 180, 1, 10])
6. plt.xlabel('Vitesse')
7. plt.ylabel('Temps')
8. plt.legend()
9. plt.show()



## Matplotlib: Une flèche

1. # Pour ajouter une flèche descriptive sur le graphique
2. . . .
3. plt.annotate('Limite', xy=(150, 7), xytext=(165, 5.5),
4. arrowprops={'facecolor':'black', 'shrink':0.05})
5. . . .

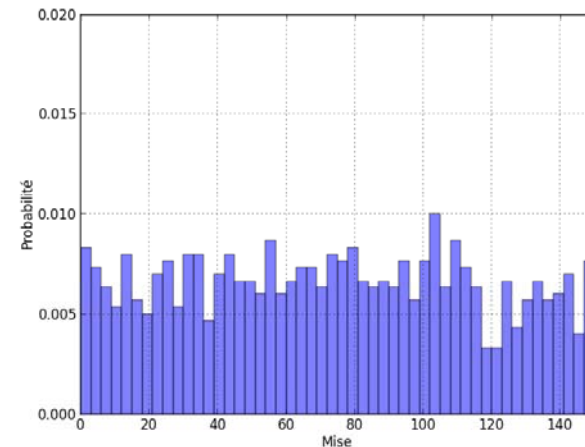


## Matplotlib: Les histogrammes (1/2)

- ❑ Pour créer un histogramme on utilise la **méthode « hist »**.
- ❑ On peut lui passer des données brutes et il s'occupera de faire les calculs nécessaires à la présentation du graphique.

```
1. # On a prit ici l'exemple de 1000 tirage au
2. # sort (random) des valeurs entre 0 et 150:
3. import matplotlib.pyplot as plt
4. import random
5. x = [random.randint(0,150) for i in range(1000)]
6. n, bins, patches = plt.hist(x, 50, normed=1,
7. facecolor='b', alpha=0.5) # La valeur 50 indique
8. # le nombre de barres à afficher.
9. plt.xlabel('Mise')
10. plt.ylabel('Probabilité')
11. plt.axis([0, 150, 0, 0.02])
12. plt.grid(True)
13. plt.show()
```

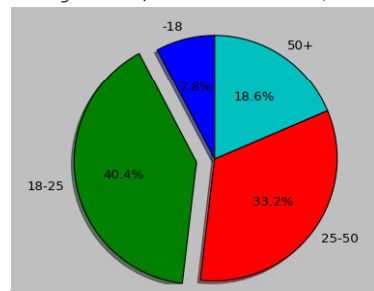
## Matplotlib: Les histogrammes (2/2)



## Matplotlib: Diagramme circulaire (1/2)

- ❑ Un des graphiques les plus utilisés doit être le **diagramme circulaire (ou camembert)** :

```
1. import matplotlib.pyplot as plt
2. name = ['-18', '18-25', '25-50', '50+']
3. data = [5000, 26000, 21400, 12000]
4. explode=(0, 0.15, 0, 0)
5. plt.pie(data, explode=explode, labels=name,
6. autopct='%1.1f%%', startangle=90, shadow=True)
7. plt.axis('equal')
8. plt.show()
```



## Matplotlib: Diagramme circulaire (2/2)

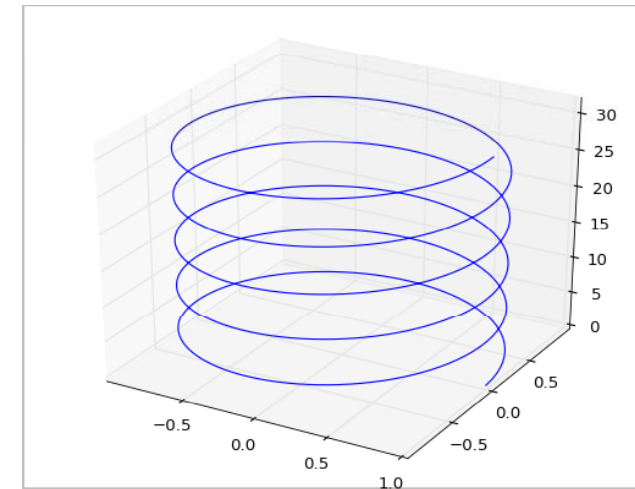
- ❑ Les données « **explode** » indiquent les espaces entre les parts.
  - Nous avons fait en sorte dans cet exemple de séparer la part 18-25 des autres part.
- ❑ Le paramètre « **autopct** » indique le pourcentage réel par rapport aux données qu'on lui a fournit.
- ❑ Enfin, la méthode **axis("equal")** rend le diagramme parfaitement circulaire.
  - En son absence lorsque vous redimensionnez le graphique celui-ci s'adapte au conteneur ce qui peut lui donner une forme elliptique.

## Graphiques à 3 dimensions (1/2)

- ❑ Il est possible de tracer des courbes dans l'espace avec la **bibliothèque Axes3D**. Par exemple une hélice :

```
1. import matplotlib.pyplot as plt
2. import numpy as np
3. from math import pi
4. from mpl_toolkits.mplot3d import Axes3D
5. t = np.linspace(0,10*pi,500)
6. fig = plt.figure()
7. ax = fig.gca(projection='3d')
8. plt.plot(np.cos(t),np.sin(t),t,label="helice")
9. plt.show()
```

## Graphiques à 3 dimensions (2/2)



## Partie 3: Résolution d'équations différentielles du premier ordre (Méthode d'Euler - scipy)

## Introduction

- ❑ Dans ce cours, nous étudions les **équations différentielles linéaires du premier ordre** (EDL1O).
- ❑ Les équations différentielles ont un lien évident avec **l'intégration numérique**.
  - Donc on va utiliser dans cette section:
    - ❖ Un découpage de l'intervalle de résolution en « **n** » **morceaux**,
    - ❖ Puis, **une approximation de la solution** sur chacun de ces intervalles.
  - C'est le principe de base de la **méthode d'Euler**, qui est la principale méthode de résolution numérique d'équations différentielles,
- ❑ Euler, **méthode numérique** permettant d'obtenir une approximation de la solution d'un **tel problème de Cauchy**.

## Équations différentielles linéaires du premier ordre - EDL1O (1/4)

□ **Définition 1:** Une équation différentielle est une équation qui relie une fonction « f » à une ou plusieurs de ses dérivées.

**Exemple :**  $f(x) = \alpha f'(x) + \beta f''(x)$

**Remarque 1:** Pour simplifier l'écriture des équations différentielles, on remplace couramment f(x) par « y » et f'(x) par « y' »

$f(x) = \alpha f'(x) + \beta f''(x)$  devient  $y = \alpha y' + \beta y''$

□ **Définition 2:** Une équation différentielle est une équation dont l'inconnue est une fonction « y » (réelle ou complexe, même si nous traiterons surtout le cas réel dans ce chapitre), et faisant intervenir les dérivées successives y', y'', ..., y(n) de la fonction « y ».  
L'équation est dite « d'ordre n » si la dérivée d'ordre le plus élevé de « y » apparaissant dans l'équation est y(n).

## EDL1O (2/4)

□ **Définition 3:** Une équation différentielle est linéaire si elle s'écrit sous la forme :

$$a_n y^{(n)} + a_{n-1} y^{(n-1)} + \dots + a_1 y' + a_0 y = b$$

où  $a_0, a_1, \dots, a_n$  et  $b$  sont des fonctions.

□ **Définition 4:** La fonction « b » est appelé le **second membre**.

➤ Si « b » est nulle, alors l'équation est dite **homogène** ou **sans second membre**.

➤ Si  $a_0, a_1, \dots, a_n$  sont des constantes, on parle d'**équation différentielle linéaire à coefficients constants**.

□ **Définition 5:** Une **courbe intégrale** associée à une équation différentielle est une **courbe représentative de la fonction solution** « y » de l'équation différentielle.

## EDL1O (3/4)

□ **Remarque 2:**

Les équations différentielles considérées doivent être écrites sous la forme :  $y' = F(y, t)$

où F est une fonction à **deux variables** « y » et « t ».

□ Par exemple, pour l'équation différentielle :

$$y'(t) + 2ty(t) = 1$$

on définit :  $F(y, t) = 1 - 2ty$  ce qui s'écrit en Python :

```
1. def F(y, t):
2. return 1-2*t*y
```

## EDL1O (4/4)

□ **Remarque 3:** Même si, comme c'est parfois le cas, on écrit l'équation sous la forme :  $y'(t) + 2ty(t) = 1$

il ne faut surtout pas définir la fonction F en écrivant :

```
1. def F(y, t):
2. return 1-2*t*y(t) #INCORRECT
```

□ **Remarque 4:** En mathématiques et en physique, l'habitude est plutôt de considérer les équations différentielles sous la forme

$$y' = F(t, y) \text{ et non } y' = F(y, t).$$

□ Mais en Python, la fonction « odeint », que nous verrons par la suite, utilise l'écriture  $y' = F(y, t)$

□ Par souci de simplicité, nous nous conformerons à cette écriture dans ce cours.

## Cadre du problème de Cauchy

- Donc, on cherche à résoudre un problème numérique simple dit **problème de Cauchy** du type :

$$\begin{cases} y' = F(t, y) \\ y(t_0) = y_0 \end{cases}$$

où :

- ❖  $y$  est l'inconnue ;
- ❖  $y$  est une fonction dérivable définie sur un intervalle  $I$  de  $\mathbb{R}$  ;
- ❖  $t_0 \in I$  ;
- On possède donc **une équation différentielle et une condition initiale**.
- Le **théorème de Cauchy Lipschitz** assure l'existence et l'unicité d'une solution à ce problème.

## Méthode d'Euler (1/3)

- La **méthode d'Euler** est une procédure numérique :
  - Pour résoudre par approximation des **équations différentielles du premier ordre** avec une condition initiale.
  - C'est **la plus simple des méthodes** de résolution numérique des équations différentielles.
- On considère une ED d'ordre 1 avec condition initiale (problème de Cauchy) :

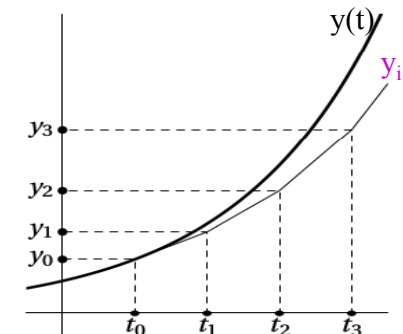
$$\begin{cases} y' = F(t, y) \\ y(t_0) = y_0 \end{cases}$$

## Méthode d'Euler (2/3)

- On cherche à résoudre le problème de Cauchy sur  $I = [a, b]$ 
  - On découpe «  $I$  » en «  $n$  » intervalles de même longueur  
 $h = (b - a)/n$
  - On suppose que  $t_0 = a$  pour faciliter les explications
  - et  $t_i = a + i.h$  pour  $0 \leq i \leq n$
  - On a donc par conséquent:  $t_n = b$ .
- La méthode d'Euler à déterminer approximativement les valeurs prises par «  $y$  » aux instants «  $t_0, t_1, \dots, t_{n-1}$  ».

## Méthode d'Euler (3/3)

- Nous utiliserons les notations suivantes :
  - $y(t_0), y(t_1), \dots, y(t_{n-1})$  : sont les valeurs exactes  
❖ on connaît seulement  $y(t_0)$  ;
  - $Y_0, Y_1, \dots, Y_{n-1}$  : sont les valeurs approchées fournies par la méthode d'Euler.



## Le principe du calcul

❑ On pose  $y_0 = y(t_0)$  et on construit ensuite  $y_{i+1}$  à partir de  $y_i$  pour tout  $i \in [0, n-2]$ .

❑ Plus précisément, on choisit  $y_{i+1}$  tel que:

$$\frac{y_{i+1} - y_i}{t_{i+1} - t_i} = F(y_i, t_i)$$

❑ Avec PYTHON, nous utiliserons des listes:

$t = [t_0, t_1, \dots, t_{n-1}]$  et  $y = [y_0, y_1, \dots, y_{n-1}]$ .

❑ On définira donc  $y[i+1]$  en posant :

$$\forall i \in [0, n-2]:$$

$$y[i+1] = y[i] + (t[i+1] - t[i]) * F(y[i], t[i])$$

## Programmation de la méthode d'Euler

❑ Etant donnés la fonction « F », la valeur «  $y_0$  » et un tableau « t » de taille « n » contenant les valeurs  $t_0, t_1, \dots, t_{n-1}$ ,

➤ on doit créer un tableau « y » de taille « n ».

```
1. #fichier euler.py
2. def euler(F, y0, t):
3. N = len(t)
4. y = [0]*N #crée une liste y de taille N
5. #ne contenant que des 0.
6. y[0] = y0
7. for i in range(N-1):
8. y[i+1] = y[i] + (t[i+1] - t[i]) * F(y[i], t[i])
9. return y
```

## Exemple Python (1/2)

❑ On considère l'équation  $y' = y$  avec la condition initiale  $y(0) = 1$ .

❑ La solution est la fonction exponentielle «  $y = e^t$  », ce qui permet de comparer les résultats obtenus.

❑ On va travailler sur l'intervalle  $[0, 1]$  en considérant tout d'abord un pas de 0.25 puis un pas de 0.1.

❑ La fonction exponentielle «  $e^t$  » (solution exacte) est représentée en pointillés pour comparer.

❑ F doit être définie comme une fonction de deux variables même si ici elle ne dépend que de y.

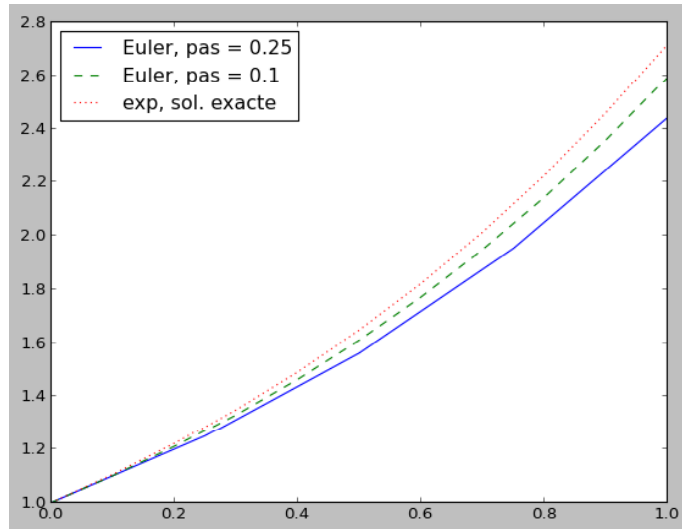
$$F(y, t) = y' = y \text{ et}$$

$$y_0 = 1$$

## Exemple Python (1/2)

```
1. import numpy as np
2. import matplotlib.pyplot as plt
3. def F(y, t): #y'=y => F(y, t)=y d'où la solution y = exp(t)
4. return y
5.
6. t = np.linspace(0, 1, 5)
7. y = euler(F, 1, t)
8. #courbe en ligne bleue contenue
9. plt.plot(t, y, 'b-', label='Euler, pas = 0.25')
10. print(t)
11. t = np.linspace(0, 1, 11)
12. y = euler(F, 1, t)
13. #courbe en ligne verte discontinue
14. plt.plot(t, y, 'g--', label='Euler, pas = 0.1')
15. print(t)
16. #courbe en rouge pointillée
17. plt.plot(t, np.exp(t), 'r:', label='exp, sol. exacte')
18. print(t)
19. plt.legend(loc='upper left')
20. plt.show()
```

## Exemple Python (1/2)



## Exemple Python (1/2)

- ❑ La comparaison graphique de la solution générée par la **méthode d'Euler avec la solution exacte** nous permet de remarquer :
  - qu'on peut **augmenter la précision** de cette méthode d'Euler en **diminuant la taille de « h »**,
  - mais un « h » petit implique une **augmentation de nombre d'intervalles  $[t_i, t_{i+1}]$** ,
  - donc on doit **évaluer la fonction un grand nombre de fois**,
  - ce qui est **coûteux en temps d'exécution** et peut provoquer un **cumul des erreurs numériques** commises par la machine.

## Fonction odeint du package scipy.integrate (1/4)

- ❑ On considère un problème de Cauchy (ED avec condition initiale) de la forme :
 
$$\begin{cases} \frac{dy}{dt} = F(t, y) \\ y(t_0) = y_0 \end{cases}$$
- ❑ On veut calculer des valeurs approchées  $y_0, y_1, \dots, y_{n-1}$  de la **solution « y »** au temps  $t_0, t_1, \dots, t_{n-1}$ .
- ❑ Avec Python on utilise la fonction « **odeint** » du package « **scipy.integrate** ».

```
1. from math import *
2. from scipy.integrate import odeint
3. import numpy as np
4. import matplotlib.pyplot as plt
```

## Fonction odeint du package scipy.integrate (2/4)

- ❑ On appelle « odeint » sous la forme **odeint(F, y0, t)** où :
  - « F » est la fonction qui intervient dans l'équation différentielle,
  - « y0 » est la valeur initiale,
  - « t » est le vecteur contenant les valeurs  $t_0, t_1, \dots, t_{n-1}$ .
- ❑ **Exemple :** On considère l'ED d'ordre 1 avec condition initiale :

$$\begin{cases} \frac{dy}{dt} = -t \cdot y \\ y(0) = 1 \end{cases}$$

- ❑ On veut calculer des valeurs approchées de y entre 0 et 2 pour 5 valeurs régulièrement espacées.
- ❑ Remarque. Le tracé est superposé avec celui de la fonction

$$t \mapsto \exp(-t^2/2)$$

## Fonction odeint du package scipy.integrate (3/4)

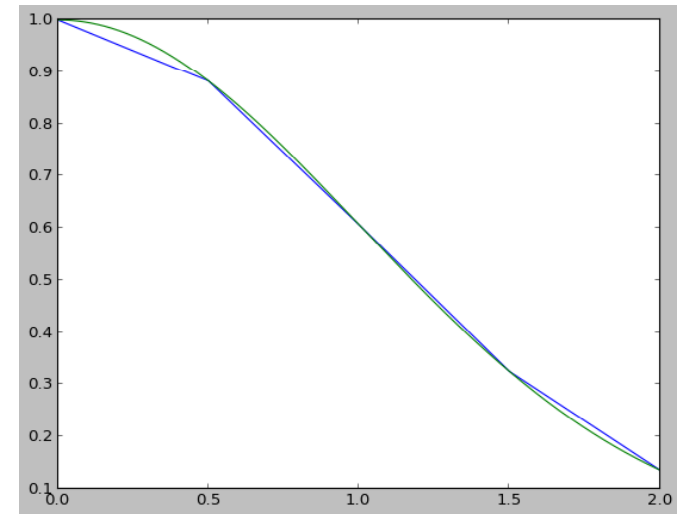
```

1. #ED
2. def F(y,t):
3. return -t*y
4. t = np.linspace(0,2,5)
5. y = odeint(F,1,t)
6. print (t,y)

7. #exp
8. def vexp (x):
9. return exp(-x**2/2)
10. exp_vectorized = np.vectorize(vexp)
11. exp_t = np.linspace(0,2,1000)
12. exp_y = exp_vectorized(exp_t)
13.
14. plt.plot(t,y,exp_t,exp_y)
15. plt.show()

```

## Fonction odeint du package scipy.integrate (4/4)



## Exercices

**Exercice 1:** On veut résoudre sur l'intervalle  $[0, 5]$  le problème de Cauchy :

$$\begin{cases} y' + 2ty = 1 \\ y(0) = 1 \end{cases}$$

**Exercice 2:** On alimente un moteur initialement à l'arrêt par une tension constante  $U = 10V$ .

❑ La vitesse de rotation «  $w$  » du moteur est solution de l'équation différentielle :

❑ Où :

$$J \frac{d\omega}{dt} + \left( \frac{K_e K_c}{R} + f \right) \omega(t) = \frac{K_e}{R} U$$

- $J$  est l'inertie du moteur,  $J = 5.5 \cdot 10^{-5} \text{kg.m}^2$ ;
- $R$  est la résistance électrique du système,  $R = 5,2\Omega$
- $K_c$  et  $K_e$  sont des constantes,  $K_c = 0,24 \text{N.m.A}^{-1}$  et  $K_e = 0,24 \text{V.rad}^{-1} \cdot \text{s}$
- $f$  est un coefficient caractérisant les forces de frottement,  $f = 0.05 \text{N.m.s.r ad}^{-1}$

❑ Déterminer et représenter l'évolution de la vitesse sur l'intervalle de temps  $[0, 10^{-2}]$  (en secondes).

## Partie 4: Méthodes de recherche du zéro d'une fonction (Méthode Newton, Méthode dichotomique, ...)

## Introduction

- ❑ Nous allons voir dans cette partie comment évaluer numériquement une solution d'équation de la forme:

$$f(x) = 0$$

avec « f » une application de IR dans IR,

- ❑ Dans des cadres où la théorie nous assure qu'une telle solution existe, mais sans en fournir d'expression analytique.
- ❑ Deux méthodes vont être présentées :
  - La **méthode de dichotomie** approche une solution avec « p bits » significatifs en « p étapes », ce qui est déjà très efficace.
  - La **méthode de Newton** a une vitesse de convergence assez diabolique.
- ❑ Une bonne compréhension de ces deux méthodes aidera à faire un choix raisonné.

## Méthode dichotomique

- ❑ La méthode de dichotomie approche une solution avec « p bits » significatifs en « p étapes », ce qui est déjà très efficace.
- ❑ Ses conditions d'utilisation sont assez simples :
  - On demande seulement à la fonction d'être continue et de changer de signe.
  - Ceci en fait une **méthode robuste**.
- 1. On divise l'intervalle [a,b] en deux, et on ne garde que la section qui contient la solution.
- 2. Si  $f(a)$  et  $f((a + b)/2)$  sont de signes contraires, on se place alors sur  $[a, (a + b)/2]$  ;
- 3. Sinon, il faudra travailler sur  $[(a + b)/2, b]$
- 4. Puis on recommence en 1.

## Principe théorique (1/3)

- ❑ Lorsque « f » est une fonction continue sur un intervalle [a, b], à valeurs réelles, avec «  $f(a)$  » et «  $f(b)$  » de signe opposé,
  - le théorème des valeurs intermédiaires nous assure que « f » s'annule entre a et b.
- ❑ Il s'agit de construire par récurrence une suite de segments emboîtés,
  - qui va « converger » vers un singleton {},
  - et on démontre alors que  $f(l) = 0$
- ❑ Le point crucial est de préserver la propriété :
  - « f » change de signe entre les deux extrémités du segment.

## Principe théorique (2/3)

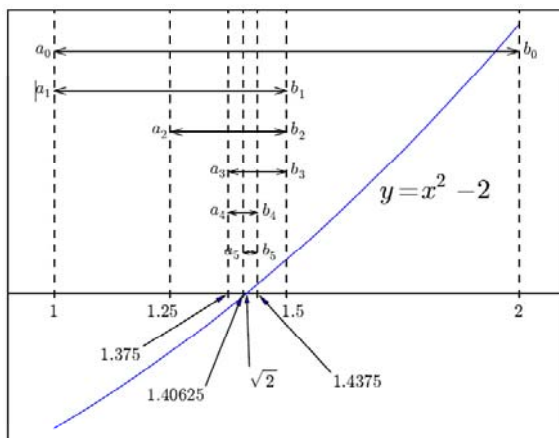
**Exemple:** Le cas de  $f(x) = x^2 - 2$  est simple et démonstratif!

- ❑  $f(1) = -1 < 0 < 2 = f(2)$ , donc l'équation  $f(x) = 0$  possède une solution dans [1, 2].
  - La valeur médiane de cet intervalle est 1.5.
- ❑  $f(1.5) = 0.25 > 0 > f(1)$ , donc l'équation  $f(x) = 0$  possède une solution dans [1, 1.5].
- ❑  $f(1.25) = -0.4675 < 0 < f(1.5)$ , donc l'équation  $f(x) = 0$  possède une solution dans [1.25, 1.5].
- ❑ ...
- ❑  $f(1.41430664062) \approx 0.000263273715973 > 0 > f(1.4140625)$ , donc l'équation  $f(x) = 0$  possède une solution dans [1.4140625, 1.41430664062].

## Principe théorique (3/3)

- On peut représenter ceci, en notant  $[a_n, b_n]$  le segment  $[c, d]$  après la  $n^{\text{ème}}$  étape :

=> On arrête le processus lorsque  $(d - c)$  a atteint une valeur correspondant à la précision demandée



## Algorithme

```

1. Données : f, a, b, ε
2. c ← a d ← b
3. f_de_c ← f(c) f_de_d ← f(d)

4. tant que d - c > 2ε faire
5. m ← (c + d) / 2
6. f_de_m ← f(m)
7. si f_de_c × f_de_m ≤ 0 alors
8. d ← m
9. f_de_d ← f_de_m
10. sinon
11. c ← m
12. f_de_c ← f_de_m
13. Fin_Tant_que

14. Résultat : (c + d) / 2

```

## Terminaison de l'algorithme

- Pour la terminaison, on peut démontrer par récurrence immédiate qu'au début de la  $k$ -ième itération, on a :

$$d - c = \frac{b - a}{2^{k-1}}$$

- Sous les hypothèses  $0 < \epsilon$  et  $a < b$ ,
- on aura  $d - c < 2\epsilon$  pour «  $k$  » suffisamment grand,
  - ce qui fera sortir de la boucle et démontre la terminaison.

## Calcul de la complexité (1/2)

- Ici, il convient de préciser ce qu'on prend en compte :
- Comptabiliser les opérations arithmétiques usuelles de la même façon que les appels de « f »
    - ❖ ceci serait imprudent car chaque appel de « f » peut être non élémentaire!
  - Les opérations arithmétiques élémentaires comme les appels de la fonction sont contrôlés par le nombre de passages dans la boucle.
- La démonstration de la terminaison donne facilement cette complexité. En effet, il y a une  $k$ -ième itération:

➢ si et seulement si :  $\frac{b - a}{2^{k-1}} > 2\epsilon$  (1)

➢ c'est-à-dire :  $k < \log_2 \left( \frac{b - a}{\epsilon} \right)$  (2)

## Calcul de la complexité (2/2)

❑ La boucle sera donc exécutée exactement:

$$N = \left\lceil \log_2 \left( \frac{b-a}{\varepsilon} \right) \right\rceil - 1 \text{ fois} \quad (3)$$

**Remarque :** Supposons donnée une fonction vérifiant les hypothèses requises sur un intervalle  $[a,b]$ . Pour  $\epsilon = 10^{-p}$ , le nombre des itérations sera de l'ordre de :

$$N = \log_2 \left( \frac{b-a}{10^{-p}} \right) = \overbrace{\log_2(b-a)}^{C^{te}} + \overbrace{p \cdot \log_2 10}^{C^{te}}$$

Autrement dit, pour obtenir une précision de l'ordre de  $10^{-p}$ , le nombre d'itération (ou opération) sera de l'ordre de  $p$ , pour  $p$  «grand»!

## Convergence

❑ L'erreur absolue de la méthode de dichotomie est au plus:

➤  $(b-a)/2^{n+1}$  après «  $n$  » étapes l'écart de l'erreur est diminué de moitié à chaque étape.

❑ Ainsi, la méthode converge linéairement, ce qui est très lent par comparaison avec la méthode de Newton.

❑ Le principal avantage pratique de cette méthode est sa robustesse, puisque si «  $f$  » est continue, alors l'algorithme est théoriquement convergent ,

➤ la taille de l'intervalle de recherche tend vers zéro.

## Programme en Python

```
1. """ Recherche approchée d'une solution d'une
 équation par dichotomie """
2. def dichotomie(f,a,b,epsilon):
3. assert f(a) * f(b) <= 0 and epsilon > 0
4. c, d = a, b
5. fc, fd = f(c), f(d)
6. while d - c > 2 * epsilon:
7. m = (c + d) / 2.
8. fm = f(m)
9. if fc * fm <= 0:
10. d, fd = m, fm
11. else:
12. c, fc = m, fm
13. return (c + d) / 2.
```

## Exemples

❑ Reprenons l'exemple 1. On pourrait définir une fonction:

➤ `def g(x): return x**2-2`, mais c'est inutile

➤ Python offre la possibilité de définir des fonctions anonymes via l'opérateur **lambda** :

```
1. from math import sqrt
2. sqrt(2), dichotomie(lambda x : x**2-2, 1, 2,
 0.000001)
3. (1.4142135623730951, 1.4142141342163086)
```

❑ Exemple 2: On utilise la fonction **sin** de la bibliothèque **math** :

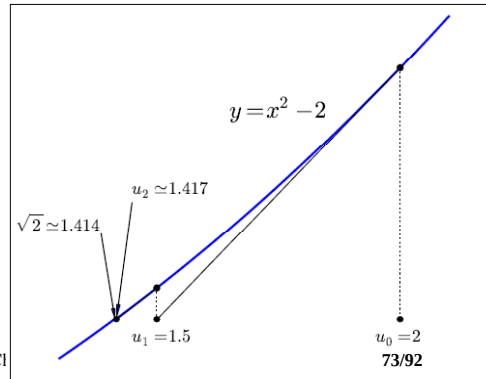
```
1. import math
2. math.pi, dichotomie(math.sin, 3, 4, 10**-10)
3. (3.141592653589793, 3.141592653642874)
```

## Méthode de Newton

- On reprend l'exemple  $f(x) = x^2 - 2$  en partant d'une valeur initiale  $u_0 = 2$  et on suit la tangente du graphe de « f » depuis le point d'abscisse 2.
- L'intersection de cette tangente avec l'axe des abscisses donne une nouvelle valeur  $u_1 = 1.5$ ,
  - qui aurait été un zéro de « f » si cette dernière avait été affine.

On reprend la tangente depuis le point d'abscisse  $u_1$ , pour trouver une nouvelle valeur  $u_2 \approx 1.4167$ .

Les deux premières itérations du calcul approché de  $\sqrt{2}$  par la méthode de Newton. Difficile de visualiser au delà !



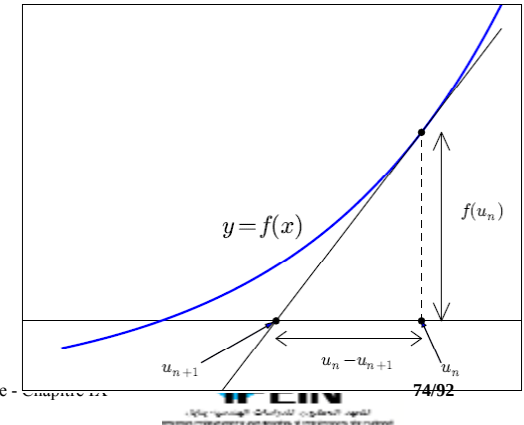
IPEIN 2017/2018

Informatique - CI

## Newton - Algorithme général (1/2)

- On part d'une première valeur et on suit la tangente ;
- On continue avec l'intersection de cette tangente avec l'axe des abscisses;
- Jusqu'à ce que la différence entre deux termes consécutifs soit assez petite.

Il est nécessaire (on croise les doigts) de ne jamais rencontrer de point en lequel la dérivée de « f » s'annule!



IPEIN 2017/2018

Informatique - Chapitre IX

## Newton - Algorithme général (2/2)

- Dans la méthode de Newton:

$$f'(u_n) = \frac{f(u_n)}{u_n - u_{n+1}}$$

Donc: 
$$u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}$$

### Algorithme:

- Données : f, g, u0, ε (g représente f')
- u ← u0
- v ← u - f(u)/g(u)
- tant que |v - u| > ε faire
- u ← v
- v ← v - f(v)/g(v)
- Fin\_Tant\_que
- Résultat : v

IPEIN 2017/2018

Informatique - Chapitre IX

75/92

## Newton – Programme en Python

- Pour ce programme, on ne fait aucune vérification d'hypothèse de convergence!

La terminaison et la correction ne sont donc pas assurées!

```
1. def newton(f,fp,x0,epsilon):
2. u = x0
3. v = u - f(u)/fp(u)
4. while abs(v-u) > epsilon:
5. u, v = v, v - f(v)/fp(v)
6. return v
```

- Dans des conditions réelles, on vérifierait à la volée que:

les valeurs prises par u et v ne sont pas trop grandes

et/ou que celles prises par f(v) ne sont pas trop petites.

```
1. >>> newton(math.sin, math.cos, 3, 10**(-3))
2. 3.141592653589793
3. >>> newton(lambda x:x**2-2,lambda x:2*x, 2., 10**(-2))
4. 1.4142156862745099
```

IPEIN 2017/2018

Informatique - Chapitre IX

76/92

## Quelle méthode choisir ?

- ❑ On ne peut pas écrire une **méthode universelle** de résolution numérique d'équation de la forme  $f(x) = 0$  (**mais on peut chercher le meilleur compromis**).
  - Si on cherche **la simplicité et la robustesse**, la méthode dichotomique s'impose :
    - ❖ la continuité de la fonction et un premier intervalle  $[a, b]$  tel que  $f(a).f(b) \leq 0$  **sont suffisants**.
    - ❖ La **lenteur de cette méthode** est toute relative.
  - Si on cherche **la rapidité**, alors on peut appliquer une méthode de la famille de Newton.
  - Dans des situations intermédiaires, **on peut pratiquer des méthodes mixtes**,
    - ❖ avec une première phase de localisation de zéros par dichotomie, puis des itérations de Newton.

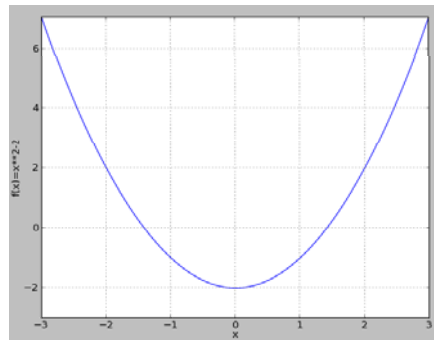
## Déjà disponible en Python (1/3)

- ❑ La plupart du temps, il est déraisonnable de coder soi-même une méthode de résolution qui existe déjà et a probablement été **plus testée et optimisée qu'une fonction « maison »**!
- ❑ Les méthodes de résolution approchée d'équations se trouvent dans le module `scipy.optimize`,
  - qui est lui-même un sous-module du (gros) module d'analyse numérique `scipy`.
- ❑ Parmi les fonctions utiles du module `scipy.optimize` on a :
  - `bisect(f, a, b)` : détermine une racine de « f » dans  $[a, b]$  en effectuant **une dichotomie**.
  - `newton(f, x0)` : détermine une racine par la **méthode de Newton** ou approchée en partant de « x0 »

## Déjà disponible en Python (2/3)

```
1. import scipy.optimize as op
2. import numpy as np
3. import matplotlib.pyplot as plt
4. f = lambda x: x**2-2
5. x = np.linspace(-3,3,100)
6. y = f(x)
7. plt.xlabel('x')
8. plt.grid(True)
9. plt.ylabel('f(x)=x**2-2')
10. plt.axis([-3, 3, -3, 7])
11. plt.plot(x,y)
12. plt.show()

13. op.bisect(f,1,2)
14. 1.4142135623724243
15. g = lambda x: x**2-9
16. op.newton(g,2)
17. 2.9999999999999876
```



## Déjà disponible en Python (3/3)

- ❑ La fonction `numpy.roots` détermine les racines d'un polynôme donné par la liste de ses coefficients.

**Exemple:**  $f(x) = (x-1)(x+2) = x^2 + x - 2$

D'où le polynôme  $P = [1, 1, -2]$

```
1. import numpy as np
2. np.roots([1,1,-2])
3. array([-2., 1.])
4. # Elle fournit même les racines complexes,
5. # le nombre imaginaire i étant noté j.
6. np.roots([1,0,0,1])
7. array([-1.0+0.j, 0.5+0.8660254j, 0.5-0.8660254j])
```

- ❑ Enfin, pour des problèmes qui ne sont pas scalaires, on peut utiliser la fonction `fsolve` :

```
1. scipy.optimize.fsolve(lambda(x,y):(x+y**2, 1-y+x**2),
(0,0))
2. array([-0.33145847, 0.75398698])
```

## Partie 5: Méthodes d'intégrations (Méthode des rectangles, Méthode des trapèzes,... - scipy).

### Principe général (1/2)

- ❑ Le principe général commun aux deux méthodes est simple :
  - découper l'intervalle d'intégration en petits morceaux,
  - et approcher sur chacun de ces petits intervalles la courbe représentative de la fonction « f » par une courbe très simple pour laquelle le calcul d'aire est facile.
- ❑ Étant donnés deux réels « a » et « b » tels que  $a < b$  et une fonction « f » continue sur le segment  $[a, b]$ ,
  - on souhaite, à l'aide de suites, obtenir une valeur approchée de l'intégrale de « f » sur  $[a, b]$ .

❑ On note:  $I(f) = \int_a^b f(t)dt$  l'intégrale à approcher.

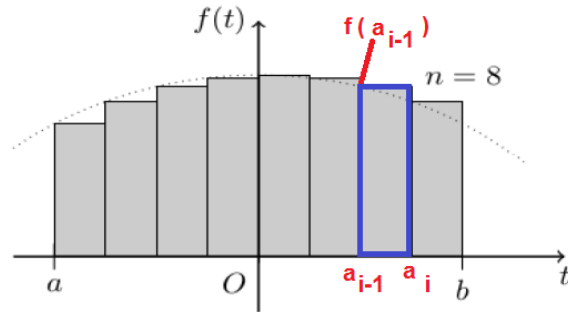
### Principe général (2/2)

- ❑ Les valeurs de « f » peuvent provenir de la connaissance de la fonction,
  - ou alors de la connaissance de valeurs de la fonction pour certaines valeurs de « t » (Exp. des mesures régulières).
- ❑  $\forall n \in \mathbb{N}^*$ , on va partager le segment  $[a, b]$  à l'aide d'une subdivision  $(a_i)_{0 \leq i \leq n}$  à pas constant égal à «  $(b-a)/n$  » en posant:
 
$$\forall i \in [0, n], a_i = a + i \frac{b-a}{n}$$
  - Sur chaque segment de  $[a_{i-1}, a_i]$ , on remplace « f » par une fonction dont on sait calculer l'intégrale.
- ❑ Nous présentons deux méthodes :
  - La méthode des rectangles où les fonctions utilisées pour approcher « f » sont des constantes.
  - La méthode des trapèzes où les fonctions utilisées pour approcher « f » sont des affines.

### La méthode des rectangles (1/3)

- ❑ Quoi de plus simple comme fonction qu'une fonction constante ?
  - Et quoi de plus simple comme aire à calculer qu'une aire de rectangle ?
- ❑ Cette méthode consiste donc à approcher notre fonction sur chaque sous-intervalle par la fonction constante prenant la même valeur que « f » à gauche de l'intervalle.
  - D'où, sur chaque segment de  $[a_{i-1}, a_i]$ , on remplace « f » par la fonction constante de valeur «  $f(a_{i-1})$  ».
  - L'intégrale de « f » est alors approchée par la somme «  $R_n(f)$  » des aires algébriques des rectangles de sommets  $(a_{i-1}, 0), (a_i, 0), (a_i, f(a_{i-1}))$  et  $(a_{i-1}, f(a_{i-1}))$ .

## La méthode des rectangles (2/3)



- Chaque rectangle a un côté de mesure (positive)  $a_i - a_{i-1} = (b-a)/n$ 
  - et un autre côté de mesure algébrique  $f(a_{i-1})$  de telle sorte que:

$$\forall n \in \mathbb{N}^*, R_n(f) = \frac{b-a}{n} \sum_{i=0}^{n-1} f(x_i)$$

## La méthode des rectangles (3/3)

### □ Algorithme :

```

1. def rectangle (f,a,b,n):
2. """ valeur approchée de l'intégrale de f sur
3. [a,b] avec la méthode des rectangles """
4. int_rect = 0
5. h = (b-a)/n
6. t = a
7. for k in range(n):
8. int_rect += h*f(t)
9. t += h
10. return int_rect

```

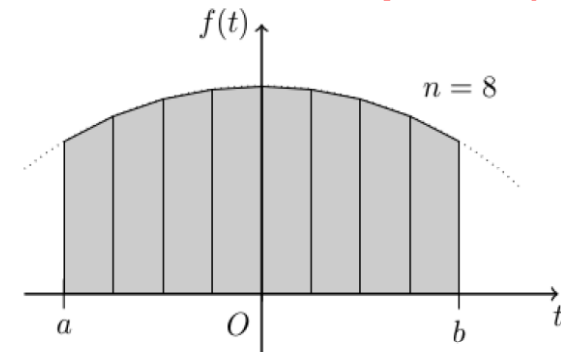
### □ Exercice :

Estimer la complexité de l'algorithme précédent.

## La méthode des trapèzes (1/3)

- Le principe général est exactement le même que celui de la méthode des rectangles,
  - mais on approche cette fois-ci la courbe sur le segment  $[a_i, a_{i-1}]$  par le segment de droite reliant les deux points de la courbe d'abscisses «  $a_i$  » et «  $a_{i-1}$  »,
  - Ce qui revient à calculer une somme d'aires de trapèzes pour approcher l'intégrale.
- Sur chaque segment de  $[a_{i-1}, a_i]$ , on remplace «  $f$  » par la fonction affine qui coïncide avec «  $f$  » en «  $a_{i-1}$  » et en «  $a_i$  ».
  - L'intégrale de «  $f$  » est alors approchée par la somme «  $T_n(f)$  » des aires algébriques des trapèzes de sommets  $(a_{i-1}, 0), (a_i, 0), (a_i, f(a_i))$  et  $(a_{i-1}, f(a_{i-1}))$ .
  - Visuellement, l'impression est nettement meilleure que pour la méthode des rectangles.

## La méthode des trapèzes (2/3)



- Chaque trapèze a une aire algébrique qui vaut:

$$\frac{b-a}{n} * \frac{f(a_i) + f(a_{i-1})}{2} \text{ de telle sorte que :}$$

$$\forall n \in \mathbb{N}^*, T_n(f) = \frac{b-a}{2n} \sum_{i=1}^n f(a_i) + f(a_{i-1})$$

## La méthode des trapèzes (3/3)

### ❑ Algorithme:

```
1. def trapezes(f,a,b,n):
2. h = (b-a)/n
3. int_trapeze = 0.
4. t = a
5. for k in range(n):
6. int_trapeze += h * (f(t) + f(t+h)) / 2
7. t += h
8. return int_trapeze
```

### ❑ Exercice :

Estimer la complexité de l'algorithme précédent.

## Déjà disponible en Python

❑ La fonction `scipy.integrate.trapz` permet de mettre en œuvre la méthode des trapèzes sans la programmer.

❑ Si on peut/doit utiliser une fonction de bibliothèque,

➤ on privilégiera `scipy.integrate.quad`,

➤ plus rapide et performante.

```
1. >>> from scipy import integrate
2. >>> y = lambda x: x**2-2
3. >>> integrate.quad(y, 1, 2)
4. (0.3333333333333333, 8.555891116643817e-15)
5. >>> import numpy as np
6. >>> invexp = lambda x: np.exp(-x)
7. >>> integrate.quad(invexp, 0, np.inf)
8. (1.0000000000000002, 5.842606742906004e-11)
```

## Références

- ❑ [1] Tentative numpy tutorial, document web, 2013.  
[http://www.scipy.org/Tentative\\_NumPy\\_Tutorial](http://www.scipy.org/Tentative_NumPy_Tutorial).
- ❑ [2] Valentin Haenel, Emmanuelle Gouillart, and G  l Varoquaux editors. Python scientific lecture notes, release 2013.1.  
<http://scipy-lectures.github.com>, 2013.
- ❑ [3] NumPy community. Numpy user guide, release 1.8.0.dev-fd6f038, 2013. <http://docs.scipy.org/doc/numpy-dev/numpy-user.pdf>.
- ❑ [4] <http://www.coursofpython.com/>
- ❑ [5] <http://matplotlib.org/>
- ❑ [6] <http://www.numpy.org/>

## Annexe 1 : Data types

- ❑ `bool_` Boolean (True or False) stored as a byte
- ❑ `int_` Default integer type (same as C long; normally either int64 or int32)
- ❑ `intc` Identical to C int (normally int32 or int64)
- ❑ `intp` Integer used for indexing (same as C ssize\_t; normally either int32 or int64)
- ❑ `int8` Byte (-128 to 127)
- ❑ `int16` Integer (-32768 to 32767)
- ❑ `int32` Integer (-2147483648 to 2147483647)
- ❑ `int64` Integer (-9223372036854775808 to 9223372036854775807)
- ❑ `uint8` Unsigned integer (0 to 255)
- ❑ `uint16` Unsigned integer (0 to 65535)
- ❑ `uint32` Unsigned integer (0 to 4294967295)
- ❑ `uint64` Unsigned integer (0 to 18446744073709551615)
- ❑ `float_` Shorthand for float64.
- ❑ `float16` Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
- ❑ `float32` Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
- ❑ `float64` Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
- ❑ `complex_` Shorthand for complex128.
- ❑ `complex64` Complex number, represented by two 32-bit floats (real and imaginary components)
- ❑ `complex128` Complex number, represented by two 64-bit floats (real and imaginary components)