Université de Strasbourg
Environnements Informatique Feuille de TP

Python - introduction à la programmation et calcul scientifique

1 Avant de commencer

Le but de ce TP est de vous montrer les bases de la programmation avec Python et ces applications au calcul scientifique.

- **Lises attentivement l'introduction et les explications des exercices.**
- **Effectuez toutes les manipulations proposées - même si elle ne sont pas marquées explicitement comme exercices ils en sont !**

Ce TP donne juste une aperçu de quelques-uns des possibilités de la programmation avec Python et de ses applications scientifiques. Python est un langage très puissant est varié - en quelques heures c'est impossible de faire même une vague introduction de toutes ses possibilités. Pour cette raison, nous avons décidé d'inclure juste quelques exemples qui vous permettront d'avoir un aperçu de ce que vous pouvez gagner en approfondissant vos connaissances par vous-mêmes.

Le contenu de ce TP a été basé sur les document suivants :

- Introduction en Python (tutoriel) `http://python.developpez.com/cours/TutoVanRossum/`.
- Apprendre à programmer avec Python. Cours détaillé sur Python. `http://www.framasoft.net/IMG/pdf/python_notes-2.pdf`.

2 Généralités

2.1 Qu'est-ce que c'est un programme (rappel)

A strictement parler, un ordinateur n'est rien d'autre qu'une machine effectuant des opérations simples sur des séquences de signaux électriques, lesquels sont conditionnés de manière à ne pouvoir prendre que deux états seulement (par exemple un potentiel électrique maximum ou minimum). Ces séquences de signaux obéissent à une logique du type « tout ou rien » et peuvent donc être considérés conventionnellement comme des suites de nombres ne prenant jamais que les deux valeurs 0 et 1. Un système numérique ainsi limité à deux chiffres est appelé système binaire.

Sachez dès à présent que dans son fonctionnement interne, un ordinateur est totalement incapable de traiter autre chose que des nombres binaires. Toute information d'un autre type doit être convertie, ou codée, en format binaire. Cela est vrai non seulement pour les données que l'on souhaite traiter (les textes, les images, les sons, les nombres, etc.), mais aussi pour les programmes, c'est-à-dire les séquences d'instructions que l'on va fournir à la machine pour lui dire ce qu'elle doit faire avec ces données.

Le seul « langage » que l'ordinateur puisse véritablement « comprendre » est donc très éloigné de ce que nous utilisons nous-mêmes. C'est une longue suite de 1 et de 0 (les "bits") souvent traités par groupes de 8 (les « octets »), 16, 32, ou même 64. Ce « langage machine » est évidemment presqu'incompréhensible pour nous. Pour « parler » à un ordinateur, il nous faudra utiliser des systèmes de traduction automatiques, capables de convertir en nombres binaires des suites de caractères formant des mots-clés (anglais en général) qui seront plus significatifs pour nous.

Le système de traduction proprement dit s'appellera interpréteur ou bien compilateur, suivant la méthode utilisée pour effectuer la traduction. Ces systèmes de traduction automatique seront établis sur la base de toute une série de conventions, dont il existera évidemment de nombreuses variantes. On appellera langage de programmation un ensemble de mots-clés (choisis arbitrairement) associé à un ensemble de règles très précises indiquant comment on peut assembler ces mots pour former des « phrases » que l'interpréteur ou le compilateur puisse traduire en langage machine (binaire).

2.2 Erreurs de syntaxe

Python ne peut exécuter un programme que si sa syntaxe est parfaitement correcte. Dans le cas contraire, le processus s'arrête et vous obtenez un message d'erreur. Le terme syntaxe se réfère aux règles que les auteurs du langage ont établies pour la structure du programme.

Tout langage comporte sa syntaxe. Dans la langue française, par exemple, une phrase doit toujours commencer par une majuscule et se terminer par un point. ainsi cette phrase comporte deux erreurs de syntaxe

Dans les textes ordinaires, la présence de quelques petites fautes de syntaxe par-ci par-là n'a généralement pas d'importance. Il peut même arriver (en poésie, par exemple), que des fautes de syntaxe soient commises volontairement. Cela n'empêche pas que l'on puisse comprendre le texte.

Dans un programme d'ordinateur, par contre, la moindre erreur de syntaxe produit invariablement un arrêt de fonctionnement (un « plantage ») ainsi que l'affichage d'un message d'erreur. Au cours des premières semaines de votre carrière de programmeur, vous passerez certainement pas mal de temps à rechercher vos erreurs de syntaxe. Avec de l'expérience, vous en commettrez beaucoup moins.

Gardez à l'esprit que les mots et les symboles utilisés n'ont aucune signification en eux-mêmes : ce ne sont que des suites de codes destinés à être convertis automatiquement en nombres binaires. Par conséquent, il vous faudra être très attentifs à respecter scrupuleusement la syntaxe du langage.

3 Introduction en Python

Python présente la particularité de pouvoir être utilisé de plusieurs manières différentes. Vous allez d'abord l'utiliser en mode interactif, c'est-à-dire d'une manière telle que vous pourrez dialoguer avec lui directement depuis le clavier. Cela vous permettra de découvrir très vite un grand nombre de fonctionnalités du langage. Dans un second temps, vous apprendrez comment créer vos premiers programmes (scripts) et les sauvegarder sur disque.

L'interpréteur peut être lancé directement depuis la ligne de commande (dans un « shell » Linux, ou bien dans une fenêtre DOS sous Windows) : il suffit d'y taper la commande "python" (en supposant que le logiciel lui-même ait été correctement installé).

Les trois caractères « supérieur à » constituent le signal d'invite, ou prompt principal, lequel vous indique que Python est prêt à exécuter une commande.

```
$ python
Python 2.6.2 (r262:71600, Aug  4 2009, 14:23:09)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Vous pouvez alors taper des commandes directement.

Par exemple, vous pouvez tout de suite utiliser l'interpréteur comme une simple calculatrice de bureau. Veuillez donc vous-même tester les commandes ci-dessous :

```
>>> 1+1    # le symbole "#" sert à créer des commentaires

>>> 1.5-2 # le texte écrit après ce symbole est ignoré par Python

>>> 2 - 9 # les espaces sont optionnels

>>> 7 + 3 * 4 # la hiérarchie des opérations mathématiques
# est-elle respectée ?
>>> (7+3)*4

>>> 1 / 2 # surprise !!!
```

4 Utilisation du Python comme calculatrice

4.1 Les nombres et les opérations mathématiques de base

Voici les opérations mathématiques de base :

- Addition, soustraction :
  `3+5, 1-3`
- Multiplication, division :
  `12*1j, 2./3`
- Modulo :
  `5%3, 3.5%2.8`
- Puissance :
  `10**2, (2.+3.j)**(4-3.5j)`

Et voici quelques exemples :

```
>>> 2+2
4
>>> (50-5*6)/4
5
>>> 7/3       # Les divisions entières retournent des entiers
2
>>> 7/-3
-3
```

Pour les divisions non entières, il faut manipuler des réels :

```
>>> 7/3
2
>>> 7.0 / 3
2.3333333333333335
>>> 7 / 3.0
2.3333333333333335
```

Les variables[1] sont affectées avec le signe '='. Ainsi la commande

```
>>> x=42
```

affecte la valeur 42 à la variable x. On peut alors la manipuler comme le montre l'exemple suivant :

```
>>> x       # afficher la valeur associé à x
42
>>> x*2
84
>>> x**2 # x au carré
1764
```

Les nombres complexes sont représentés avec l'aide d'unité imaginaire *j*. Par exemple :

```
>>> 3j
3j
>>> 1+2j
(1+2j)
>>> y = 1+4j
>>> y
(1+4j)
>>> y ** 2
(-15+8j)
>>> x * y
(42+168j)
```

4.2 Opérations mathématiques supplémentaires

Pour effectuer des opérations mathématiques plus complexes nous devons utiliser quelques fonctions supplémentaires. Pour accomplir cela, nous devons "importer" la bibliothèque avec les fonctions scientifiques appelée "NumPy".

```
>>> import numpy as np
```

Dorénavant, pour accéder aux fonctions définies par NumPy nous devons ajouter le préfixe **np**, comme par exemple :

```
>>> np.sin(3)
0.14112000805986721
```

[1] Pour pouvoir accéder aux données, le programme d'ordinateur (quel que soit le langage dans lequel il est écrit) fait abondamment usage d'un grand nombre de **variables** de différents types.

Une **variable** apparaît dans un langage de programmation sous un nom de variable à peu près quelconque auquel correspond une **valeur**. La **valeur** peut être en fait à peu près n'importe quel « objet » susceptible d'être placé dans la mémoire d'un ordinateur, par exemple : un nombre entier, un nombre réel, un nombre complexe, un vecteur, une chaîne de caractères typographiques, un tableau, une fonction, etc. Pour distinguer les uns des autres ces divers contenus possibles, le langage de programmation fait usage de différents types de variables. (le type 'entier', le type 'réel', le type 'chaîne de caractères', le type 'liste', etc.).

Voici quelques fonctions mathématiques présentes dans NumPy :

- pi - la constante π.
```
>>> np.pi
```

- rad2deg(), deg2rad() - conversion de radians en dégrées et vice versa. Notez bien, que partout en Python les angles sont donnés en radians.
```
>>> np.deg2rad(180) # Où autrement dit - pi

>>> np.rad2deg(np.pi)
```

- abs() - la valeur absolue.
```
>>> np.abs(23)

>>> np.abs(-23)

>>> np.abs(1+1j)
```

- angle() - l'angle d'un numéro complexe.
```
>>> np.angle(23)

>>> np.angle(-3)

>>> np.angle(1+1j)
```

- cos(), sin(), tan() - les fonctions trigonométriques de base.
```
>>> np.cos(np.pi)

>>> np.sin( np.deg2rad(45) )

>>> np.tan( np.pi/4 )
```

- arccos(), arcsin(), arctan() - les fonctions inverses de cos, sin et tan.
```
>>> x= np.cos(np.pi)
>>> np.arccos(x)
```

- ceil(), floor(), round() - arrondir une valeur.
```
>>> np.ceil(3.01)   #  arrondi au plus petit entier supérieur

>>> np.floor(3.99)   #  arrondi au plus grand entier inférieur

>>> np.round(3.49)   # arrondi au plus proche

>>> np.round(3.50)   # arrondi au plus proche
```

- conj() - la valeur conjugué d'un nombre complexe.
```
>>> np.conj(5)
```

```
>>> np.conj(5+1j)
```

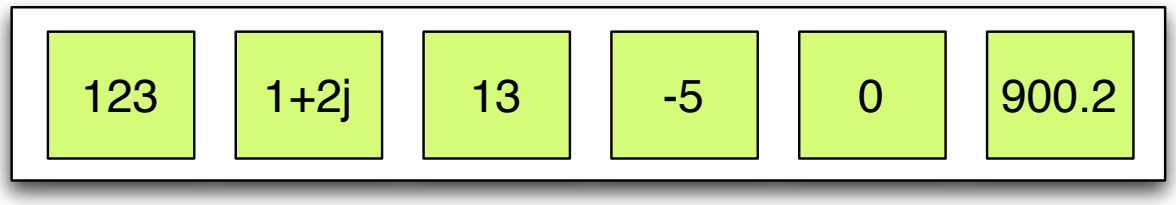
Ceux sont juste quelques exemples des fonctions définies par NumPy. Pour voir la liste complète de fonctions, utilisez la fonction **dir** :

```
>>> dir() # Affiche la liste avec toutes les variables actuellement définies

>>> dir(np) # Affiche la liste avec toutes les fonctions définies par np
```

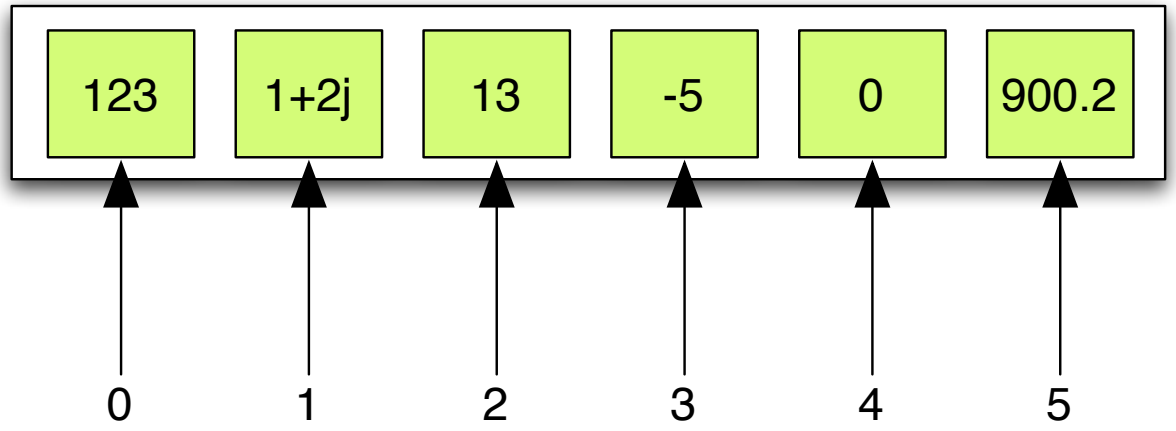
5 Tableaux, vecteurs et matrices

Toutes les opérations que vous venez d'essayer peuvent être appliquées à plusieurs nombres à la fois. Ceci est possible avec l'utilisation des **tableaux**. Un tableau (array en anglais) est une structure de données qui est un ensemble de variables auquel on a accès à travers un numéro d'indice.

Voici un exemple d'un tableau contenant 6 éléments - 123, 1+2j, 13, -5, 0 et 900.2 :



Chaque case du tableau est numérote, en **commencent par 0** :



Un tableau peut être également utilisé pour la représentation des vecteurs, matrices et tenseurs. En Python ils existent deux façons de déclarer des tableaux - comme des listes ou des tableaux mathématiques. Dans cette section nous allons voir les deux approches.

5.1 Les listes en Python

Les listes sont le moyen intégré en Python pour la définition de tableaux. Une liste peut être écrite comme une liste de valeurs entre crochets et séparés par des virgules :

```
>>> x = [123, 1+2j, 13, -5, 0, 900.2]

>>> x
[123, (1+2j), 13, -5, 0, 900.20000000000005]
```

Pour obtenir la taille d'un tableau il suffit d'utiliser la fonction **len** :

```
>>> len(x)
6
```

Puis, pour lire le contenu d'un case, on doit utiliser son indice :

```
>>> x[0]
123
>>> x[1]
(1+2j)
```

Attention - les tableaux (comme dans la plupart de langages de programmation) commencent leurs indices par 0. Question - quelle est l'indice du dernier élément du x dans l'exemple donnée ?

Qu'est-ce qui ce passe quand vous essayez de lire le contenu d'une case qui n'existe pas ?

```
>>> x[100]

>>> x[10]

>>> x[6]

>>> x[5]
```

La modification d'une case est également basé sur son indice :

```
>>> x
[123, (1+2j), 13, -5, 0, 900.20000000000005]
>>> x[0] = -2
>>> x
[-2, (1+2j), 13, -5, 0, 900.20000000000005]
>>> x[5] = 3
>>> x
[-2, (1+2j), 13, -5, 0, 3]
>>> x[1] = x[2]
```

```
>>> x
[-2, 13, 13, -5, 0, 3]
```

On peut également rajouter des éléments avec la fonction **append** :

```
>>> x
[-2, 13, 13, -5, 0, 3]
>>> x.append(1)
>>> x
[-2, 13, 13, -5, 0, 3, 1]
>>> x.append(100)
>>> x
[-2, 13, 13, -5, 0, 3, 1, 100]
```

Où en supprimer avec **remove** :

```
>>> x.remove(0)
>>> x
[-2, 13, 13, -5, 3, 1, 100]
>>> x.remove(-5)
>>> x
[-2, 13, 13, 3, 1, 100]
```

Qu'est-ce qui ce passe si on essaie de supprimer une valeur qui n'est pas présente dans la liste ?

Une liste peut mélanger plusieurs quel type de données :

```
>>> y = ["un", "deux", 1, 2, 3]
>>> y
```

Et nous pouvons facilement concaténer deux listes :

```
>>> x

>>> y

>>> z = x + y
>>> z

>>> x * 2

>>> x * 3

>>> x * 4
```

5.2 Les tableaux NumPy

Les listes sont très utilisées et indispensables pour la maîtrise de Python. Cependant, elles ont quelques limitations qui les rendent plus difficiles à utiliser dans le contexte mathématique. Par exemple, opération * (multiplication) ne correspond pas à la multiplication d'un vecteur avec un nombre. La bibliothèque NumPy a été crée pour résoudre ce problème. En plus d'un grand nombre de fonctions, elle défini un autre type de tableaux, spécialement conçus pour les opérations mathématiques, tels que vecteurs, matrices et tenseurs.

Ils existent quelques moyens pour la définition d'un tableau NumPy :

```
>>> np.zeros(10) # Créer un vecteur contenant 10 fois 0
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

>>> np.ones(10) # Créer un vecteur contenant 10 fois 1
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])

>>> np.arange(10) # Créer un vecteur [0, 10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> np.array([1,2,3,4,5]) # Créer un vecteur à partir de la liste [1,2,3,4,5]
array([1, 2, 3, 4, 5])
```

A partir du moment où vous avez un tableau NumPy, vous pouvez appliquer toutes les opérations mathématiques que nous avons déjà rencontré :

```
>>> x = np.array([1, 2, -3, 0.4, 10])
>>> x
array([  1. ,   2. ,  -3. ,   0.4,  10. ])
>>> x * 2

>>> x * 3

>>> x * 4

>>> [1,2,-3,0.4,10]*4 # Juste pour comparer avec le comportement des listes !

>>> x + 1

>>> x -2+3j

>>> np.sin(x)

>>> np.ceil(x)

>>> y = np.ones(5)
>>> y
```

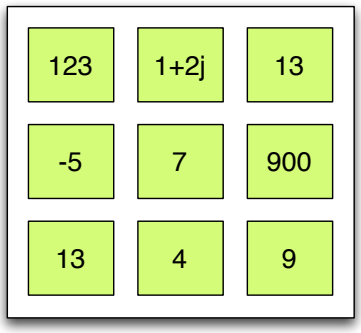
```
>>> x + y

>>> 2*x + 3*y
```

Mais il existent encore beaucoup d'autres fonctions que vous pouvez utiliser une fois que vous avez un tableau NumPy :

```
>>> x = np.array([10, 2, 2, 3, 4, 123, 3])
>>> x

>>> x.sort() # Trier les éléments
>>> x

>>> x.sum() # Trouver la somme des éléments

>>> x.var() # Trouver la variance des éléments

>>> x.std() # Trouver l'écart-type des éléments

>>> x.max() # Trouver la valeur maximale

>>> x.min() # Trouver la valeur minimale

>>> dir(x) # Afficher toutes les foncions de x

>>> help(x.mean) # Afficher l'aide pour la fonction mean de x
```

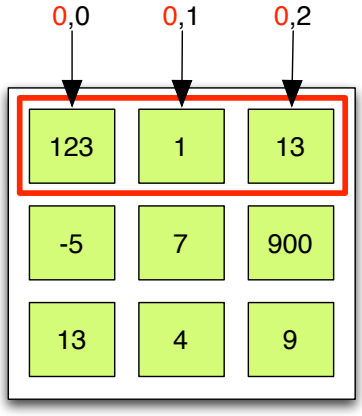
5.3 Les tableaux NumPy multidimensionnelles

Dans la section précédente nous avons vus comment traiter des vecteurs avec les tableaux NumPy. Pour la définition de matrices ou tenseurs nous avons besoin de tableaux comme celui-ci :

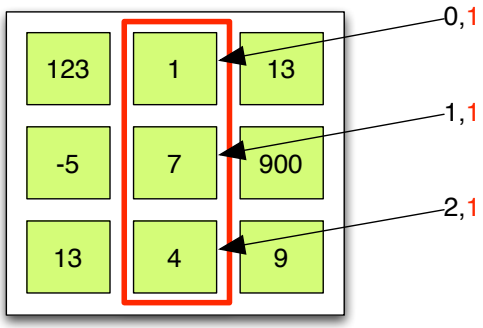


Ce tableau contient 3 lignes et 3 colonnes, donc c'est un tableau en 2 dimensions. On peut également avoir des tableaux de 3 ou plus dimensions. Maintenant, pour accéder à un élément il faut spécifier deux indices - le numéro de sa ligne et colonne. Le premier indice donne le numéro de la ligne, et le deuxième

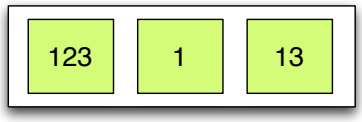
- la colonne. Par exemple, tous les éléments de la ligne 0 peuvent être accèdes en utilisant les indices suivants :



Et pour accéder aux éléments de colonne 1 il faut donner les indices de la manière suivante :



Enfin, pour définir un tableau à deux dimensions nous pouvons spécifier les éléments ligne par ligne. Chaque ligne est définie comme une liste Python, par exemple :



Puis, il faut créer une nouvelle liste Python contenant les lignes qui ont été définis :

Et voici l'exemple concret :

```
>>> x = [123, 1, 13]
>>> y = [-5, 7, 900]
```

```
>>> z = [13, 4, 9]

>>> liste2d = [ x, y, z ]
>>> liste2d

>>> matrix = np.array(liste2d)
>>> matrix

>>> matrix[0,0]

>>> matrix[0,1]

>>> matrix[0,2]

>>> matrix[1,1]
```

Très souvent, on fait la déclaration d'un tableau NumPy en une seule ligne :

```
>>> A = np.array([ [123,1,13], [-5,7,900], [13,4,9] ])
>>> A
```

Il est facile de créer une matrice ayant une diagonale précis :

```
>>> np.diag([1,2,3,4,5]) # Une matrice ayant [1,2,3,4,5] pour diagonale principa

>>> np.identity(10) # Matrice d'identité de taille 10x10.
```

Maintenant vous pouvez traiter le tableau en 2D comme une matrice ! Et NumPy propose plusieurs fonctions liées aux matrices :

```
>>> A.shape # Les dimensions de la matrice
(3, 3)

>>> np.transpose(A) # La matrice transposée de A

>>> np.trace(A) # La trace de la matrice A

>>> np.diag(A) # La diagonale principale de A

>>> B = np.linalg.inv(A) # La matrice inversé de A
>>> B

>>> np.dot(A, B) # Produit des matrices A et B.
# Notez bien que dans le cas précis B est l'inverse de A.
# Alors A * B doit donner une matrice ayant 1 sur le diagonal
# principal, et 0 partout ailleurs. Observez que certaines valeurs
# dans le résultat ne sont pas exactement 0 - elles sont très
```

```
# proches (-3.12250226e-17 = -3.12250226 * (10**17)) mais
# quand même différents de 0 ! Cette différence existe à cause
# de la représentation "imparfaite" des nombres réels dans
# les ordinateurs.


>>> np.linalg.det(A) # La déterminante de A
```

Ainsi, nous pouvons résoudre un système d'équations linéaires :
$x + 3y + 5z = 10$
$2x + 5y + z = 8$
$2x + 3y + 8z = 3$

Ce dernier peut être représenté sous la forme $A \times x = b$, où

$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{pmatrix}$ et $b = \begin{pmatrix} 10 \\ 8 \\ 3 \end{pmatrix}$.

Alors, nous pouvons simplement trouver la réponse du système $x = A^{-1} \times b$ par

```
>>> A = np.array([ [1.0,3.0,5.0], [2.0,5.0,1.0], [2.0,3.0,8.0] ])
>>> b = np.array([10.0, 8.0, 3.0])

>>> Ainv = np.linalg.inv(A)  # L'inverse de A

>>> x = np.dot(Ainv, b) # Et voici la solution !
>>> x

>>> np.dot(A, x) # Voyons si le résultat est correct !

>>> b # Et oui, en faisant la comparaison c'est le bon résultat !
```

Il y a un autre moyen pour trouver la solution d'un tel système. Il utilise un algorithme plus adapté à la résolution de ce type de problèmes, donc c'est préférable de l'utiliser directement au lieu de faire des transformations avec les matrices comme on vient de faire :

```
>>> x = np.linalg.solve(A, b) # Et c'est tout !
>>> x
```

Pour les curieux, la plupart des opérations en algèbre linéaire peuvent être effectuées avec NumPy. Un peu plus d'exemples peuvent être trouvées ici : `http://docs.scipy.org/doc/scipy/reference/tutorial/linalg.html`.

5.4 Exercices :

Trouver les solutions des systèmes suivants. Après avoir trouvé la réponse, vérifiez qu'elle satisfait le système en question.

- Système :
  $4x + 2y + z = 1$
  $3x + 8y + z = -2$
  $2x + 2y + 4z = 19$
- Système :
  $(13 + 2j)x + (3 + 1j)y + 52z = 1$
  $0.2x + (5 + 0.77j)y = 7.5 + 3j$
  $2.1x + 3.2y + (8.3 - 2.5j)z = 35$

6 Intégration

Ils existent des algorithmes pour l'intégration numérique d'une fonction donnée. La bibliothèque SciPy possède quelques techniques que nous allons voir dans cette section.

Commencez par importer la bibliothèque en question :

```
>>> import scipy.integrate as si

>>> help(si) # Voir les algorithmes d'intégration numérique
```

La fonction **quad** trouve l'intégrale d'une fonction d'une variable entre deux points. Par exemple, si vous voulez intégrer la fonction **sin** dans l'intervalle $[0; \pi]$: $I = \int_0^\pi sin(x)dx$

vous pouvez le faire comme ça :

```
>>> I = si.quad(np.sin, 0, np.pi)  # L'intégrale de la
# fonction sin entre 0 et pi

>>> I
(2.0, 2.2204460492503131e-14) # Ici la première valeur est
# l'estimation de l'intégrale. La deuxième
# nous indique l'erreur maximale par rapport aux vrai résultat.

>>> np.cos(np.pi) - np.cos(0) # Et une petite vérification du résultat...
```

Pour intégrer une fonction complexe nous devons la définir. Le moyen le plus simple c'est d'utiliser des expressions-λ en utilisant le syntaxe suivant :

```
>>> f = lambda x: np.sin(x) + np.cos(x) ** 2
>>> f

>>> f(3) # Calculer sin(3) + cos(3)^2
>>> f(4) # Calculer sin(4) + cos(4)^2

>>> g = lambda x,y: np.sin(x) + np.cos(y)

>>> g(1,1) # Calculer sin(1) + cos(1)
```

```
>>> g(0,0) # Calculer sin(1) + cos(1)
```

Maintenant pour trouver l'intégrale de f il suffit d'exécuter l'algorithme :

```
>>> si.quad(f, 0, 100) # L'intégrale de f dans [0; 100]
>>> si.quad(f, 100, 10) # L'intégrale de f dans [100; 10]

>>> si.quad(lambda x: x**3, 0, 1) # Trouver l'intégrale de x^3 dans [0; 1]
```

On peut également utiliser $\pm\infty$ comme limites d'intégration. Par exemple si on veut calculer $I = \int_0^\infty e^{-x}dx$ il suffit d'exécuter :

```
>>> f = lambda x: np.exp(-x) # La fonction à intégrer

>>> si.quad(f, 0, np.Inf) # L'intégrale
```

Pour trouver la valeur d'un double intégrale on peut utiliser la fonction *dblquad*. Par exemple, la valeur de $I = \int_0^\infty \int_1^\infty \frac{e^{-xt}}{t}dtdx$ peut être trouvée par :

```
>>> g = lambda t, x: np.exp(-x*t)/t # Fonction à intégrer.
# D'abord intégrer par t, ensuite par x.

>>> si.dblquad(g, 0, np.Inf, lambda x: 1, lambda x: np.Inf)
```

6.1 Exercices :

- Trouvez $I_1 = \int_0^5 (4x + 12)dx$.
- Trouvez $I_2 = \int_\pi^\pi sin^2 x dx$.
- Trouvez $I_3 = \int_0^\infty e^{-2x}dx$.

7 Graphiques

Il est possible d'afficher des graphiques avec Python en utilisant plusieurs bibliothèques. La plus connue est $matplotlib$ qui essaie de ressembler aux foncions graphiques de Matlab. Nous allons faire juste quelques exemples très simples pour montrer les principes de base.

Tout d'abord, importez la bibliothèque de graphismes :

```
>>> from matplotlib import pyplot as plt  # Importer la bibliothèque pyplot
# qui se trouve dans matplotlib et utiliser le nom plt comme raccourci
# (au lieu de pyplot).
```

Maintenant vois pouvez afficher des nuages de points avec la fonction **scatter** :

```
>>> x = np.random.uniform(0, 1, 1000) # Générer aléatoirement 100 valeurs
# uniformément répartis dans [0, 1]

>>> y = np.random.uniform(0, 1, 1000) # Générer aléatoirement 100 valeurs
# uniformément répartis dans [0, 1]

>>> plt.scatter(x, y) # Afficher un nuage de points avec
# cordonnées données par (x, y)

>>> plt.show() # Afficher la graphique

>>> x1 = np.random.normal(0.5, 0.1, 1000) # Générer aléatoirement 100 valeurs
# normalement répartis dans avec moyenne 0.5 et écart-type 0.1

>>> y1 = np.random.normal(0.5, 0.1, 1000) # Générer aléatoirement 100 valeurs
# normalement répartis dans avec moyenne 0.5 et écart-type 0.1

>>> plt.scatter(x1, y1, color='red') # Afficher un nuage de points rouges avec
# cordonnées données par (x1, y1)
```

Pour afficher une fonction nous pouvons utiliser les fonctions suivantes :

```
>>> x = np.arange(-np.pi, np.pi, 0.1) # Générer des valeurs pour x
# de -pi à pi avec un pas de 0.1 radians
>>> x

>>> y = np.sin(x)
>>> y

>>> plt.plot(x, y) # Afficher la graphique
```

Matplotlib donne des possibilités très vastes pour la génération des graphiques. Pour plus d'information vous pouvez consulter la gallérie des exemples à l'adresse suivante `http://matplotlib.sourceforge.net/gallery.html`.