Programmation pour la physique

UE HLPH609, Faculté des Sciences de Montpellier, 2020
Yohann Scribano (yohann.scribano@umontpellier.fr)
&
Felix Brümmer (felix.bruemmer@umontpellier.fr)

Introduction

Dans ce chapitre

Généralités

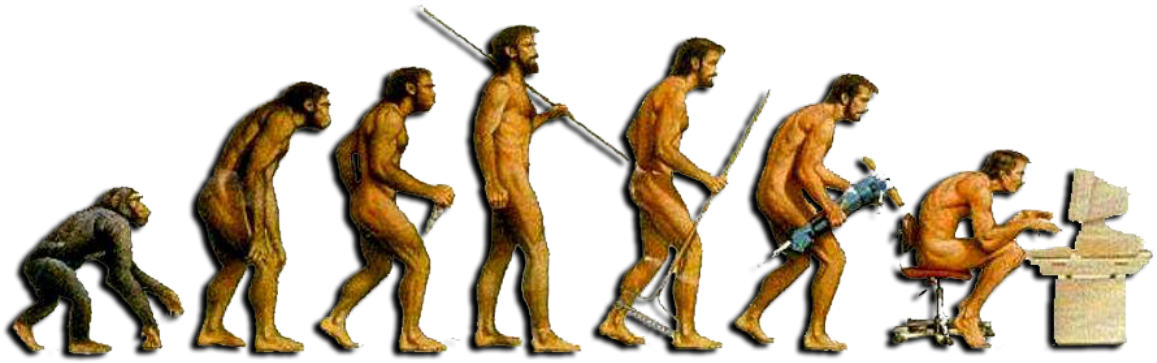
- Programmation pour la physique
- Le langage Python
- Exécuter un script Python

Python

- Les instructions et les commentaires

Pourquoi apprendre à programmer ?

- Développer des programmes.
- Modifier et adapter à ses besoins des programmes et bibliothèques existants.
- Connaitre le fonctionnement et les limitations des logiciels que l'on utilise.



Qu'est-ce qu'un programme ?

Un programme est un ensemble de commandes qui amènent l'ordinateur à changer l'état de sa mémoire interne et/ou de ses périphériques.

Typiquement nous commanderons l'ordinateur d'effectuer certaines opérations et d'afficher ou d'enregistrer les résultats.

Un langage de programmation est un ensemble de règles syntactiques que les commandes doivent suivre afin qu'elles puissent être traduites en instructions au système d'exploitation (ou directement au matériel informatique).

Pour la traduction des commandes (du "code source", compréhensible par un être humain) en instructions binaires (compréhensible par l'ordinateur) il y a deux approches : la compilation et l'interprétation. Selon l'option préférable on parle des langages compilés ou interprétés (même si les deux sont a priori possible pour tout langage).

Langages de programmation

```python
# Afficher "Hello World!" avec Python

print("Hello World!")
```

```cpp
// Afficher "Hello World!" avec C++

#include<iostream>
using namespace std;

int main() {
    cout << "Hello World!\n";
    return 0;
}
```

```fortran
! Afficher "Hello World!" avec Fortran 90

PROGRAM HelloWorld
   WRITE (*,*) 'Hello World!'
END PROGRAM HelloWorld
```

Python

Pour ce cours on va se servir du langage de programmation `Python`.

Pourqoui Python ?

- tres répandu
  (devenant le standard universel en programmation scientifique, sauf pour le calcul à haute performance)
- facile à apprendre, syntaxe simple et intuitive
  $\Rightarrow$ permet de se concentrer sur la programmation sans être encombré par les pièges du langage
- polyvalent, de multiples domaines d'application
- beaucoup des bibliothèques incluses, vaste fonctionnalité
- haut niveau d'abstraction (pas de manipulation directe du matériel informatique)
- moderne : inclut des concepts comme la programmation orientée objet, la programmation fonctionnelle, le typage dynamique, la gestion automatique de mémoire. . .

Domaines d'application de la programmation scientifique

Calcul symbolique / calcul formel

- Manipulation automatisée des objets mathématiques au niveau symbolique, c.à.d. sans forcement une application numérique : Algèbre, analyse, arithmétique...
- Les systèmes de calcul formel incluent typiquement des langages de programmation complets pour gérer le flot d'exécution. Pas seulement des "grandes calculatrices" !
- Interfaces typiquement interactives de type notebook (calepin / cahier de travail).
- Exemples : `Mathematica`, `Maple`, `MATLAB`, `SageMath/SymPy`

* propriétaire * basé sur `Python`

Domaines d'application de la programmation scientifique

Calcul numérique à haute performance

- Evaluation numérique des fonctions / des intégrales à haute précision, solution numérique de systèmes d'équations algébriques ou différentielles, optimisation, simulation de systèmes complexes avec un grand nombre de dégrés de liberté. . .
- Domaine classique de l'analyse numérique.
- Les problèmes peuvent être très demandeur côté puissance de calcul : besoin d'optimisation de code (automatisée ou de la part du programmeur) pour exploiter au mieux le matériel informatique.
- On utilise les langages de programmation compilés pour optimisation, souvent de bas niveau d'abstraction pour minimiser l'overhead.
- Interface : soit éditeur + console avec ses outils (compilateur, lieur, débogueur), soit environnement de développement intégré
- Exemples : `FORTRAN`, `C`, `C++`

Domaines d'application de la programmation scientifique

Gestion, traitement, analyse et visualisation des données

- Traitement automatisé des données expérimentales, observationnelles, numériques...
- Stockage, tri, analyse statistique, ajustement, représentation graphique (création de figures ou des animations)...
- Mons besoin de puissance de calcul, plus besoin de fonctionnalités diverses (polyvalence)
- Domaine des langages de programmation interprétés.
- Interface :
  - soit fichiers de script (crée par éditeur ou environnement de développement intégré)
  - soit interface interactif (notebook ou ligne de commande)
- Exemples de langages de script : script shell de Unix, `Perl`, `R`, `Julia`, `Python`
- Exemples de suites logicielles interactives : `MATLAB`, `Octave`, `IPython/Jupyter`

* propriétaire * basé sur `Python`

Domaines d'application de la programmation scientifique

Développement des logiciels et outils auxiliaires

- Programmation des interfaces, de l'environnement graphique, du web, du système d'exploitation...
- Interférences avec d'autres secteurs du développement des logiciels
- Domaine des langages universels, interprétés ou compilés selon la tâche à remplir
- Interface : typiquement environnement de développement intégré
- Exemples : `C++`, `Java`, `Python`

Deux façons d'exécuter son programme : Compilation et interprétation

1. Programmes compilés :

Le code source est entièrement traduit en forme exécutable par un logiciel auxiliaire, le compilateur. Il peut être assisté par un autre logiciel auxiliaire, l'éditeur de liens, pour intégrer les composantes du programme et les bibliothèques externes dans le fichier exécutable.

Forces :

- Le compilateur peut automatiquement optimiser le code lors de la traduction.
- Pour cette raison les programmes compilés sont souvent très rapides et efficaces.
- Manque de transparence : les producteurs de logiciels commerciaux peuvent cacher les détails de fonctionnement des programmes compilés.

Faiblesses :

- Après toute modification du code source il faut recompiler.
- Manque de portabilité : chaque système d'exploitation a son propre compilateur et sa propre version du code compilé.
- Pas de possibilité d'exécuter seulement une partie d'un programme
- Manque de transparence : sans le code source, l'utilisateur ne sait pas que fait le programme en détail

Deux façons d'exécuter son programme : Compilation et interprétation

2. Programmes interprétés :

Le code source est traduit ligne par ligne lors de l'exécution par un logiciel auxiliaire, l'interpréteur.

Forces :

- Après une modification du code source on peut immédiatement réexécuter le nouveau programme.
- Les erreurs de programmation sont souvent plus faciles à détecter (débogage)
- Portabilité : les seuls fichiers de programme sont celles du code source, qui sont les mêmes sur tout système

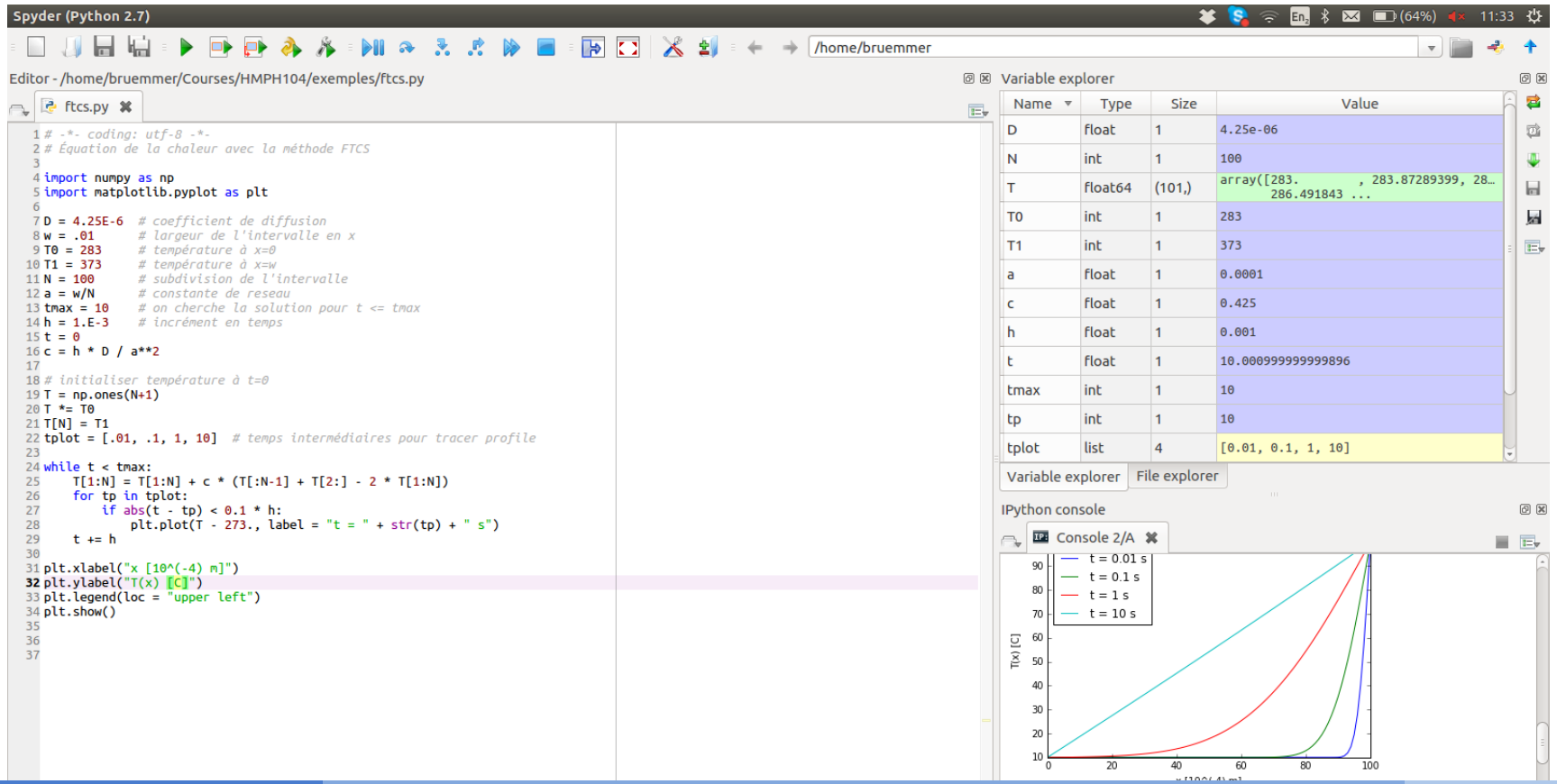
Faiblesses :

- Efficacité très réduite par rapport à un programme compilé et optimisé.

Deux façons de rédiger son code : Fichier de code source et notebook

1. Fichier de code classique / script :

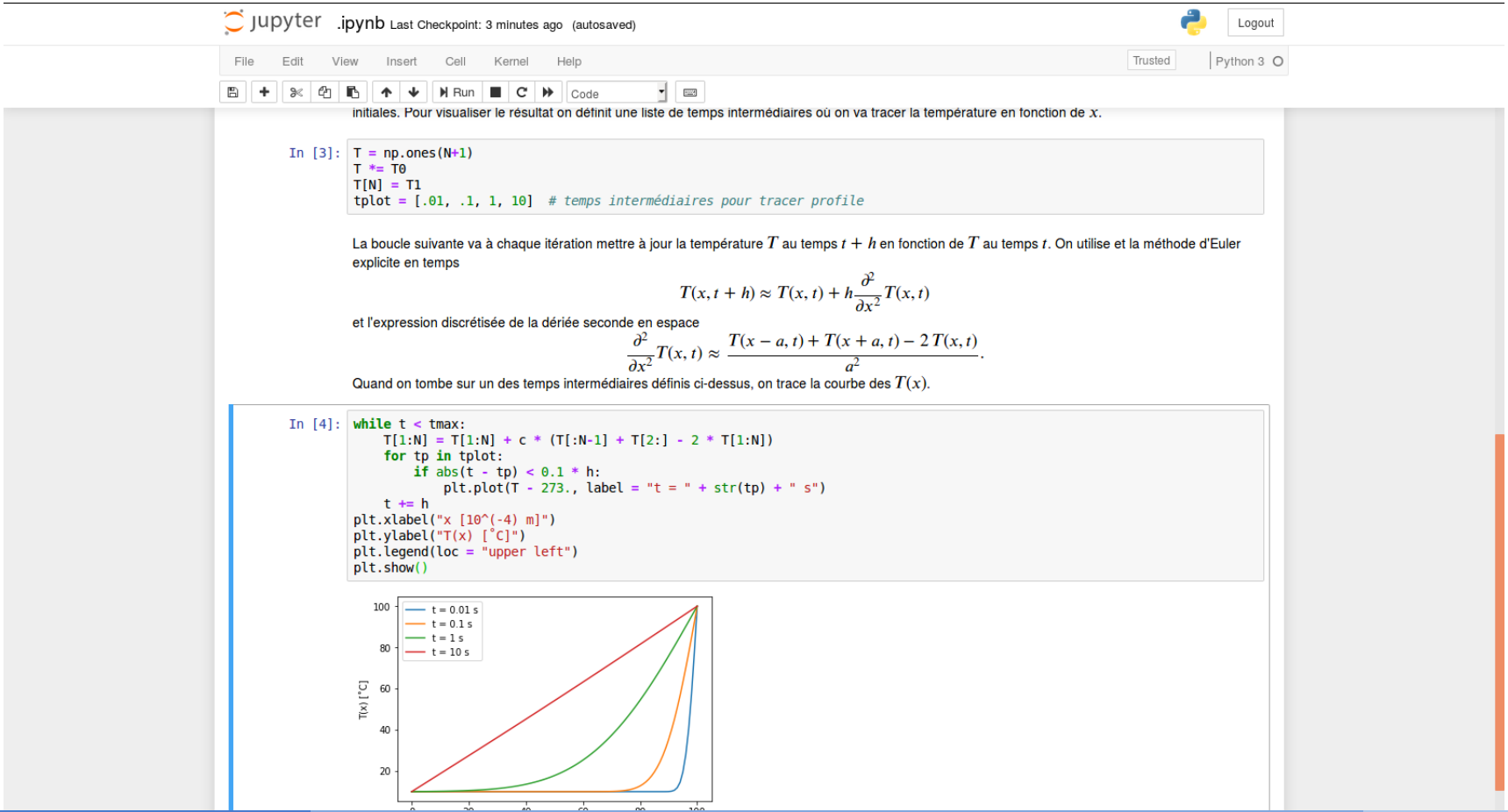
- Créé avec un éditeur de texte (qui peut faire partie d'un environnement de développement intégré).
- Format : fichier de texte, ne contient que le code de source et des commentaires
- Le code est enregistré dans un ou plusieurs fichiers que l'on passe ensuite au compilateur ou à l'interpréteur.



Deux façons de rédiger son code : Fichier de code source et notebook

2. Notebook interactif :

- Créé avec un éditeur dédié.
- Format spécial, peut contenir du texte formaté, des équations, des résultats intermédiaires, des éléments graphiques. . . ainsi que le code de source
- Interpréteur toujours intégré, on peut exécuter le notebook entier ou juste une partie



Deux paradigmes de programmation : Procédurale et orientée objet

1. Programmation procédurale :

- Date des années '50
- Séparation entre les données et les procédures qui les gèrent
- Structure des programmes plus linéaire
- Mieux adaptée aux petits projets car moins d'overhead.
- Exemples de langages bien adaptés à la programmation procédurale : `FORTRAN`, `C`, `Python`

Deux paradigmes de programmation : Procédurale et orientée objet

2. Programmation orientée objet :

- Date des années '80
- Notion centrale : L'objet qui réunit les données et les méthodes, représentant une entité abstraite définie par son état et ses capacités
- Tout objet est instance d'une classe. Il y a une hiérarchie des classes avec des propriétés héritables.
- Structures plus abstraites.
- Programmes moins linéaires, favorisant la modularité
- Mieux adaptée aux grands projets de plusieurs contributaires.
- Exemples de langages bien adaptés à la programmation orientée objet : `C++`, `Java`, `Python`

Python

Le langage Python

- est un langage interprété

  Plus précisement, son implémentation standard CPython traduira le code source en "bytecode" qui est ensuite interprété.

- s'utilise soit en mode de script soit en mode interactif
- permet tant la programmation procédurale que la programmation orientée objet.

Programmation pour la physique

Ce cours est une introduction à plusieurs sujets importants pour la programmation scientifique :

- Révision de la programmation procedurale avec le langage Python
- Initiation aux principes élémentaires de la programmation orientée objet
- Problèmes choisis de l'algorithmique : Algorithmes de tri
- Problèmes choisis de l'analyse numérique : Recherche des zéros, méthodes de l'algèbre linéaire numérique

Prérequis pour le suivre avec profit :

- Connaissances de base en informatique (Unix/Linux)...
- ...et en mathématique (nombres complexes, analyse réelle, algèbre linéaire, probabilité).

Matière à revoir indépendamment si nécessaire !

- La plupart d'entre vous ont déjà travaillé avec Python. On va pourtant revoir les éléments de base ensemble, ce qui permettra aux autres de l'apprendre.

Bibliographie

Python :

- B. Cordeau et L. Pointal, « Une introduction à Python 3 », `http://perso.limsi.fr/pointal/python:courspython3`
- G. Swinnen, « Apprendre à programmer avec Python 3 », `http://www.inforef.be/swi/python.htm`
- D. Cassagne, « Introduction à Python pour la programmation scientifique », `http://wwww.courspython.com`
- « Python Tutorial », `https://docs.python.org/fr/3/tutorial/`
- beaucoup d'autres sources à trouver en ligne et hors ligne

Algorithmes pour la physique numérique :

- M. Newman, « Computational Physics », 2012 (en anglais)
- W. H. Press, S. Teukolsky, W. Vetterling et B. Flannery, « Numerical Recipes », 3e édition 2007, Cambridge University Press (en anglais et C++)

Matériel du cours sur l'espace pédagogique

Vous trouverez sur Moodle :

- Ces notes de cours : `Programmation_Notes.pdf`
- Tous les exemples de code apparaissant ci-dedans : répertoire `Exemples/`
- Toutes les fiches d'exercices
- Anciens épreuves
- Collection de quelques commandes utiles : `Programmation_Reference.pdf`

Exécuter son code avec Python

Plusieures possibilités :

- mode interactif :
  - soit passer les instructions à l'interpréteur une par une avec une interface en ligne de commande, similaire à celle du shell Unix
  - soit utiliser un notebook pour grouper plusieures instructions à exécuter à la fois
- mode de script : passer toutes les instructions à l'interpréteur simultanément par un fichier de code source, le script Python

Ici nous préférions le mode de script.

Pour le mode de script, deux étapes :

- Enregistrer toutes les instructions du programme dans un fichier de script
- Exécuter le script avec l'interpréteur Python

En revanche, pour un langage compilé :

1. Enregistrer les instructions dans un fichier
2. Traduire le code source en code binaire avec le compilateur, assembler le fichier exécutable avec le lieur
3. Exécuter le programme

Outils utiles

Simples éditeurs de texte pour créer et éditer les scripts :

- `kwrite` ou `gedit`
- pour les spécialistes : `vim/vi` ou `nano` ou `ne` ou `emacs`

Émulateurs de terminal / console pour exécuter les scripts dans le shell :

- `konsole` ou `gnome-terminal` ou `lxterminal` ou `xterm`

Environnements de développement intégrés = éditeurs spéciaux avec accès direct à l'interpréteur (ainsi qu'au débogueur et aux outils de gestion de projet) :

- `spyder` > 2.3
- `NetBeans` 8.0.2 + plugin Python
- `kdevelop` > 4.6 + `kdev-python` $> 1.6.1$
- `Eclipse` + plugin `PyDev`

Déconseillés :

- `Canopy` (Environnement pour Python 2, ne marche pas avec Python 3)
- éditeurs sans coloration syntactique (logiciels de traitement de texte etc.)

Outils conseillés

Conseillé : soit Spyder soit éditeur de texte + console

Par exemple :

- `gedit` : cliquer sur `Applications` → `Outils` → `gedit`
- `gnome-terminal` : cliquer sur `Applications` → `Outils` → `Terminal`

Exécuter son code avec Python

La convention pour l'extension de fichier des scripts Python est `.py`.

1. Sauvegarder le code dans un fichier nommé, par exemple, `exemple.py`
2. **Soit :** Entrer `python3 exemple.py` par la console (dans le dossier où se trouve le fichier de script)
3. **Ou :**
   - Inserer `#!/usr/bin/python3` au début du fichier
   - Rendre le fichier exécutable en entrant `chmod +x exemple.py` par la console
   - Exécuter le script directement en entrant `./exemple.py`

Notre premier script Python

```
#!/usr/bin/python3
# Ecrit la phrase "Hello World!" sur l'écran

print("Hello World!")
```

Ci-dessus nous voyons

- des commentaires : précédés par un croisillon #. Tout ce qu'y fait suite dans la même ligne du fichier est ignoré par l'interpréteur.
  On utilise des commentaires surtout pour rendre son code mieux lisible par les programmeurs (soi-même inclus). Il ne faut pas les économiser !
- une ligne blanche, ignorée par l'interpréteur
- une instruction : dans ce cas, un appel à la fonction `print( )` qui écrit à l'écran la chaîne de caractères dans les parenthèses

Exercice

Exécuter ce script (`Ex01_HelloWorld.py`)

Format et interprétation du code source

- Chaque ligne du script (à part les lignes blanches et les lignes qui ne contiennent que des commentaires) correspond à une instruction.
- L'interpréteur exécutera toutes les instructions, une par une.
- Si l'interpréteur tombe sur une instruction fautive, le programme s'arrête avec un message d'erreur. Ce message peut être très utile pour identifier et réparer le problème.

```
pint("Oups!") # erreur: la fonction s'appelle print()
              #
              # Le programme va s'arrêter avec le message
              #  "NameError: name 'pint' is not defined"
              # qui indique que 'pint' n'est pas défini
```

- Sinon, le programme termine dès qu'il n'y a plus d'instructions à exécuter.

Exceptions de la règle d'une instruction par ligne

- Une instruction avec des parenthèses (ou crochets [ou accolades { pas fermés se poursuit sur les lignes suivantes jusqu'à la clôture.

```
print("Malheureusement ce texte est trop long pour une "
"seule ligne de code source, mais on veut cependant "
"l'afficher dans une seule ligne sur l'écran.")
```

- On peut aussi terminer une ligne avec un anti-slash \ pour indiquer qu'une instruction se poursuit sur la ligne suivante.
- On peut grouper plusiers instructions dans une seule ligne si on les sépare avec un point-virgule ;

```
print("Flying"); print("Circus")
```

Pour améliorer la lisibilité du code il est fortement conseillé de mettre une instruction par ligne et une ligne par instruction si possible.

Les instructions : Erreurs fréquentes

- À la différence de Python 2, `print()` est une fonction en Python 3 — il est impératif d'écrire les parenthèses `()`

```
print("Ca marche")          # ça marche
print "Ca ne marche pas"    # ça ne marche pas
```

- Attention à l'orthographe. En particulier, Python est sensible à la casse (distingue entre les minuscules et les majuscules).
- Si un programme ne fonctionne pas : Faire attention au message d'erreur. Il contient des informations utiles sur l'endroit où le programme s'est interrompu (la ligne du code source) et sur le genre d'erreur qui s'est produit.

Les types de données

Dans ce chapitre

Python

- Les variables et les affectations
- Les types de données numériques
- Les opérations arithmétiques
- Les types de données séquentiels
- Les chaînes de caractères
- La saisie du clavier

Les variables, les types et les affectations

Voici quelques exemples d'affectations qui attribuent des valeurs aux variables :

```
phrase = "Mais non!"
nombre = 25
somme = nombre + 5    # la valeur de 'somme' devient 30
nombre3 = 50.0
liste = ["lundi", "mardi", "mercredi"]
```

- Ici `phrase`, `nombre`, `somme`, `nombre3` et `liste` désignent des variables.
- Toute variable est d'un type qui résulte du format utilisé dans l'affectation. Ici '`phrase`' est du type str (chaîne des caractéres), '`nombre`' et '`somme`' sont du type int (nombres entiers), '`nombre3`' est du type float (nombre flottant) et '`liste`' est du type list (liste d'objets).
- Après initialisation la variable peut être utilisée, cf. l'usage de '`nombre`' dans la troisième ligne ci-dessus. En revanche, une commande comme

```
a = b
```

produit une erreur si la variable `b` n'a pas été donnée une valeur avant.

Les variables

Chaque variable est caractérisée par

- son nom (identifiant)
- son type
- sa valeur

Exemple :

```
ma_variable = 25
```

- Ici l'identifiant est `ma_variable`.
  Un identifiant se compose des lettres `A-Z` et `a - z`, du tiret bas `_` et des chiffres `0 - 9` (sauf pour le premier caractère).
- Le type est déterminé automatiquement à l'initialisation. Ici le type est `int` (nombre entier). Si l'initialisation était `ma_variable = 25.0` le type serait `float` (nombre flottant). Si c'était `ma_variable = "vingt-cinq"` le type serait `str` (chaîne des caractères).
- La valeur est 25, bien sûr.

Les identifiants

- Presque toute combinaison de lettres minuscules et majuscules, chiffres (sauf pour le premier caractère) et tirets bas _ est valable comme identifiant.
- Exception : les mots clé du langage Python qui ont une signification syntactique spéciale, soient `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `False`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `None`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `True`, `try`, `while`, `with`, `yield`.
- Deuxième exception : Il y a certaines fonctions natives dont les noms ne doivent pas être utilisés comme identifiants, par exemple `abs`, `complex`, `float`, `input`, `int`, `list`, `max`, `min`, `print`, `str` (même si c'est techniquement possible, ça va entraîner des conséquences imprévisibles).
- Troisième exception : l'identifiant '`self`' ainsi que tout identifiant qui commence avec un ou deux tirets bas (par exemple, '`__init__`') ont une signification spéciale en programmation orientée objet. Il ne faut pas les utiliser hors leur propre contexte.
- Conseil : Utiliser les identifiants parlants pour les variables importantes pour améliorer la lisibilité du code.

```
x = "2 place Victor Hugo, 75000 Paris, France"   # pas idéal
adresse = "34 rue Jean Moulin, 30000 Nîmes, France"  # mieux
```

Les types de données numériques

type	description
`int`	nombre entier sans limite de taille théorique
`float`	nombre flottant, précision 64 bit soit $\approx$ 15 décimales
`complex`	nombre flottant complexe correspondant à deux `float`

Les types de données numériques : `int`

- Une variable du type `int` (nombre entier) se crée par une affectation sans point décimal comme

```
x = 25
y = -100
```

- Elle est également le résultat d'un appel à la fonction de conversion `int()` :

```
x = 3.8      # x est du type float avec valeur 3.8
y = int(x)   # y est du type int avec valeur 3

z = int("42") # z est du type int avec valeur 42 et a été
              #  construit partant de la chaîne de
              #  caractères "42"
```

Les types de données numériques : `float`

- Une variable du type `float` (qui représente un nombre flottant avec valeur absolue entre environ 10^{-300} et 10^{300} ou 0, précision numérique ≈ 15 décimales) se crée par une affectation soit avec point décimal soit en notation scientifique :

```
gamma = -5.77
c = 3E8
hbar = 1.05E-34
```

- Ici la notation '3E8' signifie 3×10^{8} et la notation '1.05E-34' signifie 1.05×10^{-34}.
- Le résultat d'un appel à la fonction de conversion `float()` est également un nombre flottant :

```
x = 3           # x est entier avec valeur 3
y = float(x)    # y est un float avec valeur 3.0
```

Les types de données numériques : complex

- Une variable du type `complex` représente un nombre flottant complexe, c.à.d. un `float` pour la partie réelle et un pour la partie imaginaire. Notation en Python avec 'J' pour l'unité imaginaire $i = \sqrt{-1}$:

```
c = 3 + 4J              # le nombre complexe 3 + 4 i
d = -2.5 + 3.E-1J       # le nombre complexe -2.5 + 0.3 i
i = 1J                  # le nombre complexe i
```

- La conversion se fait avec la fonction de conversion `complex()` :

```
s = "3 + 2J"     # une chaîne de caractères
z = complex(s)   # un nombre complexe
```

Les opérations arithmétiques

Python connaît les opérations arithmétiques suivants :

$+$	addition	
$-$	soustraction	
$*$	multiplication	
$/$	division réelle	résultat est toujours `float` ou `complex`
$//$	division entière	résultat est `int` si les deux arguments sont `int`, `float` ou `complex` sinon
%	reste de la division entière	résultat est `int` si les deux arguments sont `int`, `float` ou `complex` sinon
$**$	puissance	

Exemples :

```
x = 5 + 3      # x est du type int avec la valeur 8
y = x / 5      # y est du type float avec la valeur 1.6
z = x // 5     # z est du type int avec la valeur 1
yy = 8 % 5.0     # yy = 3.0 est du type float
xx = 2 ** 3      # xx = 8
zz = 16 * 4 - 2 * (10 + 1)    # zz = 42
```

Les opérations arithmétiques

Exercices

- Calculer 5^7, 1503.0/1726.0, 7! avec un script Python, et écrire les résultats sur l'écran en utilisant la fonction `print()`.

Les affectations

Une affectation procède en deux pas :

- l'expression à droite du signe = est évaluée
  (calculée en fonction de l'état de la mémoire à cet instant)
- le résultat est affecté à la variable dont le nom figure à gauche du signe =

Cela permet des instructions comme

```
x = 0          # x est du type int, sa valeur est 0
x = x + 1      # x est augmenté de 1
```

(la 2nde ligne ne doit pas être confondue avec une équation algébrique !)
ou même

```
x = y = 7      # x et y sont des int avec la valeur 7
```

Les affectations

Combinaison des opérations arithmétiques avec des affectations :

```
x = 2    # x est du type int avec la valeur 2
x += 1   # ajouter 1 à x (équivalent: x = x + 1): x = 3
x *= 3   # multiplier la valeur de x par 3: x = 9
x -= 1   # soustrahir 1 de la valeur de x: x = 8
x /= 4   # diviser x par 4: x = 2
```

Les types séquentiels (“sequence types”)

type	description
`str`	chaîne de caractères
`list`	liste muable d’objets
`tuple`	collection immuable d’objets

Les chaînes de caractères

Le type de données `str` ("string" en anglais) représente des chaînes de caractères.

- Toute partie du code source entre apostrophes ' ou guillemets " est interprétée comme un `str`.

```
phrase = "Mon tailleur est riche."
titre_du_cours = 'Programmation pour la Physique'
```

- On peut convertir tout variable numérique en `str` avec la fonction `str()`. On peut convertir un `str` en `int` seulement s'il se compose des chiffres. Similairement, on peut convertir un `str` en `float` seulement si les caractères représentent un nombre flottant.

```
nombre = "23"
print(nombre + nombre)    # affiche "2323"
print(int(nombre) + int(nombre)) # affiche "46"
```

Les opérations sur des chaînes de caractères

Avec l'opérateur + on peut composer les chaînes de caractères :

```
age = 12
phrase = "J'ai " + str(age) + " ans"
print(phrase)  # résultat: "J'ai 12 ans"
```

Avec [ ] on les indice. Attention : l'indice du premier caractère est toujours 0 !

```
print(phrase[0])       # résultat: "J"
print(phrase[3])       # résultat: "i"
```

L'opérateur `[a:b]` retourne la sous-chaîne à partir de l'indice `a` (inclu) jusqu'à l'indice `b` (exclu). Si on ne spécifie pas `a` et/ou `b`, on obtient la sous-chaîne à partir du début et/ou jusqu'au bout.

```
print(phrase[1:3])       # résultat: "'a"
print(phrase[5:])        # résultat: "12 ans"
```

Le caractère d'échappement dans une chaîne de caractères

Dans une chaîne des caractères, le anti-slash \ a une rôle spéciale : c'est le caractère d'échappement.

- Si un string littéral commence avec une apostrophe ', les caractères " sont traités comme un caractère régulier et vice-versa. Si on veut inclure le caractère ' (ou ") dans un string littéral qui commence avec ' (ou "), il doit être précédé par un \ :

```
print("J'ai 12 ans")
print('J\'ai 12 ans')  # même résultat
print("\"M'enfin!\", s'exclame Gaston")
```

- La séquence \n dans une chaîne des caractères force la fin de la ligne.
- Pour explicitement écrire un anti-slash il faut en inclure deux :

```
print("Voici un anti-slash: \\")
```

(On rappelle que, hors d'une chaîne des caractères, le anti-slash indique par contre la continuation d'une instruction sur la ligne suivante)

Faire entrer une chaîne de caractères par le clavier

Pour récuperer des données du clavier on se sert de la fonction `input()` :

```
s = input("Entrez quelque chose:")
print("Vous avez entré \"" + s + "\"")
```

`input()` retourne toujours un str. Si on veut lire des données numériques du clavier, il faut les convertir :

```
s = input("Entrez un nombre entier:")
i = int(s)
print(s + " fois " + s + " est " + str(i**2))
```

Les types séquentiels : `list`

Le type `list` représente une liste d'objets. Toute partie du code source entre des crochets `[ ]` est interprétée comme liste. Les éléments sont séparés par des virgules `,`.

```
nombres_premiers = [2, 3, 5, 7, 11]
mois_hiver = ["décembre", "janvier", "février"]
```

- Les éléments d'une liste ne sont pas forcement du même type. Il peut y avoir des doublons.

```
liste_bizarre = [2, 2.0, 2 + 0J, "deux"]
L = [0, 0, 0]
```

- On peut même construire des listes dont les éléments sont des listes :

```
matrice3x2 = [[1, 2, 3], [6, 5, 4]]
```

Opérations sur les listes

Comme pour les `str` :
Avec l'opérateur + on peut joindre une liste à une autre :

```
mois = ["Janvier", "Février", "Mars"]
mois2 = ["Avril", "Mai"]
print(mois + mois2)
# affiche "['Janvier', 'Février', 'Mars', 'Avril', 'Mai']"
```

Avec [ ] on les indice. Attention : l'indice du premier élément est toujours 0 !

```
print(mois[1])     # affiche "Février"
print(mois[0][0]) # première lettre du premier élément = "J"
```

L'opérateur `[a:b]` retourne la sous-liste entre l'indice `a` (inclu) et l'indice `b` (exclu). Si on ne spécifie pas a (ou b), on obtient la sous-liste à partir du début (ou jusqu'au bout).

```
print(mois[:1])       # affiche "['Janvier']"
print(mois2[:])       # affiche "['Avril', 'Mai']"
```

Les types séquentiels : `tuple`

Le type `tuple` représente une collection d'objets similaire à une liste, mais avec une différence importante : Les `tuple` sont immuables, ils ne peuvent pas être modifiés après initialisation. En pratique ils sont moins utilisés que les listes.

Un `tuple` est crée par des parenthèses `( )` avec les éléments séparés par des virgules `,`. On peut même supprimer les parenthèses en absence d'ambiguïté. Exemples d'utilisation :

```
t = (1, 2, 3)   # crée un tuple
u = 1, 2, 3     # le même tuple
print(u[2])     # affiche "3"

a, b = 1, 2     # affecte les valeurs 1 à a et 2 à b

jour = 21
print("Nous sommes le", jour, "janvier")
```

Exercices

- Examiner le code `Ex03_Strings.py` et faire l'interpréteur Python : Qu'est-ce que ce script affichera ? Puis vérifier votre résultat en exécutant le code sur l'ordinateur, et le modifier à volonté.
- Réaliser un script qui lit deux nombres flottants du clavier et qui retourne leur produit.
- Réaliser un script qui demande à l'utilisateur de fournir trois mots par le clavier. Puis, il les écrit sur l'écran
  - dans l'ordre inverse qu'ils ont été fournis
  - en supprimant le premier caractère
  - en supprimant tous les caractères à part le premier
  - sans espaces.

  Exemple :

```
Tapez le premier mot:chien
Tapez le deuxième mot:chat
Tapez le troisième mot:souris
souris chat chien
hien hat ouris
c c s
chienchatsouris
```

Les variables et les types de données : Erreurs fréquentes

- Orthographe : `ma_var`, `Ma_var` et `mavar` sont trois identifiants différents.
- Effectuer une opération, puis ne rien faire avec le résultat ne sert à rien :

```
nombre = 5
nombre * 3  # calcule 5 * 3 et oublie le résultat
```

- Ne pas oublier de convertir ses variables au bon type :

```
age = input("Votre age: ") # input() retourne un str
nais = 2017 - age # erreur: faut convertir age en int
```

- L'indice du premier objet dans une séquence est 0. Si la séquence contient n objets, l'indice du dernier est alors $n - 1$.

```
ma_liste = ["un", "deux", "trois"]
print(ma_liste[1])  # affiche "deux"
print(ma_liste[3])  # erreur: pas de 4-ème élément
```

Les structures de contrôle

Dans ce chapitre

Python

- Les blocs d'instructions
- La structure conditionnelle
- Les expressions logiques
- La priorité des opérateurs
- La boucle `while`
- La boucle `for`
- Les exceptions

Les blocs d'instructions

Un bloc est une séquence de lignes d'instructions distingués par leur indentation. Une ou plusieures lignes consecutives indentées au même niveau constituent un bloc.

```
INSTR1
INSTR2
    INSTR3      # première ligne d'un bloc
                # (les lignes blanches sont ignorées)
    INSTR4      # deuxième ligne du même bloc
INSTR5          # cette ligne ne fait plus partie du bloc
INSTR6
    INSTR7      # ici commence un deuxième bloc
    INSTR8      # cette ligne fait partie du deuxième bloc
        INSTR9  # ici commence un sous-bloc
        INSTR9
        INSTR10
    INSTR11   # cette ligne fait toujours partie du 2ème bloc
...
```

Les structures de controle permettent d'exécuter toutes les instructions d'un bloc plusieurs fois, ou de les exécuter seulement en fonction d'une condition.

La structure conditionnelle

La structure conditionnelle `if` prend la forme suivante :

```
if CONDITION:
    INSTRUCTION1      # bloc à exécuter si CONDITION vérifiée,
    INSTRUCTION2      #  à sauter sinon
    ...
    DERNIERE_INSTRUCTION  # ici le bloc se termine
CONTINUER_ICI    # en tout cas le programme reprend ici
...
```

- Ici `CONDITION` est une expression logique de valeur `True` (vrai) ou `False` (faux).
- Le bloc d'instructions suivant est exécuté seulement si `CONDITION` est `True`.
- Autrement le script saute le bloc et continue après à `CONTINUER_ICI`.
- Le deux-points : après `CONDITION` fait partie de la structure et ne doit pas être omis

La structure conditionnelle

Exemples :

```
i = int(input("Entrez un nombre entier: "))
if i < 0:
    i = -i  # un bloc qui ne contient qu'une seule ligne
print("La valeur absolue de ce nombre est", i)
```

```
print("Vous voulez savoir un sécret?")
reponse1 = input("Entrez 'o' si oui: ")
if reponse1 == "o":
    reponse2 = input("Vous êtes sur? Entrez 'o' si oui:")
    if reponse2 == "o":
        print("Le voici:\nLa cuillère n'existe pas.")
```

Parenthèse : Les expressions logiques

Le type de données `bool`

Ce type de données représente la valeur booléenne d'une expression logique. Les variables du type `bool` ne peuvent prendre que deux valeurs différentes : `True` (vrai) ou `False` (faux).

On peut définir des variables booléennes de la même manière que des variables numériques, par exemple

```
flag = True
...
if flag:
    FAIRE_QUELQUE_CHOSE
    ...
```

Par la fonction `bool()` on peut convertir un `str` en `bool` (s'il s'agît de la chaîne de caractères `"True"` ou `"False"`). De même pour une variable numérique (dans ce cas le résultat est `False` si le nombre est 0 et `True` sinon), ou on peut directement utiliser la valeur numérique correspondante dans une structure conditionnelle au lieu de la condition.

Les expressions logiques

Les opérateurs de comparaison :

expression	`True` si ...
`x == y`	`x` est égal à `y`
`x != y`	`x` est différent de `y`
`x > y`	`x` est strictement supérieur à `y`
`x < y`	`x` est strictement inférieur à `y`
`x >= y`	`x` est supérieur ou égal à `y`
`x <= y`	`x` est inférieur ou égal à `y`

Les opérateurs logiques : soient `a` et `b` du type bool (`True` ou `False`)

`not a`	`True` si `a` est `False` et vice-versa
`a and b`	`True` si `a` est `True` et `b` est `True`, `False` autrement
`a or b`	`True` si au moins un de `a` ou `b` est `True`, `False` autrement

Les opérateurs `in` et `is`

L'opérateur `in`

teste si un objet est contenu dans une séquence :

```
animaux = ["giraffe", "gazelle", "guépard"]
"giraffe" in animaux   # True
"goléon" in animaux    # False
"elle" in "gazelle"    # True
```

L'opérateur `is`

teste si deux identifiants désignent le même objet (il ne teste pas l'égalité des valeurs) :

```
x = [1, 2]      # une liste avec deux entrées
y = [1, 2]      # une autre liste avec les mêmes entrées
z = x     # z est un autre nom pour x
x is y    # False
x is z    # True
```

Cet opérateur peut parfois donner des résultats inattendus sur des variables immuables (types numériques, `str`...). On comprendra plus tard pourquoi.

Fin de parenthèse : La priorité des opérateurs

En ordre ascendant :

- `or`
- `and`
- `not`
- comparaisons : `==`, `!=`, `>`, `<`, `>=`, `<=`, `in`, `is`
- addition et soustraction : `+`, `-`
- multiplication et division : `*`, `/`, `//`, `%`
- signe : `+x`, `-x`
- exponentiation : `**`

Ainsi l'expression "`not x > y or - x ** y + 2 * y == 0`" est interprétée

$$(\neg(x > y)) \vee (((-(x^y)) + (2 \times y)) = 0)$$

On peut toujours insérer des parenthèses pour changer les priorités :
"`(-x) ** (y + 2) * y == 0`" devient

$$((-x)^{y+2} \times y) = 0$$

La structure conditionnelle augmentée

Ajouter un bloc `else` ("sinon"), à exécuter seulement si la condition `CONDITION` était `False` :

```
if CONDITION:
    INSTRUCTION # si CONDITION est True
    ...
else:
    AUTRE_INSTRUCTION  # si CONDITION est False
    ...
CONTINUER_ICI          # en tout cas on reprend ici
```

Exemple :

```
i = int(input("Entrez un nombre entier:"))
if i % 2 == 0:
    print(i, "est pair")
else:
    print(i, "est impair")
```

La structure conditionnelle augmentée

Ajouter des blocs `elif` (“sinon, si”) :

```
if CONDITION1:
    INSTRUCTION # si CONDITION1 est True
    ...
elif CONDITION2:
    AUTRE_INSTRUCTION  # si CONDITION1 est False
    ...                #    mais CONDITION2 est True
elif CONDITION3:       # etc.
    ENCORE_AUTRE_INSTRUCTION
    ...
else:
    DERNIERE_CHANCE    # si toutes CONDITIONs sont False
    ...
...
```

Exercices

- Éliminer les parenthèses superflues dans les expressions suivantes :

```
(a + b) - (2 * c)
(-a) / (-(b + c))
(2 * x) / (y * z)
(x / y) % (-z)
(x + 3) * (n % p)
(x / (y % z) > 0) and (z < 0)
```

- Réaliser un script qui teste qu'une année (qui sera fournie au clavier par l'utilisateur) est bissextile. On rappelle que les années bissextiles reviennent tous les 4 ans, sauf les années séculaires, si celles-ci ne sont pas multiples de 400. Ainsi, 1900 n'était pas une année bissextile, alors que 2000 l'était.
- Réaliser un script qui lit trois nombres flottants a, b, c du clavier et qui affiche le nombre des solutions réelles x de l'équation $ax^2 + bx + c = 0$. (Attention aux cas spéciaux !)

La boucle `while`

La boucle `while` (“tant que”) sert à répéter les instructions d'un bloc en fonction d'une condition :

```
while CONDITION:
    FAIS_QUELQUE_CHOSE # bloc est repeté tant que
    ...                #   CONDITION demeure True
CONTINUER_ICI          # Après on arrive ici
...
```

Exemple :

```
i = 1
while i % 2 != 0:    # condition remplie si i est impair
    i = int(input("Entrez un nombre pair:"))
print(i, "/ 2 = ", i / 2)
```

La boucle `while`

Deuxième exemple : Conjecture de Collatz.

Suite (n_i) définie par un $n_0 \in \mathbb{N}$ et la règle de récurrence

$$n_{i+1} = \begin{cases} \frac{n}{2}\,, & n \text{ pair} \\ 3n+1\,, & n \text{ impair} \end{cases}$$

Conjecture : $\forall\ n_0\ \exists\ i\ :\ n_i = 1\,.$

Le programme suivant calcule le i minimal pour un n_0 fourni par l'utilisateur :

```
n = int(input("Entrez n0: "))
i = 0
while n != 1:   # on suppose que la conjecture est vraie
                # (sinon la boucle ne terminera jamais!)
    if n % 2 == 0:   # n pair:
        n /= 2       # remplacer n <- n/2
    else:            # n impair:
        n *= 3       # remplacer n <- 3n + 1
        n += 1
    i += 1
print("i =", i)
```

La boucle `for`

La boucle `for` sert à répéter les instructions d'un bloc une fois pour chaque élément d'une séquence :

```
for VAR in SEQUENCE:
    FAIRE_QUELQUE_CHOSE # bloc repeté pour tous VAR
    ...                 #
CONTINUER_ICI           # après on arrive ici
...
```

L'utilisation du mot clé `in` est différente dans ce contexte qu'avant.

Exemple :

```
somme = 0
for x in [2, 3, 5, 7, 11, 13, 17, 19]:
    print("On ajoute", x)
    somme = somme + x
print("La somme des nombres premiers < 20 est", somme)
```

La boucle `for`

La fonction `range()` retourne un n-uplet des nombres entiers :

- `range(y)` retourne `(0,1,2,..., y-1)`
- `range(x, y)` retourne `(x,x+1,x+2,..., y-1)`
- `range(x, y, s)` retourne `(x,x+s,x+2s,..., y-s)`

Application typique de `range()` dans une boucle `for` :

```
for x in range(ITER):
    FAIRE_QUELQUE_CHOSE  # bloc repeté ITER fois
    ...
```

Exemple :

```
print("Les carrés et les cubes des nombres entre 0 et 9:")
for x in range(10):
    print(x**2)
    print(x**3)
    print("\n")
```

La boucle `for`

Avec un str, une boucle `for` se répète pour tous caractères :

```
for caractere in "jeu":
    print(caractere + caractere)    #    "jj
                                    #     ee
                                    #     uu"
```

Boucles `for` imbriquées :

```
animaux = ["Poisson", "Tortue", "Cachalot"]
n = 0
for animal in animaux:
    for caractere in animal:
        if caractere == "o":
            n += 1
print("Le nombre des 'o' dans la liste est", n)
```

Commandes utiles pour les boucles

La commande `break` abandonne une boucle. Exemple :

```
while True:                 # toujours vrai
    i = int(input("Entrer un nombre pair: "))
    if i % 2 == 0:    # vrai si i est pair
        print(i, "/ 2 = ", i / 2)
        break
```

Après une boucle, la commande `else` marque un bloc à exécuter seulement si la boucle n'a pas été abandonnée avec `break` mais s'est terminée régulièrement. Exemple :

```
binaire = input("Entrez un nombre binaire (des 0 et 1): ")
somme = 0
for i in range(len(binaire)): # len(x) = longueur du str x
    bit = int(binaire[i])
    if  bit == 0 or bit == 1:
        somme += bit * 2**(len(binaire) - i - 1)
    else:
        print("Expression non valide")
        break
else:
   print("Ce nombre en notation décimale est", somme)
```

Commandes utiles pour les boucles

Dans une boucle, la commande `continue` saute les instructions restants et continue avec la prochaine itération

```
s = input("Entrer une phrase: ")
for caractere in s:
    if caractere == "e":
        continue  # sauter l'instruction suivante
    print(caractere) # écrire toutes les lettres sauf les 'e'
```

Exemple : Boucles et structures conditionnelles

Un parachutiste est en chute libre pendant 20 s. Après il ouvre son parachute et il descend à une vitesse constante de 2 m/s. On s'intéresse à sa position en fonction du temps.

```
g = 9.81    # accélération gravitationnelle en m/s^2
v = 2.0     # vitesse après ouverture du parachute en m/s

h0 = float(input("Hauteur initiale en m: "))

for t in range(0, 22, 2):     # on affiche h tous les 2 s
    h = h0 - 0.5 * g * t**2   # nouvelle hauteur
    if h <= 0:                # ça fait mal!
        break
    print("A t =", t, "s, la hauteur est de", h, "m.")
else:
    print("Le parachute s'ouvre.")
    while h > 0:
        print("A t = ", t, "s, la hauteur est de", h, "m.")
        t += 10 # on affiche la hauteur tous les 10 s
        h -= 10 * v

print("Atterrissage!")
```

Exercices

- Examiner et corriger le code `Ex10_Primes.py`, l'exécuter, et le modifier à volonté .
- Réaliser un script qui demande à l'utilisateur de déviner un nombre (qui sera entré par le clavier) et ne termine pas sans avoir reçu la bonne réponse, 42. Employer une boucle `while`.
- Réaliser un script qui lit un entier naturel n du clavier, teste que $n > 0$, et calcule $n!$ avec une boucle `for`
- Réaliser un script qui lit un nombre positif k du clavier, teste que $k > 0$, et écrit toutes les éléments f_i de la suite de Fibonacci qui satisfaient $f_i < k$. (Définition : $f_0 = 0$, $f_1 = 1$, et $f_{n+2} = f_{n+1} + f_n \; \forall \, n$)
- Réaliser un script qui lit une phrase `phrase` et une seule lettre `let` du clavier et qui affiche l'indice de la première occurrence de `let` dans `phrase`. Si `let` n'apparaît pas dans `phrase`, le programme terminera avec un `RuntimeError`.

Les exceptions

Un programme en exécution peut se trouver dans une situation imprévue, "exceptionnelle", qu'il faut detecter et réparer avant de continuer. On dit que le programme produit une exception. Par exemple, on pourrait essayer d'effectuer une division par une variable dont la valeur est 0 :

```
a, b = 42, 0
print(a / b) # bonne syntaxe, mais cause une exception
```

Python permet de traiter les situations exceptionnelles avec `try` et `except` :

```
a = 42
b = int(input("Entrez un nombre non nulle:"))
try:
    print("42 divisé par ce nombre est", a / b)
except ZeroDivisionError:
    print("On ne peut pas diviser par 0!")
```

Une exception non traitée va causer la terminaison du programme.

Les exceptions

La structure `try ... except` :

```
try:
    FAIS_QUELQUE_CHOSE
    ...
except EXCEPTION_1:
    TRAITER_EXCEPTION_1
    ...
except EXCEPTION_2:
    TRAITER_EXCEPTION_2
    ...
...
except:
    TRAITER_TOUTE_EXCEPTION_RESTANTE
    ...
```

Le bloc suivant la commande `try` est toujours exécuté. Si pendant l'exécution de ce bloc le programme rencontre une des exceptions `EXCEPTION_1` ou `EXCEPTION_2`, il saute au bloc `except` correspondant. Un block `except` sans spécification d'exception peut être ajouté pour traiter toute autre exception.

Les exceptions

Quelques exceptions pré-définies :

- `ZeroDivisionError` : division par zero
- `TypeError` : une fonction appellée avec un mauvais type d'argument

```
a = [1, 2]
int(a)   # TypeError: on ne peut pas convertir une
         # liste en int
```

- `ValueError` : une fonction appellée avec le bon type mais la mauvaise valeur d'un argument

```
a = "xyz"
int(a)   # ValueError: on peut convertir un str en
         # int seulement s'il se compose des chiffres
```

- `RuntimeError` : exception générique pendant l'exécution

Les exceptions

On peut forcer le programme à produire des exceptions avec la commande `raise`. Ce mécanisme est utile, par exemple, pour vérifier que les données fournies par l'utilisateur soient dans le bon format :

```
i = int(input("Entrez un nombre pair:"))
if i % 2 != 0:
    raise RuntimeError("Ce nombre est impair!")
```

Comme dans l'exemple, on peut inclure un message d'erreur.

Exercices

- Réaliser un script qui invite l'utilisateur à entrer un nombre non nulle par le clavier, et puis affiche son inverse. Si par contre l'entrée ne peut pas être convertit en `float`, ou est nulle, le programme affichera un message d'erreur spéficiant le problème, et répète la procédure.

Les structures de contrôle : Erreurs fréquentes

- Deux-points oublié après `if`, `while`, `for` etc.
- Pour l'indentation des blocs :
  Ne jamais mélanger les éspaces et les tabulatrices.
  Conseillé : eviter les tabulatrices, indentation 4 espaces par niveau
- L'opérateur d'affectation est =, l'opérateur de comparaison est ==
  Donc `a == b` est une expression logique (qui vaut `True` si les valeurs de `a` et `b` sont égales, et `False` sinon) mais `a = b` est une affectation qui attribue à `a` la valeur de `b`.

```
cont = int(input("Combien y a-t-il de continents?"))
if cont = 6:          # Erreur! Ici il faut utiliser ==
    print("C'est correct!")
```

- Boucles infinies : assurez-vous que vos boucles se terminent !

```
x_n, r, i = 0.5, 3.6, 1
while i < 100:
    print(x_n)
    x_n = r * x_n * (1 - x_n)
print("Ca y est!")    # On n'arrive jamais ici
```

Les fonctions

Dans ce chapitre

Python

- Les définitions de fonctions
- La commande `return`
- La portée des identifiants
- Les fonctions anonymes et la commande `lambda`
- Les modules et la bibliothèque standard
- Le module `math`

Généralités

- La récursivité
- Les fonctions d'ordre supérieur

Exemples des fonctions

On a déjà vu des exemples de fonctions intégrées dans Python : `print()`, `input()` et `range()`. En général une fonction

- peut accepter un ou plusieurs paramètre(s) dit argument(s)
- effectue une tâche en fonction de ces arguments
- retourne une valeur

Par exemple, comme nous avons vu, la fonction `range()` accepte d'un à trois arguments et retourne un *n*-uplet de nombres entiers.

Autres exemples :

fonction	argument(s)	tâche	valeur de retour
`print()`	un, d'un type quelconque	affichage sur l'écran	aucun
`input()`	un str	affiche son argument, attend saisie du clavier	le `str` entré par le clavier
`int()`	un nombre ou `str` qui peut être converti en `int`	convertit son argument en `int`	le résultat de la conversion

Nouvelles fonctions

Voici un exemple d'une définition d'une nouvelle fonction `cube()` :

```
def cube(x):          # un argument, nommé x
    return x ** 3     # retourne x au cube
```

Les instructions dans les définitions de fonction sont exécutées lorsque l'interpreteur tombe sur un appel de fonction :

```
a = cube(5) # appelle la fonction cube(),
            #  affecte la valeur de retour à a
print(a)    # affiche "125"
```

Définir une fonction

La syntaxe pour une définition d'une fonction est

```
def NOM_DE_FONCTION(ARG1, ARG2, ...):
    INSTRUCTION1
    INSTRUCTION2
    ...
```

- Pour les noms des fonctions, les mêmes règles que pour les autres identifiants s'appliquent.
- Une fonction peut accepter un nombre quelconque d'arguments `ARG1`, `ARG2` etc.
- Après avoir été définie, on l'appelle avec la commande
  `NOM_DE_FONCTION(VAL1, VAL2, ...)`
  où les arguments `ARG1`, `ARG2` etc. sont à substituer par `VAL1`, `VAL2` etc.
  La valeur de cette expression est la valeur de retour de la fonction.
- Le bloc qui suit la commande `def` contient les instructions à exécuter à chaque appel. Comme tous les blocs, il peut contenir des sous-blocs gérés par des structures de contrôle, et même des autres appels de fonctions.

Valeurs par défaut des arguments

Il est possible de spécifier des valeurs par défaut pour quelques arguments :

```
def NOM_DE_FONCTION(ARG1=DEF1, ARG2=DEF2,...):
    ...
```

Si une valeur par défaut est spécifiée dans la définition d'une fonction, il est facultatif d'inclure l'argument correspondant lors d'un appel de fonction.

Exemple : Calculer une approximation de la fonction zêta de Riemann,

$$\zeta(z) = \lim_{N \to \infty} \sum_{k=1}^{N} \frac{1}{k^z}$$

```
def zeta(z, N=100):
  resultat = 0.
  for k in range(1, N+1): # k entre 1 et N inclus
    resultat += 1/k**z  # ajouter le k-ème terme à la somme
  return resultat
```

Possibles appels pour calculer $\zeta(2)$: `zeta(2)` ou `zeta(2, 1000)` ou `zeta(2, N=500)`

Valeurs par défaut des arguments

Au cas de plusieurs valeurs par défaut, les arguments non nommés doivent toujours précédér les arguments nommés pour éviter toute ambiguïté.

Exemple :

```
# x n'a pas de valeur par défaut. z=1 et y=1 par défaut.
def multiplier(x, y=1, z=1):
    return x * y * z
```

Exemples d'appels de cette fonction :

- `multiplier(25, 5, 2)`
- `multiplier(-2, 16)` (en utilisant la valeur par défaut du dernier argument z)
- `multiplier(3)` (dans ce cas les valeurs par défaut pour z et y sont utilisées)
- `multiplier(17, z=5)` (ce qui pose $x = 17$, $z = 5$, et $y = 1$ sa valeur par défaut)

Par contre, `multiplier(z=5, 17)` est un appel invalide (un argument nommé ne peut précéder un argument non nommé)

La commande `return`

La commande `return` peut figurer quelque part dans la définition d'une fonction. Dès que l'interpréteur la rencontre, il abandonne la fonction et continue au lieu où elle a été appellée. L'argument de `return` est renvoyé et devient la valeur de l'expression d'appel de fonction.

```
def heaviside(r):
    if r >= 0:
        return 1.0
    else:
        return 0.0

print("Theta(1) =", heaviside(1))
print("Theta(-1) =", heaviside(-1))
```

Si une fonction n'est pas terminée par un `return`, ou si `return` est rencontré sans aucun argument, la fonction renvoie l'objet abstrait `None` ("aucun").

Les fonctions

Exercices

- Réaliser une fonction `ld()` à un argument (un entier $n > 0$) qui retourne le seul entier p positif ou nul tel que $2^p \leq n < 2^{p+1}$
- Réaliser une fonction `backwards()` à un argument du type str. Elle retournera son argument à l'envers :
  `backwards("Monty Python")` donnera `"nohtyP ytnoM"`
  (Se servir de la fonction `len()` qui retourne la longueur d'un str)
- Réaliser une fonction `zeta()` à deux arguments `n` et `z`, où `n` est un nombre naturel (valeur défaut 100), et `z` est un nombre complexe avec partie réelle > 1.
  Si $n \leq 0$ ou $\mathrm{Re}(z) \leq 1$, la fonction produira un `ValueError`.
  Autrement la fonction retournera une approximation de la fonction zêta de Riemann au point z, $\zeta(z) = \sum_{k=0}^{\infty} \frac{1}{k^z}$, dont elle calcule seulement les `n` premiers termes.
- Eprouver ces fonctions avec des scripts Python

Parenthèse : Coding style

Récommandations pour créer du code plus lisible :

- Adopter des conventions cohérents et les suivre partout.
- Ne pas économiser des commentaires.
- Une instruction par ligne. Eviter les point-virgules ou des lignes comme

```
if CONDITION: FAIRE_QC    # légal mais mauvais style
```

- 4 espaces par niveau d'indentation. Pas de tabulatrice.
- Pas d'espace juste après des parenthèses, crochets et accolades ([{
  ni juste avant)]}
  ni juste avant des virgules, point-virgules et deux-points
- une seule espace après , ; :
- une seule espace à chaque coté des opérateurs d'affectation et de comparaison
- Désignations parlantes pour les variables, fonctions et méthodes importantes.
  Employer des minuscules et éventuellement des chiffres et tirets bas _
- Employer des majuscules pour les classes (voir chapître correspondant).

Deuxième parenthèse : La portée des identifiants

La portée (lexicale) d'un nom de variable est la portion du code où la variable peut être adressée par ce nom. Par défaut, pour les affectations à l'intérieur d'une définition de fonction, la portée des noms des variables n'est que cette définition de fonction. De ce fait, le code

```
def f():
    x = 0    # définit variable x dans la portée de f()
f()
print(x)     # erreur: x pas défini dans cette portée
```

produit une erreur. Par contre, si une variable est définie hors d'une définition de fonction, on peut néanmoins l'utiliser là-dedans :

```
x = 0
def f():
    print(x)    # variable x définie hors de f()
f()             # mais pas de problème
```

La portée des identifiants

On peut définir une variable, dans la portée locale d'une fonction, avec le le même nom qu'une variable déjà définie hors de la fonction. Cela crée une nouvelle variable, et dans la portée locale, le nom se réfère toujours à cette nouvelle variable locale. Exemple :

```
x = 0       # définit variable globale x
def f():
    x = 1   # définit variable locale x
    print(x)    # "1" (priorité de la variable locale)
f()
print(x)    # "0" (hors portée de la variable locale)
```

Des affectations aux variables globales dans une définition de fonction sont toutefois possibles (mais déconseillées si évitable), avec la commande `global` :

```
x = 0
def f():
    global x # x référera à la variable globale:
    x = 1     # ne crée pas une nouvelle variable locale
f()
print(x)      # "1"
```

Récursivité

Une fonction peut appeler elle-même :

```
def factorielle(n): # calcule n! recursivement
    if n < 0:       # factorielle pas définie
        raise ValueError("Factorielle pas définie!")
    if n == 0:      # 0! = 1
        return 1
    return n * factorielle(n - 1)   # n! = n (n-1)!
```

On parle de la récursivité. Attention à la terminaison des algorithmes qui s'en servent !

Fonctions comme arguments

On peut passer des fonctions comme arguments aux autres fonctions :

```
def iterer(f, debut, n_fois):  # f(f(...f(debut)))
    resultat = debut
    for n in range(n_fois):
        resultat = f(resultat)
    return resultat

def logistique(x, r = 3.6):
    return r * x * (1 - x)

print(iterer(logstique, 0.5, 100)) #   0.43172
```

Dans l'appel de `iterer()` le nom de la fonction `f` (`logistique` en ce cas) est traité comme un nom d'une variable.

Fonctions comme valeurs de retour

Une fonction peut renvoyer une autre :

```
def racine(n): # renvoie la fonction "n-ème racine"
    def f(x):
        return x**(1/n)
    return f

print(racine(5)(32))    # affiche "2.0"
```

Une fonction qui soit prend une autre fonction comme argument soit renvoye une fonction est dite une fonction d'ordre supérieur. Les fonctions d'ordre supérieur sont particulièrement importantes dans la programmation fonctionnelle.

Exercices

Algorithme d'Euclide pour trouver le PGCD (plus grand commun diviseur) :
Soient x et y des nombres entiers avec $x \geq y > 0$; soit r le reste de la division entière de x par y. Alors le PGCD de x et y est égal au PGCD de y et r. (Par définition, le PGCD de x et 0 est x.)

Exercices

- Réaliser une fonction `pgcd()` à deux arguments, des nombres naturels `x`, `y`, qui calcule leur PGCD récursivement par l'algorithme d'Euclide.

Dérivée d'une fonction : Soit $f : \mathbb{R} \to \mathbb{R}$ dérivable et soit $x \in \mathbb{R}$. La dérivée de f au point x est la limite

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- Réaliser une fonction `deriv()` à trois arguments `f`, `x`, `epsilon`, où `x` et `epsilon` sont des `float` et `f` est une application dérivable $f : \mathbb{R} \to \mathbb{R}$. Les valeurs défaut de `x` et `epsilon` sont 0 et 10^{-5}. La valeur de retour sera la dérivée de `f` au point `x` avec une précision numérique de $\mathcal{O}(\texttt{epsilon})$.
- Eprouver ces fonctions avec des scripts Python

Les fonctions anonymes avec `lambda`

La commande `lambda` permet des très courtes definitions des fonctions dans une seule ligne. En lieu de

```
def carre(x):
    return x ** 2
```

on écrit

```
carre = lambda x: x ** 2
```

(exception de la règle d'une instruction par ligne).
Plus généralement,

```
lambda ARGUMENTS: EXPRESSION
```

définit une fonction avec arguments `ARGUMENTS` qui retourne `EXPRESSION`.

Exemple d'une fonction anonyme

Retournons à l'exemple de la fonction `iterer()` :

```
def iterer(f, debut, n_fois):  # f(f(...f(debut)))
    resultat = debut
    for n in range(n_fois):
        resultat = f(resultat)
    return resultat
```

Au lieu de

```
def logistique(x):
    return 3.6 * x * (1 - x)
print(iterer(logistique, 0.5, 100)) #   0.43172
```

on peut utiliser `lambda` comme suit :

```
print(iterer(lambda x: 3.6 * x * (1 - x), 0.5, 100))
```

Les modules

Avec Python on peut réemployer dans un nouveau projet son vieux code avec la commande `import`. Par exemple, si on enregistre les définitions de fonctions

```
# fichier puissance.py
def carre(x):
    return x ** 2
cube = lambda x: x ** 3
```

dans un fichier `puissance.py`, on peut les utiliser ailleurs :

```
# fichier nouveauprojet.py
import puissance
print(puissance.carre(42))
```

sans devoir inclure tout le code explicitement. Pour seulement importer `carre()`,

```
from puissance import carre
print(carre(42))    # ici: pas 'puissance.carre(42)'
```

La bibliothèque standard

Python est fourni avec un grande bibliothèque de fonctions pré-definies pour toutes sortes de tâches. Entre autres il y a des modules pour

- manipuler les chaînes de caractères
- manipuler les tableaux de données
- les fonctions mathématiques
- les nombres rationnelles
- accéder et manipuler les fichiers
- les interfaces aux bases de données
- la compression et la sauvegarde des données
- services cryptographiques
- interactions avec le système d'exploitation
- les services de réseau
- les services internet et les pages web
- multimédia
- les interfaces graphiques

La bibliothèque standard : le module `math`

Fonctions et constantes utiles du module `math` de la bibliothèque standard :

```
import math

# Les constantes e et pi
math.e                  # 2.71828...
math.pi                 # 3.14159...

# Les fonctions trigonométriques
math.sin(1.2)
math.cos(math.pi)
math.tan(0)

# Les fonctions trigonométriques inverses
math.asin(1/2)
math.acos(0.5)
math.atan(-1)
```

La bibliothèque standard : le module `math`

Fonctions et constantes utiles du module `math` de la bibliothèque standard :

```
import math

# Les fonctions exponentielle et logarithme
math.exp(-3.0)
math.log(1.0)      # logarithme naturel
math.log(4, 2)     # logarithme de base 2

# La racine carrée
math.sqrt(2.0)          # équivalent: 2**(1/2)
```

- Toutes ces fonctions prennent des arguments du type `float` (ou des arguments `int`, que Python convertit automatiquement en `float`).
- Si on veut les appliquer aux nombres complexes, il faut importer le module `cmath` en lieu de `math`.

D'autres exemples de modules de la bibliothèque standard

Le module `time` met à disposition des fonctions pour accéder à l'horloge interne de l'ordinateur. Exemples :

```
import time

# Retourner un str représentant la date et l'horaire
# présente
time.asctime()

# Retourner un float qui représente le nombre de secondes
# depuis 1 janvier 1970 0:00:00 UTC
time.time()

# Arrêter l'exécution du programme pendant 7.5 secondes
time.sleep(7.5)
```

D'autres exemples de modules de la bibliothèque standard

Le module `random` contient des fonctions pour générer des nombres (pseudo-)aléatoires. Exemples :

```
import random

# Retourner un nombre pseudoaléatoire entre 0 et 1
# avec une distribution uniforme
random.random()

# Retourner un élément aléatoire d'une liste
L = [1, 19, 23, 47]
random.choice(L)

# Retourner un entier aléatoire entre a (inclu)
# et b (exclu) avec une distribution uniforme
a, b = 10, 20
random.randint(a, b)
```

Exercices

- Réaliser une fonction `gauss()` à trois arguments `x`, `sigma`, `mu` du type float. Elle retournera la valeur de la fonction de Gauss au point x,

$$\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- Réaliser une fonction `rotate()` à deux arguments : un angle `theta` représenté par un float et un vecteur à deux composantes, representé par une liste de deux float. Elle retournera le vecteur tourné par l'angle θ autour de l'origine,

$$\begin{pmatrix} v_1' \\ v_2' \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$$

- Eprouver ces fonctions avec des scripts Python

Exercice : Programmation procédurale (application)

Réaliser un programme qui permet à l'utilisateur de jouer le jeu aux cartes de blackjack. (Pour un aperçu des règles simplifiées, voir la fiche d'exercices.)

Les fonctions : Erreurs fréquentes

- Deux-points oubliés après la commande def
- Il faut qu'une fonction soit définie avant l'appel.

```
x, y = ma_fonction() # erreur: ma_fonction
                     # pas encore definie ici!

def ma_fonction():   # définition trop tardive
    a = int(input("Entrez un nombre entier:"))
    return a // 5, a % 5
```

- Pour les fonctions importées :
  attention à la différence entre import et from ... import :

```
from math import sqrt
x = sqrt(10)          # pas math.sqrt()
import cmath
y = cmath.exp(1.0J) # pas exp()
```

Application I : Zéros des fonctions non linéaires

Dans ce chapitre

Algorithmes

- La méthode de bisection
- La méthode de relaxation
- La méthode de Newton
- Les généralisations de la méthode de Newton

Zéros des fonctions

Problème : Etant donné une fonction réelle sur l'intervalle $I = [a, b]$ qui a un zéro unique sur I ; où précisement se trouve ce zéro ?

Au sens plus strict :

Soit $I = [a, b]$ un intervalle et $f : I \to \mathbb{R}$ continue (ou même dérivable si necessaire) et monotone sur I, avec $f(a)f(b) \leq 0$. Alors d'après le théorème des valeurs intermédiaires il existe $x_0 \in I$ tel que $f(x_0) = 0$. Cet x_0 est unique a cause de la monotonie.
Problème : déterminer x_0 numériquement.

Théorème des valeurs intermédiares :
Soit $f : [a, b] \to \mathbb{R}$ continue et soit y compris entre $f(a)$ et $f(b)$. Alors il existe $x \in [a, b]$ tel que $f(x) = y$.

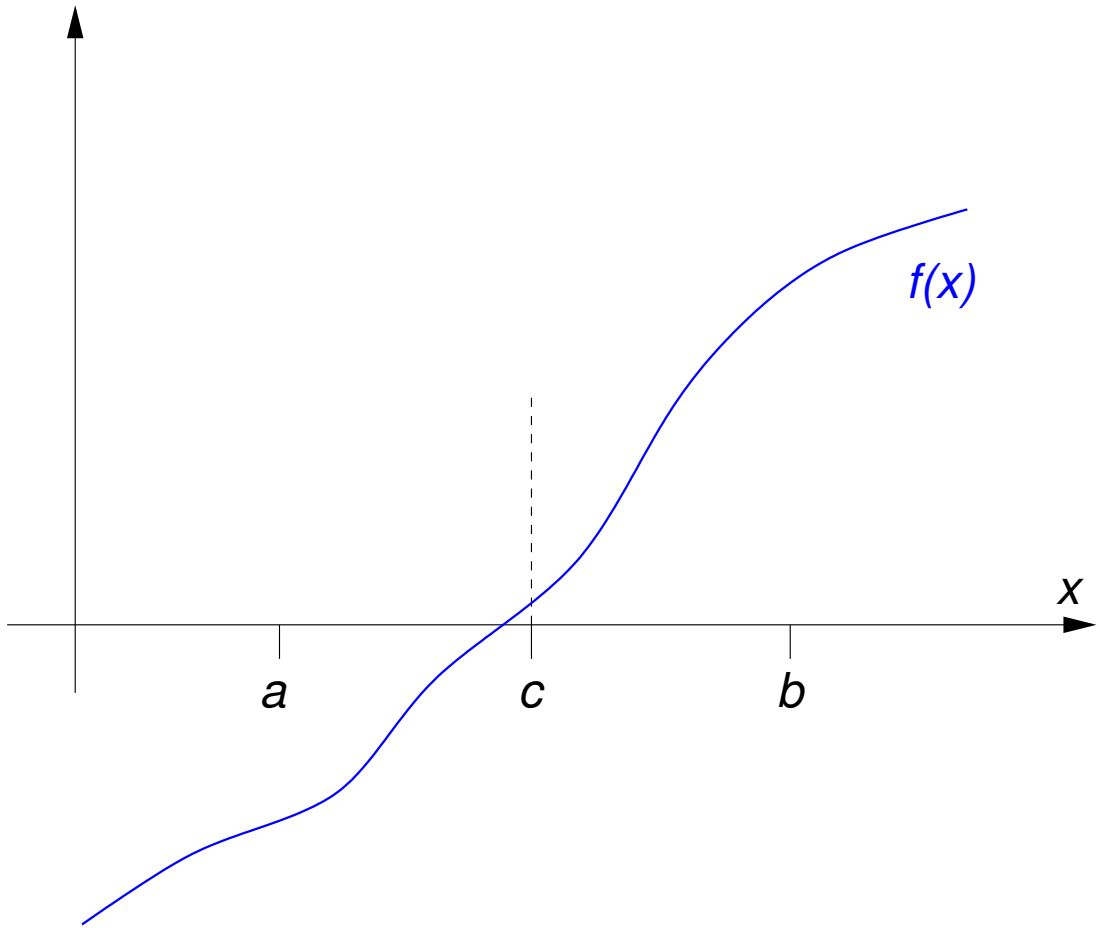
Méthode de bisection

Soit $\epsilon > 0$ l'inexactitude tolérée (on sera content de déterminer x_0 avec une précision $\pm\epsilon$).

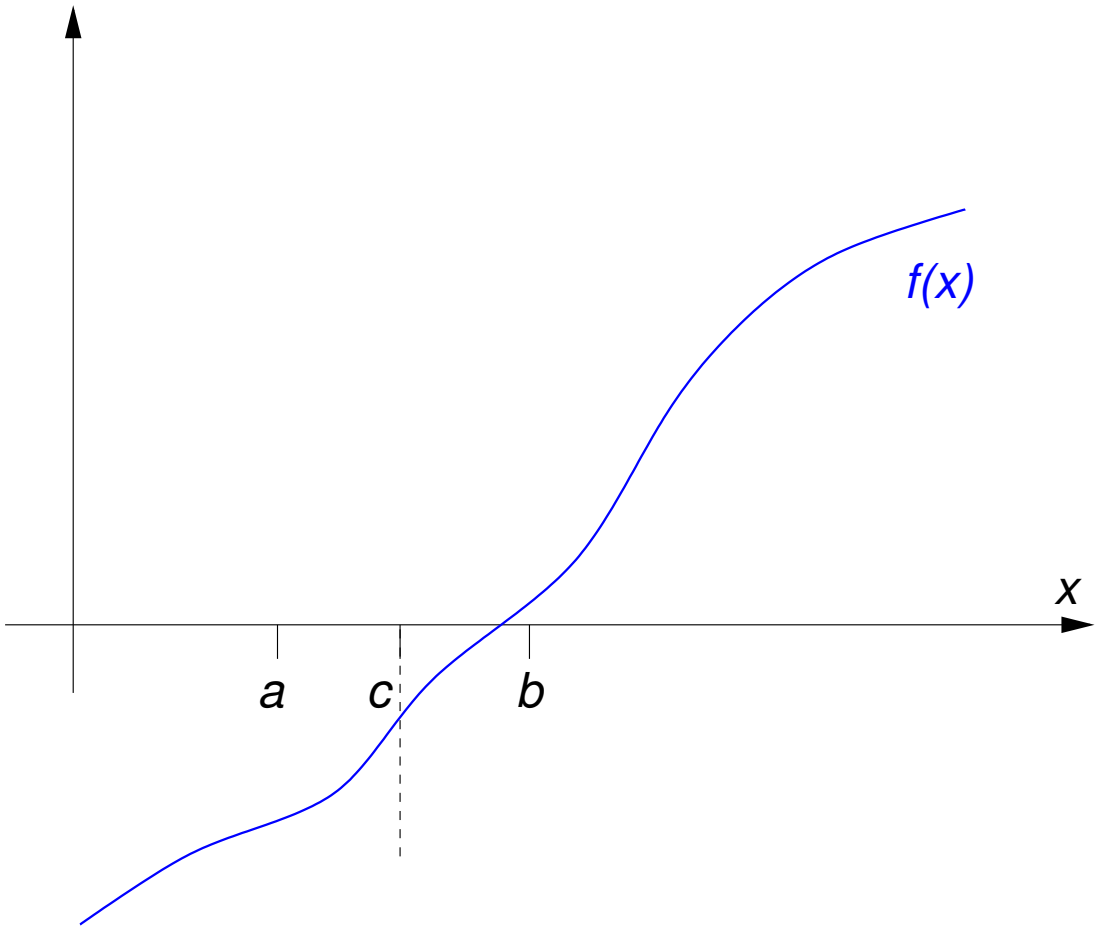
Algorithme :

1. Poser $c = (a + b)/2$, le milieu de l'intervalle I.
2. Si $b - a < 2\,\epsilon$: terminer et retourner $x_0 = c$.
3. Partager I en deux : $I_1 = [a, c]$ et $I_2 = [c, b]$.
4. Si $f(a)f(c) \leq 0$: Répeter avec $I = I_1$.
5. Si par contre $f(a)f(c) > 0$, alors $f(c)f(b) \leq 0$ d'après le theorème des valeurs intermédiaires. Répeter donc avec $I = I_2$.

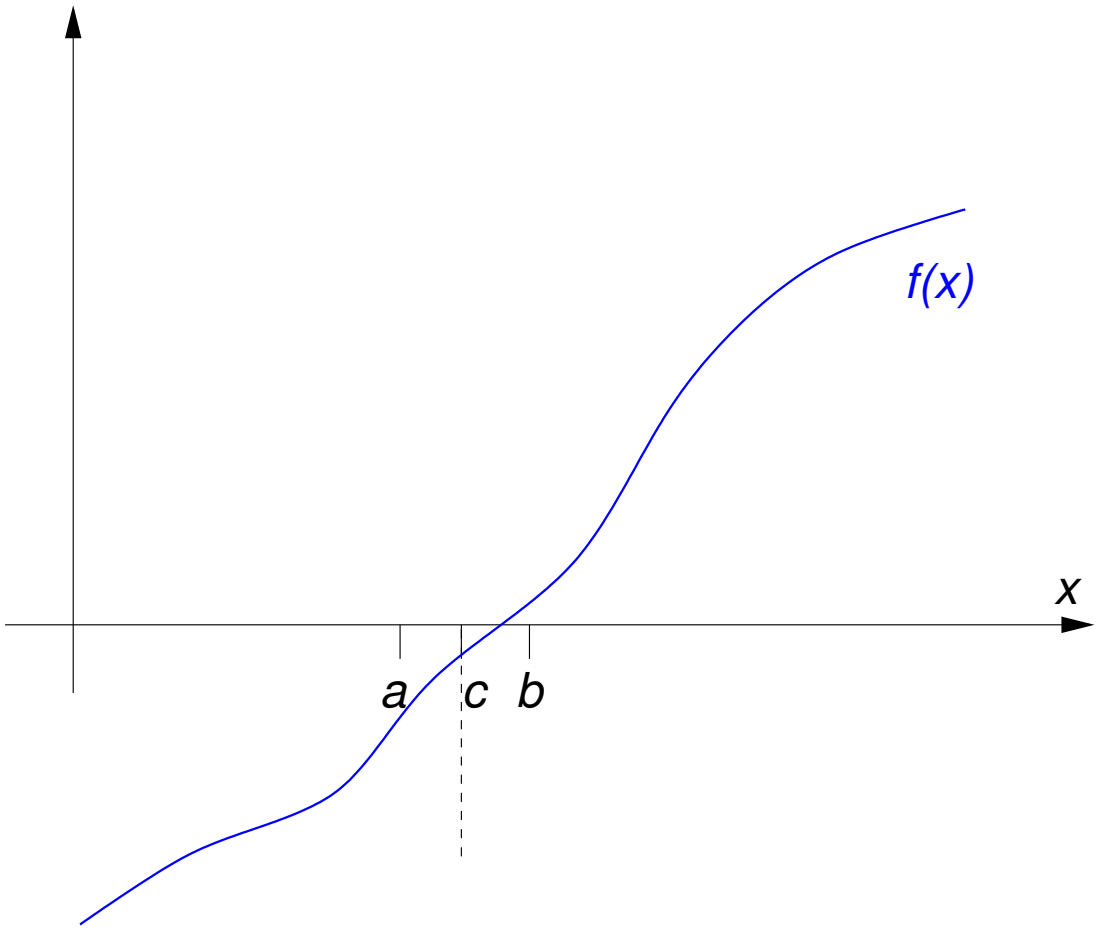
Méthode de bisection



Méthode de bisection



Méthode de bisection



Implementation en Python

Voici le code Python pour $f(x) = x^5 - x - 1$, $I = [1, 2]$, $\epsilon = 10^{-5}$:

```
def f(x):
    return x**5 - x - 1

a, b = 1.0, 2.0     # intervalle
c = (a + b) / 2     # point au milieu
epsilon = 1.0E-5    # tolérance

while b - a >= 2 * epsilon:
    if f(a) * f(c) <= 0:
        b = c
    else:
        a = c
    c = (a + b) / 2

print("Le zéro est à x =", c)
```

Exercices

Modifier et améliorer ce script (`Ex20_Bisection.py`) :

- Expérimenter avec d'autres fonctions f (se servir du module `math`)
- Laisser l'utilisateur entrer `a` et `b` par le clavier, et puis vérifier que les données entrées sont raisonnables.
- Laisser l'utilisateur entrer la fonction `f` par le clavier. Se servir de la fonction `eval()` : Cette fonction prend un `str` comme argument, interprète son contenu comme une instruction en Python, et l'évalue.
- Au lieu de choisir, pour découper l'intervalle, son milieu $c = (a + b)/2$: Construire une fonction linéaire sur $[a, b]$ dont les valeurs à a et b coïncident avec f, et poser $c =$ son zéro ("méthode regula falsi").
- Pour cette dernière méthode, introduire une variable `maxiter`. Après `maxiter` itérations, faire terminer le code en affichant un message "Aucune solution trouvée après [`maxiter`] itérations".

Méthode de relaxation

Idée :

- Pour résoudre l'équation $f(x) = 0$, trouver une équation équivalente

$$\phi(x) = x$$

(il y a une infinité de choix pour ϕ — le succès de la méthode va dépendre du choix).

- On cherche alors un point fixe x^* de la fonction ϕ.
- Essayer de trouver un point fixe par l'application repétée de la fonction ϕ sur un x_1 :

$$\begin{aligned} x_2 &= \phi(x_1) \\ x_3 &= \phi(x_2) = \phi(\phi(x_1)) \\ x_4 &= \phi(x_3) = \phi(\phi(\phi(x_1))) \\ &\dots \\ \lim_{n \to \infty} x_n &\stackrel{?}{=} x^* \end{aligned}$$

Méthode de relaxation

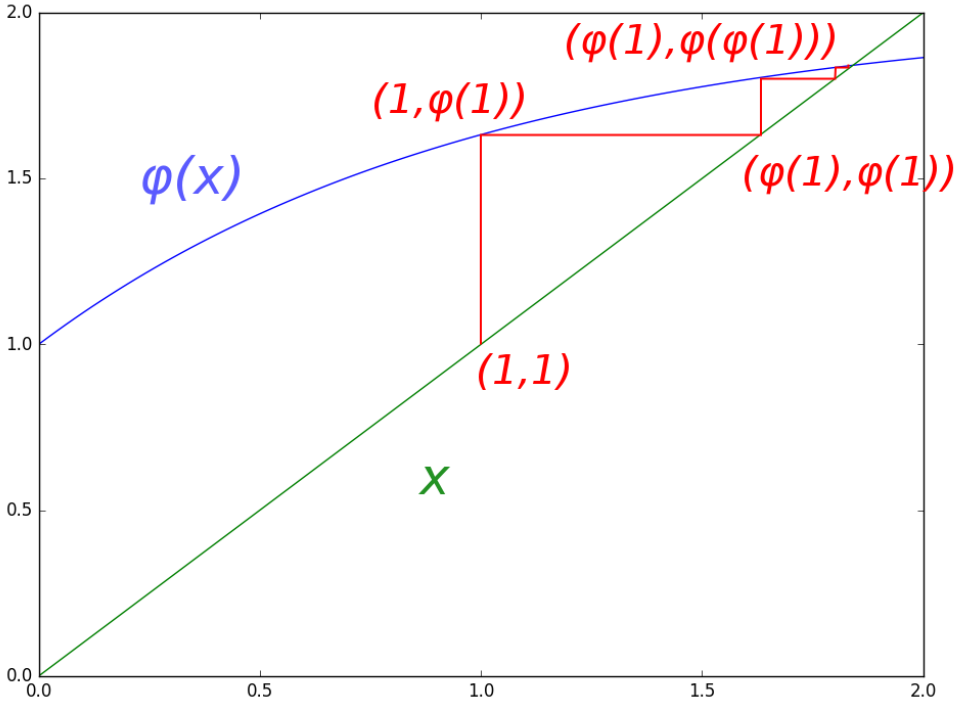
Exemple :

- On cherche un zéro x^* de la fonction $f(x) = 2e^x - xe^x - 1$.
- Équivalent : on cherche un point fixe x^* de la fonction $\phi(x) = 2 - e^{-x}$.
- Avec $x_1 = 1$ on trouve

$$
\begin{aligned}
x_2 &= \phi(x_1) = 1.63212 \\
x_3 &= \phi(x_2) = 1.80448 \\
&\dots \\
x_9 &= \phi(x_8) = 1.84141 \\
x_{10} &= \phi(x_9) = 1.84141
\end{aligned}
$$

Conclusion : Pour $x^* \approx 1.84141$ on a $\phi(x^*) = x^*$, alors $2 - e^{-x^*} = x^*$,
alors $2e^{x^*} - x^* e^{x^*} - 1 = f(x^*) = 0$.

Méthode de relaxation : Illustration



Condition suffisante pour la convergence : $|\phi'(x)| < 1$ partout
ou bien : ϕ est une contraction.

Explications mathématiques

Théorème du point fixe :
Soit $I = [a, b]$ un intervalle, $0 \leq k < 1$, et $\phi : I \to I$ continue tel que $|\phi(x) - \phi(y)| \leq k|x - y| \quad \forall\, x, y \in I$ (on dit que ϕ est une contraction).
Alors il existe un point fixe unique $x^* \in I$.
De plus, si (x_n) est une suite dans I qui vérifie $x_{n+1} = \phi(x_n)$, alors (x_n) converge vers x^*.

Démonstration :
Soit $x_1 \in I$ quelconque et $x_{n>1}$ défini par récurrence comme $x_{n+1} = \phi(x_n)$. On a $|x_{n+1} - x_n| \leq k^{n-1}|x_2 - x_1|$, donc

$$\begin{aligned} |x_{n+m} - x_n| \leq & |x_{n+1} - x_n| + |x_{n+2} - x_{n+1}| + \ldots + |x_{n+m} - x_{n+m-1}| \\ & \leq \left(k^{n-1} + k^n + \ldots + k^{n+m-1}\right) |x_2 - x_1| \\ & = k^{n-1} \frac{1 - k^m}{1 - k} |x_2 - x_1| \end{aligned}$$

Comme $k < 1$, l'expression dans la dernière ligne tend vers zero quand $n \to \infty$, alors les (x_n) forment une suite de Cauchy qui converge vers un x^*. On a $\phi(x^*) = x^*$ à cause de la continuité.

Méthode de Newton

Cas spécial : méthode de Newton

L'idée de la méthode de Newton est de linéariser f autour d'un point x_1, de trouver le zéro de la fonction tangente g ainsi définie, et d'itérer :

- Posons alors
$$g(x) = f(x_1) + f'(x_1)(x - x_1)$$
(premiers deux termes dans le développement limité de f en x_1, donc $g(x) = f(x) + \mathcal{O}(|x - x_1|^2)$)
- Le zéro de $g(x)$ est à $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$.
- On réitère l'opération tant que les deux valeurs consécutives x_i, x_{i+1} diffèrent par plus que ϵ.

D'après le théorème du point fixe la suite des x_i converge vers un point fixe x^* de la fonction auxiliare $\phi : x \mapsto x - \frac{f(x)}{f'(x)}$ (si cette dernière est une contraction).

Or $\phi(x^*) = x^*$, alors $\frac{f(x^*)}{f'(x^*)} = 0$, alors $f(x^*) = 0$.

Méthode de Newton

Zéro de la fonction f = point fixe de la fonction auxiliaire

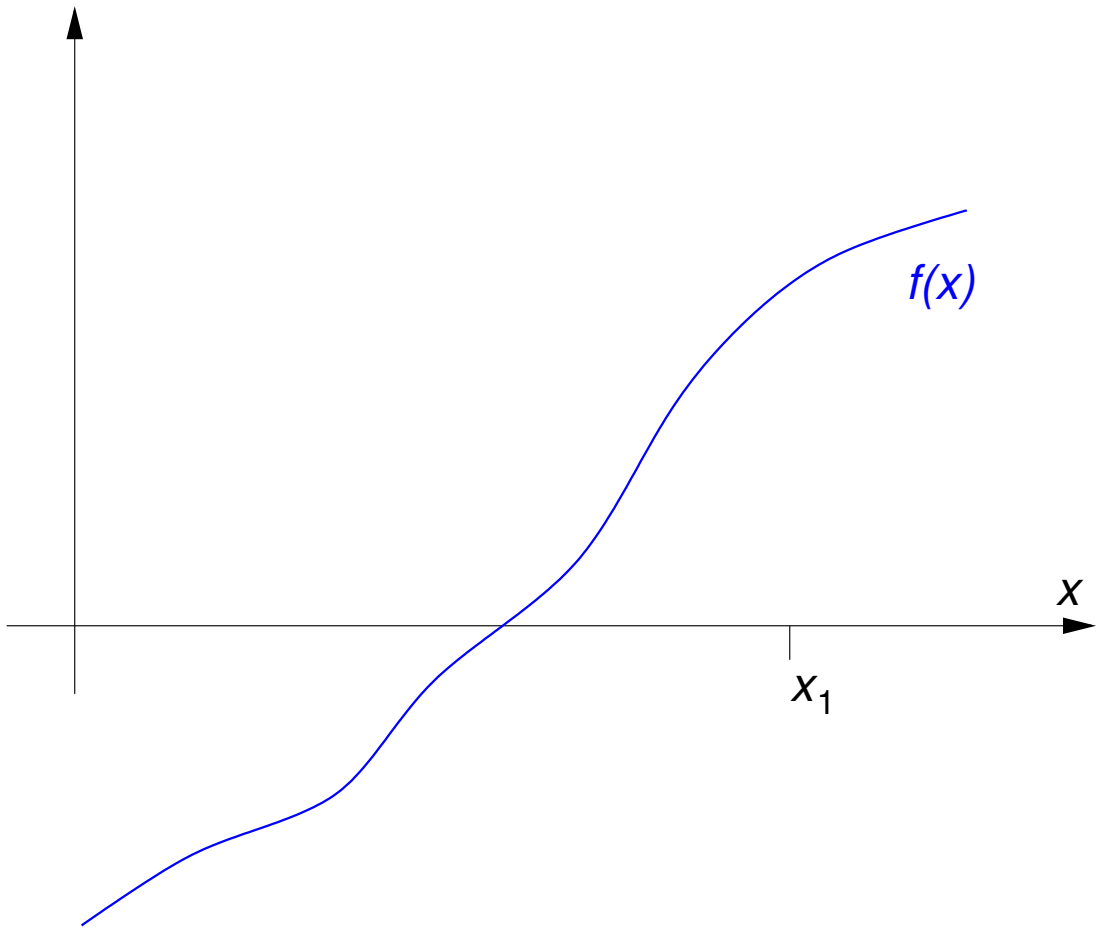
$$\phi : x \mapsto x - \frac{f(x)}{f'(x)}$$

Remarques :

- Si f est deux fois dérivable et $f'(x) \neq 0 \; \forall \, x \in I$, alors $\phi'(x) = \frac{f(x)f''(x)}{f'(x)^2}$.
- Dans ce cas : ϕ est une contraction $\Leftrightarrow \exists \, k < 1$ avec $|\phi'(x)| \leq k$, soit $\left| \frac{f(x)f''(x)}{f'(x)^2} \right| \leq k$
  ("$\Leftarrow$" vient du théorème des accroissements finis, "$\Rightarrow$" de la définition de la dérivée)
- Pour que l'algorithme converge : Il est suffisant mais pas nécessaire que ϕ soit une contraction.

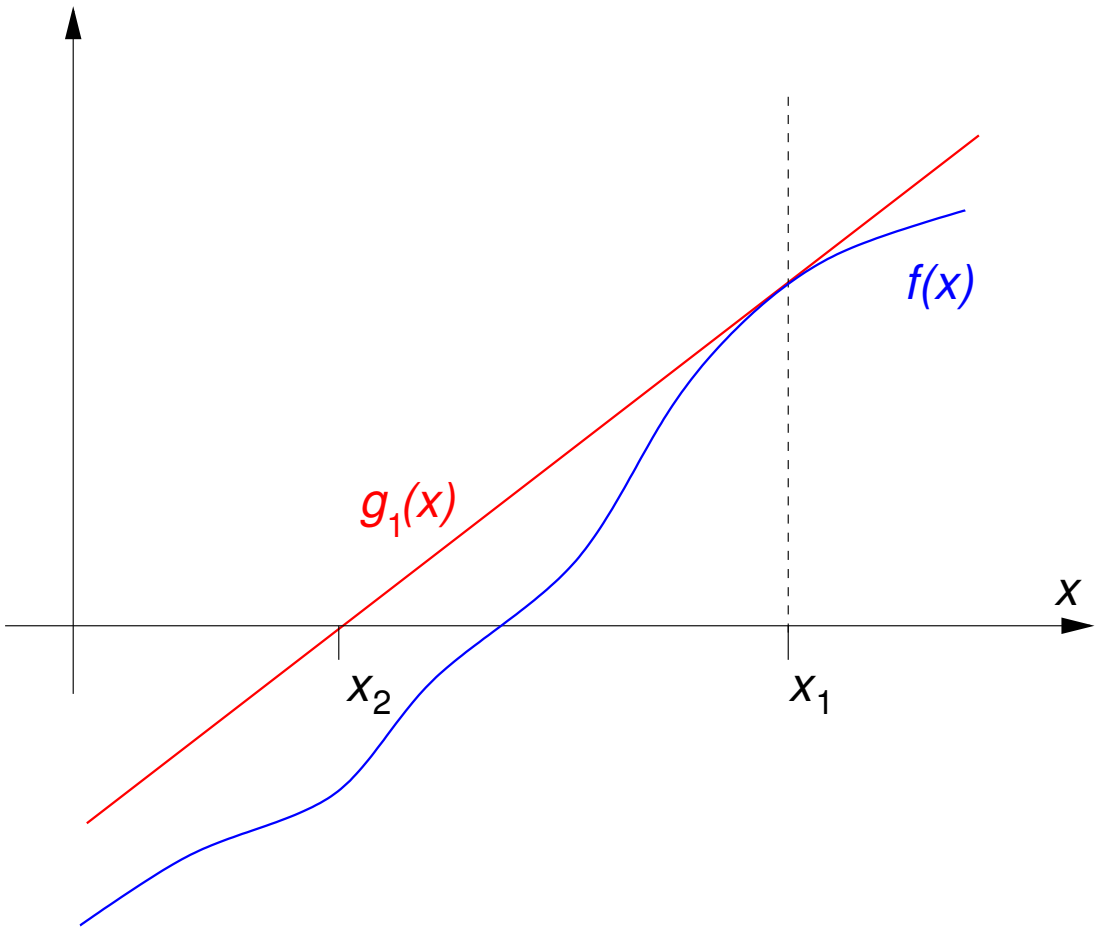
  Etablir un critère suffisant et nécessaire peut être très difficile voir impossible, voir exercices sur le cas complexe.

Méthode de Newton



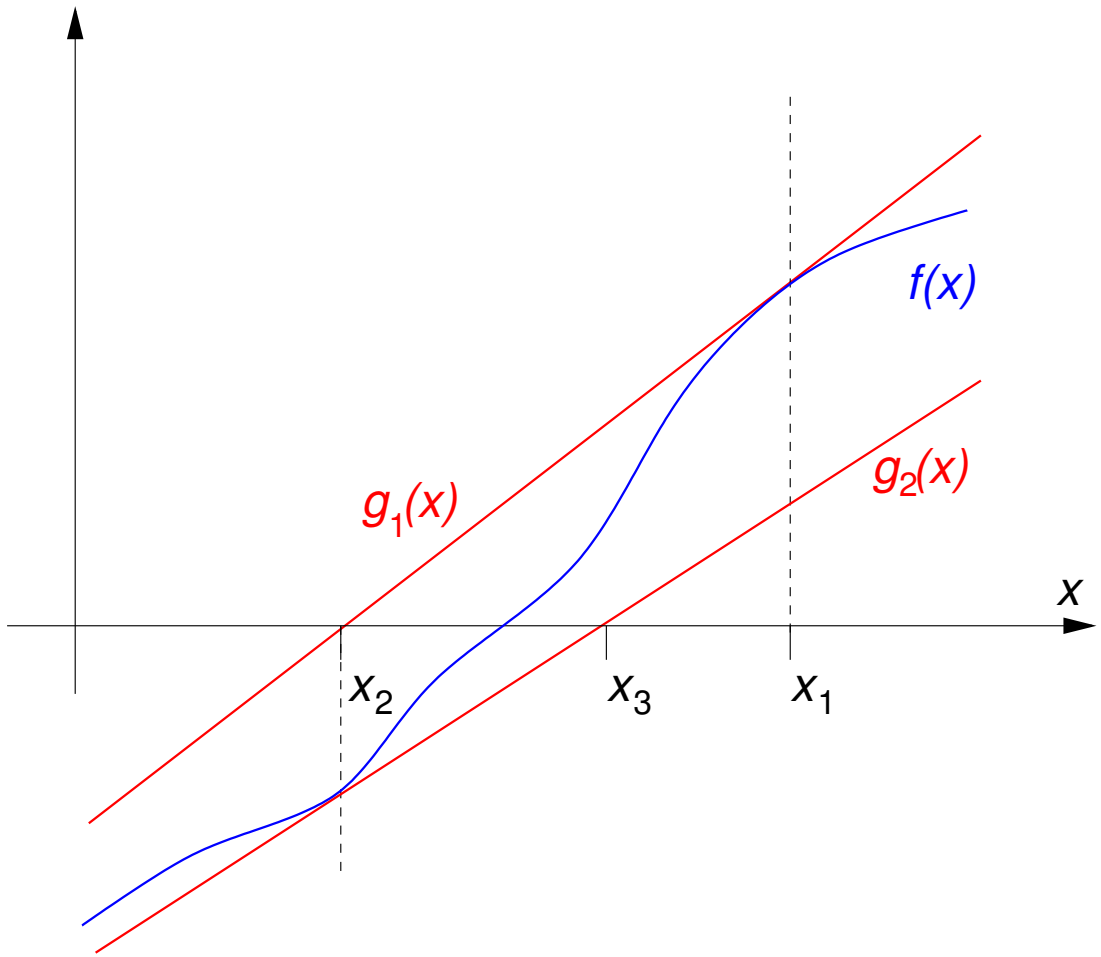
$$g_i(x) = f(x_i) + f'(x_i)(x - x_i)$$

Méthode de Newton



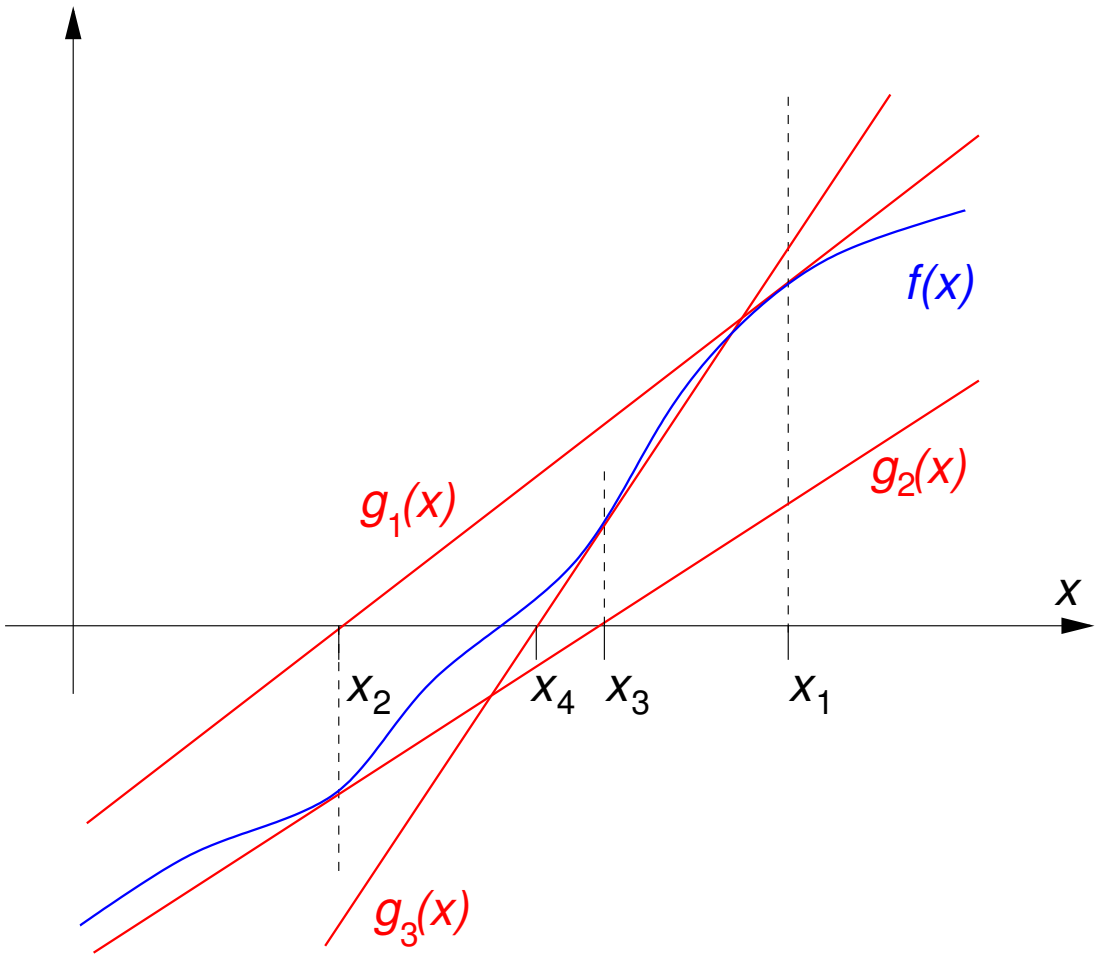
$$g_i(x) = f(x_i) + f'(x_i)(x - x_i)$$

Méthode de Newton



$$g_i(x) = f(x_i) + f'(x_i)(x - x_i)$$

Méthode de Newton



$$g_i(x) = f(x_i) + f'(x_i)(x - x_i)$$

Méthode de Newton

Algorithme :

1. Partir avec x au choix. Calculer $f'(x)$.
2. Remplacer $x_{\text{anc}} \leftarrow x, \qquad x \leftarrow x - \frac{f(x)}{f'(x)}$.
3. Si $|x - x_{\text{anc}}| < \epsilon$, terminer et retourner x. Sinon, répéter.

Exercices

- Implémenter la méthode de Newton. (Se servir de la fonction `deriv()` déjà réalisée, et de la fonction pré-definie `abs()` qui renvoie la valeur absolue de son argument.)
- Appliquer alors votre code à la fonction $f : I \to \mathbb{R}$, $f(x) = x^5 - x - 1$, avec $I = [1\,, 2]$ et avec une précision numérique de $\epsilon = 10^{-5}$. La convergence est-elle garantie par le theorème du point fixe dans ce cas, pour toutes les valeurs initiales $x \in I$?
- Assurez-vous numériquement que la méthode de Newton ne converge pas pour la fonction $f(x) = x^3 - 2\,x + 2$ avec la valeur initiale $x_1 = 1$. Puis, montrez-le analytiquement.

Comparaison des méthodes

La table suivante montre l'erreur absolu après n itérations pour le zéro de $x^5 - x - 1$ (avec le choix $\phi(x) = (x+1)^{1/5}$ pour la méthode de relaxation) :

Itérations	Bisection	Relaxation	Newton
1	-8.27e-02	-7.84e-02	-4.66e-01
2	4.23e-02	-8.33e-03	-2.06e-01
3	-2.02e-02	-8.96e-04	-5.63e-02
4	1.11e-02	-9.65e-05	-5.43e-03
5	-4.57e-03	-1.04e-05	-5.59e-05
6	3.24e-03	-1.12e-06	-6.01e-09

On voit que la méthode de Newton a besoin de beaucoup moins d'itérations que les deux autres. Le gain du temps n'est pas signifiant pour cet exemple, car la méthode de Newton effectue plus d'opérations arithmétiques par itération (calcul numérique de la dérivée !), et surtout parce que notre implémentation n'est pas optimisée. Voici le temps en secondes requis sur mon ordinateur pour atteindre une précision fixe :

Precision	Bisection	Relaxation	Newton
1.00e+00	1.27e-06	6.60e-07	1.76e-06
1.00e-05	1.45e-05	2.19e-06	9.34e-06
1.00e-10	2.90e-05	3.93e-06	1.09e-05
1.00e-15	4.25e-05	5.01e-06	1.04e-05

Généralisations de la méthode de Newton

- La méthode peut être appliquée sans modifications pour des fonctions complexes. Les domaines de convergence vers les zéros constituent des structures mathématiques très intéressantes : les “fractales de Newton” $\rightarrow$ Exercices.
- Méthode de Halley : basée sur l'itération

$$x_{n+1} = x_n - \frac{2\, f(x_n)\, f'(x_n)}{2\, f'(x_n)^2 - f(x_n)\, f''(x_n)}$$

  pour des fonctions au moins deux fois dérivables. Converge plus rapidement que la méthode de Newton, mais nécéssite l'évaluation de la deuxième dérivée.
- Méthodes de Householder : Soit f k-fois dérivable avec des dérivées continues. On itère

$$x_{n+1} = x_n + k \left. \frac{\frac{d^{k-1}}{dx^{k-1}} \left(\frac{1}{f(x)} \right)}{\frac{d^k}{dx^k} \left(\frac{1}{f(x)} \right)} \right|_{x=x_n}$$

  Pour $k = 1$ on retrouve la méthode de Newton, pour $k = 2$ celle de Halley.

Exercices

On va analyser la convergence de la méthode de Newton numériquement, dans le cas où la fonction f est un polynôme en une variable complexe z, $f(z) = z^5 - z - 1$.

- Calculer à la main la dérivée $f'(z)$ de $f(z)$.
- Réaliser une fonction `newton_conv(x, y)` à deux arguments du type `float`, $\mathtt{x} = \mathrm{Re}\, z_1$ et $\mathtt{y} = \mathrm{Im}\, z_1$. Cette fonction retournera le nombre d'itérations pour converger vers une racine quelconque de $f(z)$ avec une précision de $\epsilon = 10^{-3}$, ou 20 si après 20 itérations aucune racine a été trouvée.
- Visualiser la fonction `newton_conv(x, y)` pour 500 valeurs de $\mathrm{Re}\, z_1 = \mathtt{x}$ entre -10 et $+10$, et 500 valeurs de $\mathrm{Im}\, z_1 = \mathtt{y}$ entre -10 et $+10$. Utiliser la fonction `plot.plot()` du fichier `Exemples/plot.py`.
- Trouver les valeurs numériques des racines de $f(z)$ avec une préc. de 10^{-5}.
- Réaliser une fonction `which_root(x, y)` à deux arguments du type `float`, $\mathtt{x} = \mathrm{Re}\, z_1$ et $\mathtt{y} = \mathrm{Im}\, z_1$. Cette fonction retournera 1, 2, 3, 4 ou 5 selon la racine vers laquelle la méthode de Newton converge, en fonction de la valeur initiale z_1. Si après 20 itérations elle n'a pas trouvé une des racines, elle retournera 0.
- Visualiser la fonction `which_root(x, y)` pour 500 valeurs de $\mathrm{Re}\, z_1 = \mathtt{x}$ entre -10 et $+10$, et 500 valeurs de $\mathrm{Im}\, z_1 = \mathtt{y}$ entre -10 et $+10$.

Application II : Trier une liste

Dans ce chapitre

Python

- Les opérations sur les listes

Algorithmes

- Le tri par insertion
- Le tri rapide
- Le tri fusion

Généralités

- La complexité algorithmique

Opérations sur les listes

Rappel : on peut indicer les listes avec `[]`, et les couper avec `[:]`

```
liste = list(range(13, 18))
print(liste)                    # "[13, 14, 15, 16, 17]"
print(liste[0], liste[2])       # "13 15"
print(liste[3:])                # "[16, 17]"
```

On peut aussi les indicer à l'envers :

```
print(liste[-1],  liste[-3])        # "17 15"
```

Contrairement aux chaînes de caractères (immuables), les éléments des listes peuvent être modifiées :

```
liste[0] = 12
print(liste)      # "[12, 14, 15, 16, 17]"
liste[1:3] = []
print(liste)      # "[12, 16, 17]"
```

Les opérations sur les listes

La fonction `len()` retourne la longueur d'une liste (ou d'un `str` ou `tuple`) :

```
L = list(range(13, 18))
print(L)          # "[13, 14, 15, 16, 17]"
print(len(L))     # "5"
```

Avec l'opérateur + on peut concaténer des `list` comme on fait pour les `str`. De plus, certains fonctions (dites méthodes) sont associées aux objets du type `list`. Pour les appeller, préfixer le nom de la liste :

```
L.append(x)       # ajoute l'élément x à la liste L
L1.extend(L2)     # ajoute les éléments de L1 à L2 (comme +=)
L.insert(i, x)    # insère un nouvel élément x à la position i
L.remove(x)       # efface le premier element de valeur x
L.pop(i)          # efface l'élément à la position i
                  #   et renvoie sa valeur
```

Trier une liste

Des routines pour le tri des listes sont déjà fournies dans Python. On va pourtant étudier ce problème classique de l'algorithmique, très instructif pour comprendre certains aspects de l'analyse de complexité :

Soit $L = (x_1, x_2, \ldots x_n)$ un n-uplet de nombres naturels.
On cherche un n-uplet $L' = (x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)})$ tel que $x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)}$, où $\pi \in S_n$ est une permutation.

Exemple : Soit $L = (3, 7, 1, 5, 9, 5)$ la liste originale, alors $L' = (1, 3, 5, 5, 7, 9)$ est la liste triée.

Evidemment un algorithme de tri ne va pas se limiter aux nombres naturels, mais pourra facilement être adapté à toute suite finie pour laquelle un ordre total peut être défini (p.ex. des nombres réels, chaînes de caractères. . .)

Trier une liste

Pour tester nos algorithmes nous nous servirons des listes des entiers aléatoires. On les génère avec la fonction `randrange()` du module `random`. L'appel `randrange(RANGE)` retourne un nombre entier aléatoire entre 0 et `RANGE`−1 inclus.

```
from random import randrange
n = 20
liste = [randrange(100) for i in range(n)]
        # 20 entiers aléatoires entre 0 et 99
```

A noter la syntaxe `[EXPRESSION for X in LIST]` pour créer la liste, une nouvelle application des mots clé `for...in`.

Le tri par insertion

Cet algorithme est très facile à implémenter mais plutôt inefficace :

1. Supposons que les premiers k éléments de la liste sont déjà dans le bon ordre (au début, $k = 1$).
2. Insérer le $(k+1)$-ème élément à sa propre position par rapport aux premiers k éléments.
3. Les premiers $k+1$ éléments sont désormais dans le bon ordre. Itérer.

Exemple : Soit `[15, 4, 2, 8, 23, 17, 0, 11]` une liste à trier.

```
[15, 4, 2, 8, 23, 17, 0, 11]
[4, 15, 2, 8, 23, 17, 0, 11]
[2, 4, 15, 8, 23, 17, 0, 11]
[2, 4, 8, 15, 23, 17, 0, 11]
[2, 4, 8, 15, 23, 17, 0, 11]
[2, 4, 8, 15, 17, 23, 0, 11]
[0, 2, 4, 8, 15, 17, 23, 11]
[0, 2, 4, 8, 11, 15, 17, 23]
```

Tri par insertion

Le code utilise une double boucle `for` :

- La boucle sur `k` parcourt toute la liste.
- La boucle sur `i` parcourt les premiers `k` éléments en cherchant la bonne position du (`k`+1)-ème élément `x` = `L[k]` parmi eux.

```
def tri_insertion(L):
  for k in range(1, len(L)): # premiers k él. déja triés
    x = L[k]                 # valeur du (k+1)-ème élément
    for i in range(k):       # parcourir les premiers k él.
      if x <= L[i]:      # faut-il mettre L[k] devant L[i]?
        L.pop(k)         # si oui, l'effacer...
        L.insert(i, x)   # ...le reinsérer à position i...
        break            # ...et quitter la boucle sur i
```

Parenthèse : Analyse de la complexité des algorithmes

Exemples des problèmes numériques :

- Trier une liste de n éléments
- Résoudre une équation avec une précision de n décimales
- Évaluer une fonction à une précision de 1 sur n
- Diagonaliser une matrice $n \times n$
- Trouver la position d'un élément particulier dans une liste de n éléments

Mesure de la performance d'un algorithme :
Comment le temps de calcul $T(n)$ depend-il de la taille des données d'entrée/de sortie n ?

(Autres mesures selon la situation : besoin de mémoire $M(n)$ ou de largeur de bande du réseau $B(n)\ldots$)

On est intéressé surtout par le comportement asymptotique de $T(n)$ pour des grands n.

Complexité en temps.

Analyse de la complexité : Exemple simple

Un algorithme pour trouver le maximum dans une liste `L` :

```
def maximum(L):
    m = L[0]
    for x in L:          # parcourir toute la liste
        if x > m:        # si élément x > ancien maximum:
            m = x        #  alors x devient le nouveau maximum
    return m
```

- La boucle `for` parcourt toute la liste et effectue une ou deux opérations simples (une comparaison, potentiellement une affectation) à chaque itération
- Si la liste a n éléments $\Rightarrow n \leq$ (opérations simples) $\leq 2n$.
- Chaque opération simple prend un temps constant (indépendant de n).
- Asymptotiquement quand $n \to \infty$, $T(n)$ est alors bornée par une fonction linéaire de n. On écrit $T(n) = \mathcal{O}(n)$ et on parle de la complexité linéaire.
- Rappel : $f(n) = \mathcal{O}(g(n)) \overset{\text{def.}}{\Leftrightarrow} \limsup_n \left| \frac{f(n)}{g(n)} \right| < \infty$

Analyse de la complexité : Bisection

Rappel de l'algorithme de la bisection pour trouver un zéro d'une fonction f :

- Partir avec un intervalle $[a, b]$ dans lequel f a un zéro.
- Poser $c = \frac{a+b}{2}$ le milieu de l'intervalle.
- Si le zéro est dans $[a, c]$, itérer avec $b \leftarrow c$. Sinon, itérer avec $a \leftarrow c$.
- Terminer lorsque $b - a < 2\epsilon$ avec ϵ la précision désirée.

Analyse de complexité : Supposons que on cherche le zéro dans l'intervalle $[0, 1]$ avec n chiffres significatifs, alors $\epsilon = 10^{-n}$.

- Dans chaque itération on effectue un nombre constant d'opérations simples : comparaisons, affectations, évaluations de la fonction f...
- Pour chaque itération l'intervalle est reduit par la moitié.
- Pour augmenter la précision par un facteur 10 il faut donc $\log_2(10) \approx 3.32$ itérations en moyenne
- Pour atteindre une précision de 10^{-n} il faut alors $n \log_2(10)$ itérations
  $\Rightarrow$ (cte.) $\times$ n opérations simples

$$T(n) = \mathcal{O}(n), \qquad \text{complexité linéaire}$$

Analyse de la complexité

- Quelques autres possibilités que la complexité linéaire :
  - complexité constante : $T(n) = \mathcal{O}(1)$
    Le nombre d'opérations simples nécessaires ne dépend pas de n
  - complexité logarithmique : $T(n) = \mathcal{O}(\log n)$
    $T(n) \sim \text{const} \cdot \log n$ quand $n \to \infty$, croissance lente avec n.
  - complexité polynomielle : $T(n) = \mathcal{O}(n^p)$
    $T(n) \sim \text{const} \cdot n^p$ ($n \to \infty$). En redoublant n, T augmentera par un facteur 2^p.
  - complexité exponentielle : $T(n) = \mathcal{O}(e^n)$
    Croissance très rapide avec n.
- On rappelle encore : $f(n) = \mathcal{O}(g(n)) \overset{\text{def.}}{\Leftrightarrow} \limsup_n \left| \frac{f(n)}{g(n)} \right| < \infty$
- Pour certains algorithmes $T(n)$ peut varier en fonction des données.
  Par exemple, pour le tri d'une liste déjà triée / triée à l'envers / aléatoire, le besoin de temps $T(n)$ peut être différent (selon l'algorithme)
- Dans ce cas il faut distinguer la complexité dans le meilleur cas, le pire cas et la complexité en moyenne.

Analyse de la complexité

Exemple hypothétique : Supposons qu'un algorithme a besoin de $T(10) = 10\,\mu$s pour effectuer une tâche caractérisée par $n = 10$. Alors en fonction de sa complexité ça va prendre, pour $n > 10$:

	$n = 10$	$n = 20$	$n = 30$	$n = 100$	$n = 1000$	$n = 10\,000$
$\mathcal{O}(1)$	10 μs	10 μs	10 μs	10 μs	10 μs	10 μs
$\mathcal{O}(\log n)$	10 μs	13 μs	15 μs	20 μs	30 μs	40 μs
$\mathcal{O}(\sqrt{n})$	10 μs	14 μs	17 μs	32 μs	100 μs	320 μs
$\mathcal{O}(n)$	10 μs	20 μs	30 μs	100 μs	1 ms	10 ms
$\mathcal{O}(n^2)$	10 μs	40 μs	90 μs	1 ms	100 ms	10 s
$\mathcal{O}(n^3)$	10 μs	80 μs	270 μs	10 ms	10 s	3 h
$\mathcal{O}(e^n)$	10 μs	220 ms	1.5 h	10^{26} ans*	10^{417} ans*	10^{4326} ans*

(* Âge de l'univers : 10^{10} ans)

Analyse de la complexité

Exercice

Analyser la complexité du test de primalité par des essais successives de division. Le code est

```
def est_premier(n):
    k = 2
    while k**2 <= n:
        if n % k == 0:
            return False
        k += 1
    return True
```

Complexité des algorithmes de tri

Pour analyser la complexité des algorithmes de tri :

On compte le nombre d'opérations simples en fonction de la longueur de la liste n

Voici encore le code du tri par insertion :

```
def tri_insertion(L):
  for k in range(1, len(L)): # (n-1) itérations
    x = L[k]                 # opération simple
    for i in range(k):       # entre 1 et n-1 itérations
      if x <= L[i]:      # opération simple
        L.pop(k)         # opération simple
        L.insert(i, x)   # opération simple
        break            # opération simple
```

Complexité du tri par insertion

Deux boucles imbriquées parcourant la liste `L` de longueur n =`len(L)` :

```
def tri_insertion(L):
    for k in range(1, len(L)):
        ...
```

- $(n-1)$ itérations de la boucle extérieure

```
    ...
    for i in range(k):
        ...
```

- Pire cas : Liste déjà triée, `k` itérations sans tomber sur le `break` $\Rightarrow (n-1)/2$ itérations en moyenne
- En moyenne : Liste aléatoire, `k`$/2$ itérations en moyenne jusqu'au `break` à chaque passage $\Rightarrow (n-1)/4$ itérations en moyenne
- Meilleur cas : Liste triée à l'envers, 1 itération et `break` immédiat

Complexité du tri par insertion

Résumé : $(n-1)$ itérations de la boucle sur `k`. A chaque itération, $\frac{n-1}{2}$ ou $\frac{n-1}{4}$ ou 1 itérations en moyenne de la boucle sur `i`, et un nombre constant d'opérations simples.

- Pire cas : $T(n) \sim (n-1)(n-1)/2 = \mathcal{O}(n^2)$, complexité quadratique
- En moyenne : $T(n) \sim (n-1)(n-1)/4 = \mathcal{O}(n^2)$, complexité quadratique
- Meilleur cas : $T(n) \sim (n-1) = \mathcal{O}(n)$, complexité linéaire

(On peut facilement adapter l'algorithme de la sorte que le "meilleur cas" est une liste déjà triée et le "pire cas" est une liste triée à l'envers.)

Le tri par insertion est cher en matière de temps de calcul (complexité quadratique = inefficace) sur des grandes listes sans ordre particulier. Néanmoins, il est souvent utilisé pour trier des listes courtes.

Tri rapide

L'algorithme du tri rapide (quicksort) est plutôt efficace sur les grandes listes, et très répandu.

Algorithme :

1. Sélectionner un élément au choix dans la liste (le pivot)
2. Placer tous les éléments supérieurs au pivot à sa droite, et tous les éléments inférieurs à sa gauche. Le pivot est maintenant à la position correcte.
3. Appliquer recursivement à la sous-liste à gauche et à la sous-liste à droite du pivot.

Tri rapide

Suggestions pour l'implémentation :

- On réalisera trois fonctions `echanger(L, i, j)`, `partitionner(L, g, d)` et `tri_rapide(L, g, d)`.
- La fonction `echanger(L, i, j)` échangera les éléments de `L` aux positions `i` et `j`.
- La fonction `partitionner(L, g, d)` prendra trois arguments : une liste `L` ainsi que deux indices `g`, `d` qui délimitent la sous-liste `L[g:d]`. Son objectif sera de mettre un élément (le pivot) de cette sous-liste au bon endroit par rapport aux autres, et de renvoyer sa nouvelle position.
  - On choisira un pivot, disons le premier élément de la sous-liste, `L[g]`.
  - Ensuite, la fonction parcourira la sous-liste en mettant tous les éléments qui sont inférieurs au pivot à la gauche (pour cela il faudra deux indices, le premier pour parcourir la sous-liste et le deuxième pour marquer la position ultérieure du pivot).
  - Enfin, elle placera le pivot au bon endroit et retournera sa nouvelle position.
  - Elle se servira de la fonction `echanger()`.
- La fonction `tri_rapide(L, g, d)` divisera la sous-liste `L[g:d]` en deux avec l'aide de la fonction `partitionner()`, et ensuite appliquera soi-même recursivement aux deux parties.

Tri rapide

Exercices

- Dérouler à la main l'algorithme du tri rapide sur le tableau `[15, 4, 2, 8, 23, 17, 0, 11]` .
- Implémenter le tri rapide en Python et l'appliquer sur ce tableau.

Toujours choisir le premier élément du segment comme pivot est inefficace si la liste est déjà (partiellement) triée, situation fréquemment rencontrée dans les applications réalistes. Il vaut mieux choisir un élément au hasard.

Exercices

- Montrer que la complexité du tri rapide dans le pire cas (pivot = premier élément, liste déjà triée) est quadratique.
- Modifier votre code de la sorte que le pivot est choisi par hasard en utilisant la fonction `random.randrange()`.

On peut montrer que la complexité moyenne est seulement $T(n) = \mathcal{O}(n \log n)$.

Tri fusion

Le tri rapide est efficient parce qu'il se sert du principe *diviser et regner* : il partage le tableau à trier en sous-tableaux et s'applique recursivement à eux.

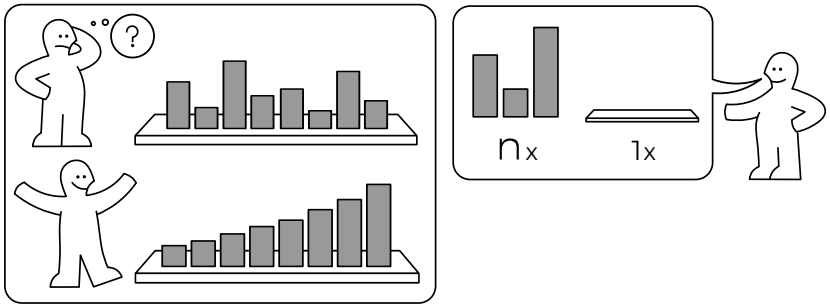
Un autre algorithme qui utilise ce principe est le tri fusion (merge sort). Il repose sur le fait qu'il est très économique de "fusionner" deux listes qui sont déjà triées en une seule liste triée.

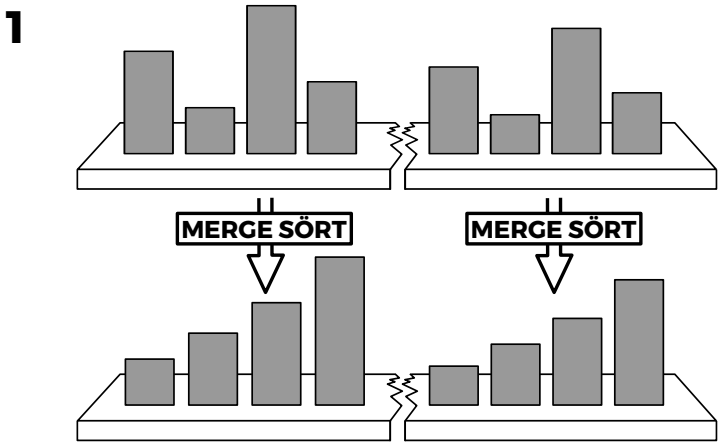
Algorithme pour trier une liste de longueur n :

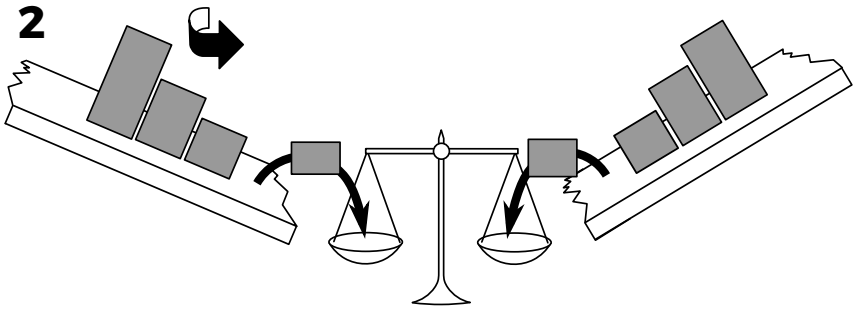
1. Si $n \leq 1$, il n'y a rien à faire.
2. Si $n > 1$, diviser la liste à trier en deux sous-listes de longueur $n/2$ (si n est pair) ou $(n+1)/2$ et $(n-1)/2$ (si n est impair). Recursivement appliquer l'algorithme aux deux moitiés.
3. Fusionner les deux moitiés, qui sont maintenant triées.

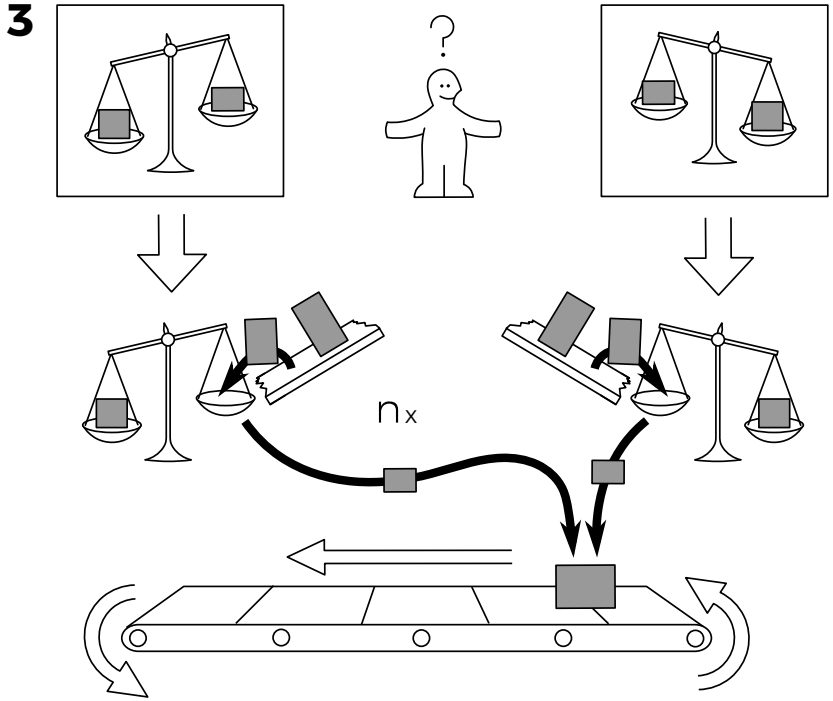
MERGE SÖRT

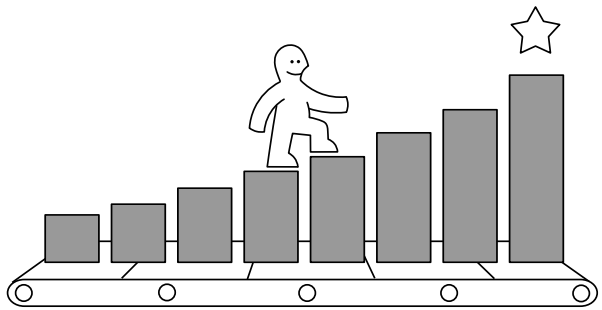
idea-instructions.com/merge-sort/











Tri fusion

Fonction pour trier la sous-liste `L[g:d]` par le tri fusion (ou bien toute la liste si `g` et `d` ne sont pas spécifiés) :

```
def tri_fusion(L, g=0, d=None):
    if d is None:  # si d pas donné, d -> dernier élément
        d = len(L)
    n = d - g      # longueur de la sous-liste à trier
    if n <= 1:     # L[g:d] vide ou contient un seul élément
        return     #    (rien à faire dans ce cas)
    m = g + n // 2       # m = milieu de la liste
    tri_fusion(L, g, m)  # recursion: trier L[g:m]...
    tri_fusion(L, m, d)  # ...et L[m:d]
    fusionner(L, g, m, d)   # fusionner les sous-listes
```

La fonction `fusionner()` va fusionner les sous-listes `L[g:m]` et `L[m:d]`.

Tri fusion

Algorithme pour fusionner deux sous-listes avec la fonction `fusionner()` :

- Créer une nouvelle liste `F` qui deviendra la liste fusionnée
- Parcourir les deux sous-listes avec deux indices `gi` et `di`.
  - Si `L[gi]` $\leq$ `L[di]` : Ajouter à `F` l'élément `L[gi]` et augmenter `gi`.
  - Si `L[gi]` $>$ `L[di]` : Pareil pour `di`.
- Lorsque une des deux sous-listes s'épuise, ajouter le reste de l'autre à `F`.
- Remplacer `L[g:d]` par `F`.

Tri fusion

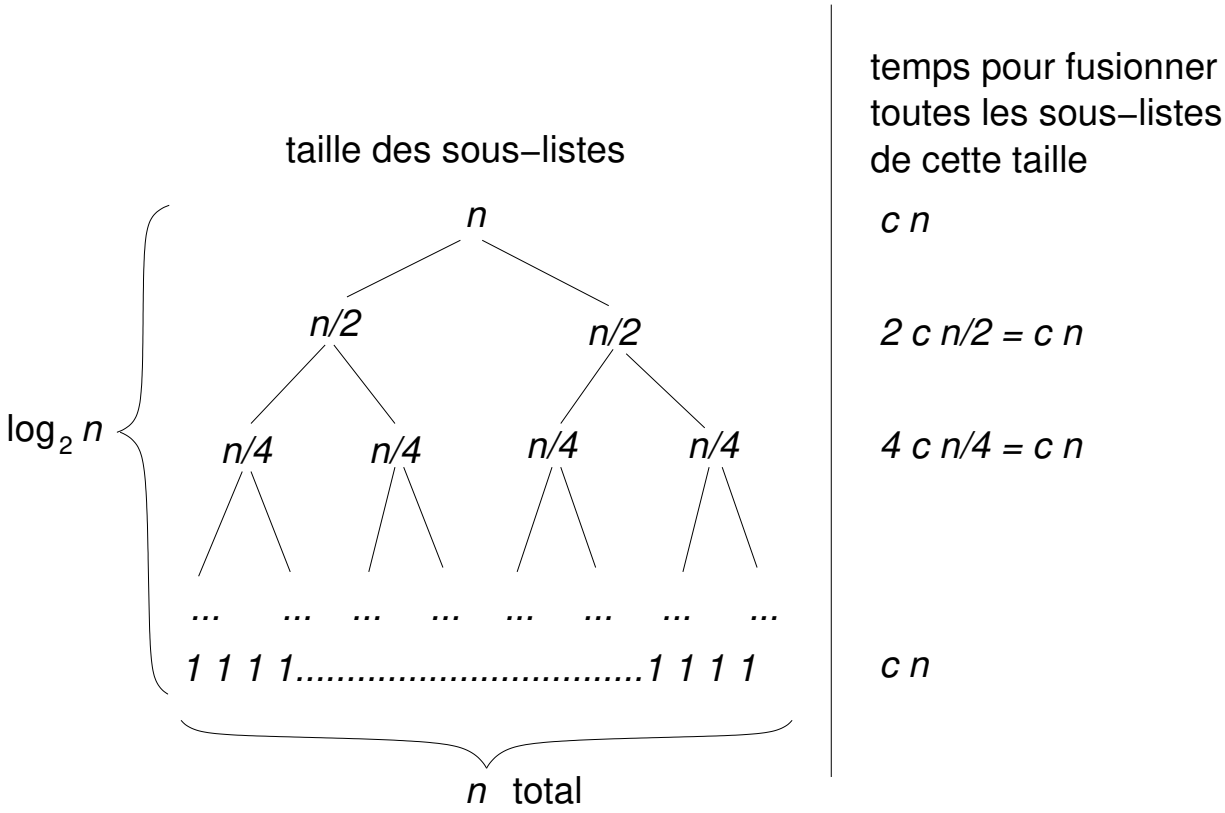
```
# Fonction pour fusionner les sous-listes L[g:m] et L[m:d]
def fusionner(L, g, m, d):
    F = []                  # la liste fusionnée
    gi, di = g, m           # indices parcourant les sous-listes

    while True:
        if gi == m:         # sous-liste L[g:m] épuisée?
            F += L[di:d] #  ajouter le reste de L[m:d] à F
            break           #   et terminer la boucle
        elif di == d:       # sous-liste L[m:d] épuisée?
            F += L[gi:m] #  ajouter le reste de L[g:m] à F
            break           #   et terminer la boucle
        elif L[gi] <= L[di]:
            F.append(L[gi])     # ajouter élément gauche
            gi += 1             # augmenter indice à gauche
        else:
            F.append(L[di])     # ajouter élément droit
            di += 1             # augmenter indice à droite

    L[g:d] = F    # remplacer liste par liste fusionnée
```

Complexité du tri fusion

- Pour simplicité on suppose que la taille n de la liste à trier est une puissance de 2 $\Rightarrow \log_2 n$ divisions à effectuer.
- Pour fusionner deux sous-listes de taille k, il faut $2ck$ opérations simples, où $c = \text{cte}$.



- Complexité du tri fusion est $T(n) \sim c\, n \log_2 n = \mathcal{O}(n \log n)$.

Complexité du tri fusion

- La complexité du tri fusion est $\mathcal{O}(n \log n)$ en tout cas.
- En moyenne il est même plus efficace que le tri rapide.
- Faiblesse : il ne fait pas le tri sur place mais nécessite de la mémoire supplémentaire pour y stocker les sous-listes fusionnées. Complexité en temps réduite mais complexité "en espace" (de mémoire) élévée.

Un des corollaires du **master theorem** (théorème général sur les récurrences de partition) est qu'il n'y a pas d'algorithme de tri par comparaison qui est plus efficace que $\mathcal{O}(n \log n)$ en moyenne.

Complexité des algorithmes de tri

Exercices

- Dérouler à la main l'algorithme du tri fusion sur le tableau `[15, 4, 2, 8, 23, 17, 0, 11]` .
- La complexité du tri fusion est $\mathcal{O}(n \log n)$ en tous cas. Rendez-vous compte de la différence entre les algorithmes $\mathcal{O}(n^2)$ et $\mathcal{O}(n \log n)$: Générez une liste de 100 000 entiers aléatoires et triez-la (sans rien afficher) avec le tri fusion ou avec le tri rapide. Puis, essayez de faire pareil avec le tri par insertion.
- Un algorithme de tri qui n'est pas appliqué en pratique, pour des raisons évidentes, est le *bogosort* ("tri stupide") : Aléatoirement réarranger les éléments, puis tester s'ils sont par hasard dans le bon ordre, et sinon, répéter. Déterminer la complexité en moyenne du tri stupide.

Trier une liste avec `sorted()` et `sort()`

Pour les applications pratiques, Python contient déjà des fonctions de tri :

```
L = [24, 52, 3, 54, 13, 12]
L2 = sorted(L)      # générer une nouvelle liste triée
                    #     sans changer l'original
L.sort()            # trier la liste L
```

L'algorithme sous-jacent est timsort, un algorithme hybride optimisé qui combine des aspects du tri par insertion et du tri fusion. Fonctionnement approximatif :

- Identifier des parties déjà triés, renverser des parties triés à l'envers.
- Diviser en q tronçons de taille entre 32 et 64 éléments, où la taille est choisie de manière que q est (proche de) une puissance de 2.
- Trier les tronçons pas encore triés avec le tri par insertion.
- Fusionner les tronçons avec le tri fusion.
- Complexité $\mathcal{O}(n)$ au meilleur cas, $\mathcal{O}(n \log n)$ en moyenne et au pire cas.

Les classes

Dans ce chapitre

Généralités

- La programmation orientée d'objets

Python

- Les classes
- Les méthodes
- Les méthodes spéciales
- L'héritage
- Copier un objet

La programmation orientée objet

- La programmation orientée objet (object oriented programming) est une approche à la programmation qui sert surtout à mieux structurer les grands projets avec plusieurs contributaires en développement continue.
- Elle favorise la modularité, l'extensibilité et la réutilisation du code.
- Même si elle n'est pas forcement nécessaire pour les petits programmes que nous implémentons ici, il est utile d'en connaître les principes.

Notions centrales : La **classe**, l'**objet**, le **champ de données**, la **méthode**.

- Une **classe** généralise le concept d'un type de données. Contrairement aux types de données élémentaires pré-définies en Python (p.ex. `int`, `float`, `str`, `list` ...) une classe est définissable par le programmeur.
- Un **objet** est un représentant d'une classe (comme une variable ordinaire est un représentant d'un type de données).
- La définition d'une classe comporte les attributs (**champs de données**) ainsi que les fonctionnalités (**méthodes**) qui caractérisent ses objets.

Qu'est-ce que les objets d'une classe peuvent représenter ?

Tout ce qui le programme doit manipuler, par exemple :

- un fenêtre de l'interface graphique, un bouton, un menu utilisateur...
- une fiche / un enregistrement dans une base de données...
- un vecteur, une matrice, une transformation de symétrie, une fonction mathématique, un ensemble de coordonnées...
- un circuit, un point matériel, un atome, un cristal...
- ...

La programmation orientée d'objets

Les objets d'une classe sont caractérisés par

- ses attributs : quel est l'état de l'objet ? ⇒ Champs de données
- ses fonctionnalités : qu'est-ce qu'on peut faire avec ? ⇒ Méthodes

Pour l'exemple d'une classe fictive `Fenetre` qui représenterait une fenêtre sur l'écran (supposant que cette classe a été définie ailleurs) :

```
f = Fenetre()          # crée une fenêtre f
f.largeur = 1000       # un champ de données
f.hauteur = 500        # un autre champ
f.titre = "Programmation pour la physique" # un 3ème champ
f.deplacer(20, 20) # une méthode
f.afficher()           # une autre méthode
```

Principes de la POO

- **Encapsulation :** On réunit les données (les champs) et les fonctions qui les gèrent (les méthodes) dans une seule entité, l'objet.
- **Héritage :** Il y a une hiérarchie des classes et sous-classes, du plus général au plus concret. Les propriétés d'une classe (les champs caractérisant ses objets, et les méthodes associées) sont hérités par ses sous-classes.
- **Polymorphisme :** Ayant hérité une méthode de ses ancêtres, une classe garde toujours la capacité de la redéfinir ou completer. Alors les méthodes peuvent exister en plusieurs versions différentes. En fonction du contexte, le programme décidera dynamiquement pendant l'exécution du code quelle est la bonne version à utiliser.

Les classes

Exemple réaliste : Une classe `Vecteur3D` pour représenter un vecteur réel à trois composantes. Les champs d'un objet du type `Vecteur3D` sont les coefficients x, y, z.

```
class Vecteur3D:

    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def fois(self, r):
        self.x *= r
        self.y *= r
        self.z *= r

    def norme(self):
        return (self.x**2 + self.y**2 + self.z**2)**(1/2)

v = Vecteur3D(1, 2, -0.5) # créer un nouveau vecteur
v.fois(2)                 # le multiplier par 2
print(v.x)                # afficher sa composante x
print(v.norme())          # afficher sa norme
```

Les classes

Si `v` est un objet de la classe `Vecteur3D` :

- `v.__init__()`, `v.fois()` et `v.norme()` sont des méthodes
- Les coefficients `v.x`, `v.y`, `v.z` du vecteur sont des champs.
- La méthode d'initialisation `Vecteur3D.__init__()` est appellée automatiquement lorsqu'un objet du type `Vecteur3D` est créé, dans notre exemple par la commande `v = Vecteur3D(1, 2, -0.5)`
  - Le premier argument de toute méthode est toujours une reférence à l'objet soi-même qui est passée automatiquement à chaque appel. Par convention on le dénomme `self`.
  - Dans notre exemple, la méthode `__init__(self, x, y, z)` initialise les champs `v.x`, `v.y`, `v.z` avec ses autres trois arguments `x`, `y`, `z`
  - Si la définition d'une classe ne contient pas une méthode `__init__()`, on peut toujours créer un objet, mais aucun champ ne sera initialisé. Par exemple, un objet `t` d'une classe hypothétique `Truc` peut être crée avec `t = Truc()`.
- La méthode `v.fois()` multiplie tous les coefficients avec l'argument `r`.
- La méthode `v.norme()` calcule et renvoie la norme du vecteur.

Les méthodes

À noter la différence entre les méthodes et les fonctions :

- Une méthode est toujours associée à une classe. Une fonction est indépendante des classes.
- La notation pour la définition d'une méthode est la même que pour une fonction, mais il y a deux différences importantes :
  - La définition d'une méthode se trouve toujours dans la définition de la classe correspondante.
  - Toute méthode doit accepter au moins un argument (l'argument `self`).
- On appelle une fonction `fonc()` avec les valeurs des arguments `VAL1`, `VAL2` etc. avec la commande `fonc(VAL1, VAL2, ...)`
- On appelle une méthode `Cls.meth()` d'une classe `Cls` avec la commande `obj.meth(VAL1, VAL2, ...)`, où `obj` est un objet de la classe `Cls`.
- Lors d'un appel d'une méthode, le premier argument `self` est passé implicitement. Sa valeur est toujours l'objet lui-même qui figure dans l'appel de la méthode (`obj` dans l'exemple ci-dessus). Les valeurs `VAL1`, `VAL2`, ... correspondent alors aux valeurs du deuxième, troisième, ... argument.
- La commande `obj.meth(VAL1, VAL2, ...)` est en fait équivalente à la commande `Cls.meth(obj, VAL1, VAL2, ...)`, mais en pratique on n'utilisera pas cette dernière forme.

Les méthodes et les champs : Un deuxième exemple

```
from math import pi, sqrt

class Cercle:

    def __init__(self, x=0.0, y=0.0, rayon=1.0):
        self.x = x  # des champs pour représenter
        self.y = y  #   le centre d'un cercle
        self.rayon = rayon  # et son rayon

    def aire(self):  # une méthode pour calculer l'aire
        return pi * self.rayon**2

    def contient(self, x, y): # le point (x,y) est-il
        dx = self.x - x         #  à l'intérieur du cercle?
        dy = self.y - y
        if sqrt(dx**2 + dy**2) < self.rayon:
            return True
        else:
            return False
```

Les méthodes et les champs : Un deuxième exemple

Exemples d'utilisation des méthodes de la classe `Cercle` :

```
c1 = Cercle(rayon=2.0)  # cercle centré à l'origine, rayon 2
c2 = Cercle(-1.0, 3.5, 2.5) # cercle de rayon 2.5 à (-1,3.5)
print(c1.aire()) # affiche l'aire de c1

if c2.contient(c1.x, c1.y):
    print("Le centre du cercle c1 est à l'intérieur du cercle
        c2.")
```

(Évidemment la classe `Cercle` et ses méthodes ne sont pas trop utiles en pratique ; leur seul intérêt est la démonstration de l'utilisation des méthodes associées à une classe.)

Les méthodes et les champs

Exercices

- Réaliser une méthode `Vecteur3D.prodscal(self, v)`, de la sorte que `u.prodscal(v)` renvoie le produit scalaire entre les vecteurs `u` et `v`.
- Eprouver votre code dans un script Python.

Exercices

- Réaliser une nouvelle classe `Adresse` qui peut représenter une adresse dans une base de données. Les champs seront `nom`, `numero`, `rue`, `code_postal` et `ville`. Un objet peut être crée soit avec la commande
  `Adresse("Jean Dupont", 17, "rue Victor Hugo", 75000, "Paris")`
  soit simplement avec la commande `Adresse()`. Dans ce dernier cas, les valeurs seront 0 pour le numéro et le code postal, et des `str` vides `""` pour les autres champs. Il y aura une méthode `Adresse.changer_donnees()` qui utilise la fonction `input()` pour laisser l'utilisateur entrer les données par le clavier.
- Eprouver votre code dans un script Python.

Les variables de classe

Il est parfois utile de définir un champ dont la valeur est la même pour tous les objets de la classe. Un tel attribut s'appelle variable de classe. Exemple :

```
class Vecteur3D:

    dimension = 3    # une variable de classe

    def __init__(self, x, y, z):

        (....)

v = Vecteur3D(1, 1, 1)
print(v.x)           # affiche la composante x du vecteur v
print(v.dimension)   # affiche toujours 3 pour tout vecteur
```

Les méthodes spéciales

La définition d'une classe peut contenir des implémentations pour certaines méthodes spéciales. Normalement on n'appelle pas explicitement ces méthodes dans un programme. Leur rôle est plutôt de déterminer le comportement des objets sous des opérations spécifiques.

- Comme nous avons vu, une méthode `__init__()` contient des instructions à exécuter lors de la création d'un objet.
- Si la définition d'une classe contient une méthode `__str__()`, cette méthode doit renvoyer un `str` qui représente l'objet. Ainsi, si la méthode `Truc.__str__()` est définie, l'interpréteur l'utilisera pour convertir un objet `t` de la classe `Truc` avec `str(t)`, ou pour l'afficher à l'écran avec `print(t)`.

```
def Cercle:

    (....)

    def __str__(self):
        return "Cercle de rayon " + str(self.rayon)\
           + " autour de (x, y) = (" \
           + str(self.x) + ", " + str(self.y) + ")"
```

Les méthodes speciales

- Les méthodes spéciales `__add__()`, `__sub__()`, `__mul__()` et autres permettent de définir le comportement des objets sous les opérations arithmétiques : addition, soustraction, multiplication etc.
- Les méthodes spéciales `__getitem__()` et `__setitem__()` définissent l'accès aux composantes d'un objet (si sa classe est une classe d'objets séquentiels). La méthode spéciale `__len__()` est censée retourner sa longueur.
- Il existe aussi la possibilité d'implémenter des méthodes spéciales pour définir
  - la comparaison des objets
  - la conversion en types numériques
  - le comportement si un objet est appellé comme une fonction
  - et de nombreuses autres fonctionnalités (voir documentation)

Les méthodes spéciales

Implémentation de la méthode `__str__()` pour la classe `Vecteur3D` :

```
class Vecteur3D:

    (....)

    def __str__(self):
        return "(" + str(self.x) + ", " \
                   + str(self.y) + ", " \
                   + str(self.z) + ")"
```

Cette méthode spéciale permet d'afficher les coefficients d'un vecteur `v` avec `print(v)`.
Plus intéressant : Définir la somme de deux vecteurs avec `__add__()`,

```
    def __add__(self, autre):
        sommex = self.x + autre.x
        sommey = self.y + autre.y
        sommez = self.z + autre.z
        return Vecteur3D(sommex, sommey, sommez)
```

Ainsi l'expression `v1 + v2` pour deux vecteurs est définie par `v1.__add__(v2)`.

Résumé : Nos méthodes de la classe `Vecteur3D`

commande	effet	appel interne de méthode
`v = Vecteur3D(1, 3, -1)`	crée un vecteur $v = \begin{pmatrix} 1 \\ 3 \\ -1 \end{pmatrix}$	`Vecteur3D.__init__(v, 1, 3, -1)`
`v.fois(3)`	multiplie `v` par 3 $v = \begin{pmatrix} 3 \\ 9 \\ -3 \end{pmatrix}$	`Vecteur3D.fois(v, 3)`
`n = v.norme()`	`n` = la norme `|v|`	`Vecteur3D.norme(m)`
`w = v + v`	la somme de v et v $w = \begin{pmatrix} 2 \\ 6 \\ -2 \end{pmatrix}$	`Vecteur3D.__add__(v, v)`
`s = str(v)`	la représentation `str` de `v` `s = "(3, 9, -3)"`	`Vecteur3D.__str__(v)`
`p = v.prodscal(w)`	produit scalaire `p` $= v \cdot w = 66$	`Vecteur3D.prodscal(v, w)`

Les méthodes spéciales

Exercices

- Implémenter les méthodes speciales `__sub__()` et `__mul__()` (syntaxe équivalente à `__add__()`). Ici `__sub__(self, autre)` sert à définir l'expression `v - w` par la valeur de retour de `Vecteur3D.__sub__(v, w)` qui sera la différence habituelle des vecteurs, composante par composante. De plus `__mul__(self, autre)` définit l'expression `v * w` par la valeur de retour de `Vecteur3D.__mul__(v, w)` qui sera le produit vectoriel $\mathtt{v} \wedge \mathtt{w}$.
- Eprouver votre code dans un script Python

L'héritage

Une classe peut être une sous-classe d'une autre : ses objets héritent les champs et méthodes de la classe dont elle dérive, mais ils peuvent également disposer des champs et méthodes supplémentaires. Pour définir une sous-classe `ClasseDerivee` de la classe `ClasseParentale` :

```
class ClasseDerivee(ClasseParentale):
    INSTRUCTION1
    INSTRUCTION2
    ...
```

- Une méthode définie dans une sous-classe remplace la méthode du même nom de la classe parentale.
- Pour utiliser la définition originale dans la classe parentale, préfixer `super()` :

```
super().nom_de_methode(*arguments)
```

(utile pour élargir la définition d'une méthode dans la sous-classe, sans la remplacer completement).

L'héritage

```
class Animal:
    def manger(self):
        ...
        ...
    def dormir(self):
        ...
        ...
    ...
```

```
class Chien(Animal):
    def manger(self):
        ...
        ...
    def aboyer(self):
        ...
        ...
    ...
```

- Les objets de la classe `Animal` disposent de toutes méthodes de leur propre classe : `Animal.manger()`, `Animal.dormir()`...
- Les objets de la classe `Chien` disposent de toutes méthodes de la classe `Animal`, ainsi que de la méthode supplémentaire `Chien.aboyer()`.
- Si `milou` est un objet de la classe `Chien`, la commande `milou.manger()` appelle la méthode `Chien.manger()`, pas `Animal.manger()`.
- Dans la définition des méthodes de la classe `Chien`, la commande `super().manger()` appelle la méthode `Animal.manger()`.

L'héritage

Exemple : Définissons une sous-classe `VecteurUnitaire` de la classe `Vecteur3D` dont les objets sont des vecteurs unitaires.

```
class VecteurUnitaire(Vecteur3D):

    import math

    def __init__(self, theta, phi):
        x = math.sin(theta) * math.cos(phi)
        y = math.sin(theta) * math.sin(phi)
        z = math.cos(theta)
        super().__init__(x, y, z)

    def norme(self):
        return 1.0
```

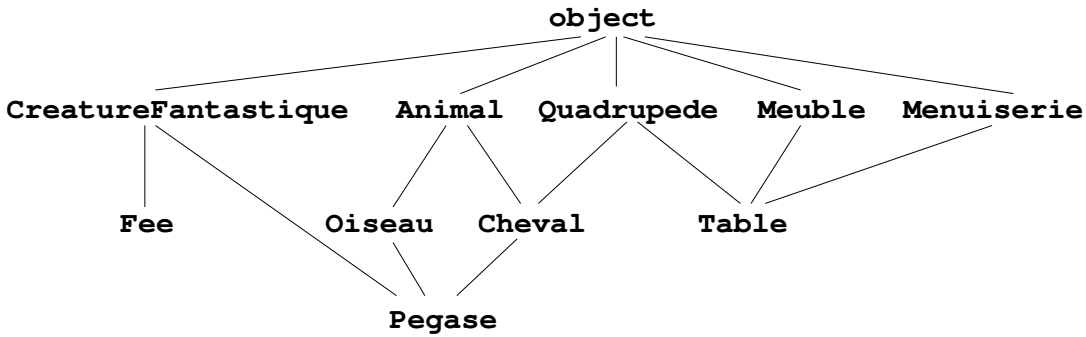
- Les méthodes `__init__()` et `norme()` ont été redéfinies pour la sous-classe.
- En particulier, la nouvelle implémentation de `__init__()` appelle la méthode `__init__()` de la classe parentale.

Exercices

- Réaliser une méthode `VecteurUnitaire.angle(self, autre)` telle que `v.angle(w)` renvoie l'angle entre `v` et `w`. Se servir de la fonction `math.acos()` et de la méthode `Vecteur3D.prodscal()`.

L'héritage multiple

- Une sous-classe peut avoir d'autres sous-classes elle-même.
- Une classe peut avoir plusieurs classes parentales dont elle hérite les propriétés. On parle de héritage multiple.
- Le résultat peut être toute une hiérarchie de classes :



Syntaxe :

```
# toutes les classes parentales sont en parenthèses:
class Table(Quadrupede, Meuble, Menuiserie):
    (....)
```

- Complication : Si deux classes parentales rédéfinissent la même méthode, quelle version faut-il hériter ? ⇒ Method resolution order, sujet avancé.
- Toute classe hérite automatiquement de la classe `object`, native à Python.

Passage par référence/par valeur

```
def ma_fonction(x):
    ...
...
ma_fonction(y)
```

Il y a deux possibilités pour un langage de programmation de passer un argument à une fonction :

- Passage par valeur : Quand `ma_fonction()` est appellée avec la commande `ma_fonction(y)`, une nouvelle variable `x` est créée avec la même valeur que `y`. Si la fonction change la valeur de `x`, cela ne change pas la valeur de `y`.
- Passage par référence : Lors de l'appel de `ma_fonction(y)`, une nouvelle référence `x` aux données contenues dans `y` est créée. Après, la fonction peut changer ces données en accédant `x`.

Python utilise le passage par référence. Mais certains types de données sont immuables : leurs valeurs ne peuvent pas être changées. Une nouvelle affectation à un identifiant d'une variable immuable va créer une nouvelle variable locale (voir diapos 90/91).

Les types muables et immuables

Code pour illustrer la différence :

```
ma_liste = [1, 2, 3]    # list est un type muable
mon_nombre = 4          # int est un type immuable

def une_fonction(une_liste, un_nombre):

 # cette commande change la valeur de une_liste:
    une_liste[0] = 0

 # cette commande NE CHANGE PAS la valeur de un_nombre
 # mais crée une nouvelle variable locale:
    un_nombre = 5

    print(une_liste)    # "[0, 2, 3]"
    print(un_nombre)    # "5"

une_fonction(ma_liste, mon_nombre)
print(ma_liste)         # "[0, 2, 3]"
print(mon_nombre)       # "4"
```

Les types muables / immuables

Types immuables :

- types numériques : `int`, `float`, `complex`
- chaînes de caractères (`str`)
- *n*-uplets (`tuple`)

Types muables :

- `list`
- toutes les classes

Copier un objet

Comportement similaire si on associe un objet muable (comme une `list` ou un objet d'une classe) à un nouvel identifiant. Aussi en cette situation Python ne crée pas un nouvel objet, mais seulement une nouvelle référence :

```
a = Vecteur3D(0, 0, 0) # vecteur nul
b = a  # nouvelle référence au même objet,
       # pas un nouvel objet!
b.x = 1
print(a)  # "(1, 0, 0)"
print(b)  # "(1, 0, 0)"
```

Créer une copie indépendante de l'objet original avec `copy.deepcopy()` :

```
from copy import deepcopy
a = Vecteur3D(0, 0, 0) # vecteur nul
b = deepcopy(a)        # "copie profonde"
b.x = 1
print(a)  # "(0, 0, 0)"
print(b)  # "(1, 0, 0)"
```

Copier un objet, plus de détails

- Pour des objets muables (par exemple les listes), une affectation crée une nouvelle référence à l'objet sans faire une copie dans la mémoire :

```
li1 = [1, 2, [3, 4]]
li2 = li1
li2[0] = 5
print(li1)  # "[5, 2, [3, 4]]"
```

- La fonction copy.copy() (ou, pour les listes, l'opérateur [:]) fait une copie superficielle sans copier les objets composés imbriqués.

```
import copy
li3 = copy.copy(li1)  # même effet: "li3 = li1[:]"
li3[0] = 6            # change li3 mais pas li1
li3[2][0] = 7  # change li3[2] et li1[2]
print(li1)     # "[5, 2, [7, 4]]"
print(li3)     # "[6, 2, [7, 4]]"
```

- copy.deepcopy() fait une copie profonde incluant les sous-objets.

Exercices

- Réaliser une classe `Carte` pour représenter une carte dans un jeu de cartes. Il y aura deux champs, `Carte.rang` pour représenter le rang de la carte (2, 3, 4, 5, 6, 7, 8, 9, 10, V, D, R ou A) et `Carte.couleur` pour représenter sa couleur (pique ♠, coeur ♡, carreau ♢ ou trèfle ♣). De plus il y aura deux méthodes, `Carte.__init__(self, r, c)` pour initialiser un objet et `Carte.__str__(self)` pour générer sa représentation `str`.
- Réaliser une classe `Main` pour représenter une main de cartes. Il y aura un seul champ `Main.cartes`, qui sera une liste d'objets de la classe `Carte`. De plus il y aura trois méthodes, `Main.__init__(self)` pour construire une nouvelle main vide, `Main.__str__(self)` pour générer la représentation `str`, et `Main.piocher(self, c)` pour ajouter la carte `c` à la main.
- Réaliser un programme qui permet de simuler le jeu aux cartes de *poker fermé* avec des règles simplifiées (voir énoncé).

Les bibliothèques NumPy et matplotlib

Dans ce chapitre

Python

- La bibliothèque `numpy`
- Manipulation des `array`
- Importer et exporter des données
- Visualiser des données avec `matplotlib`

Les tableaux (la classe numpy.ndarray)

On rappelle les caractéristiques des listes de Python :

- contient plusieurs éléments qui ne sont pas forcement de même type
- taille variable, peut changer (p.ex. avec `list.append()`)
- 1-dimensionnel = 1 seul indice (sauf si les éléments sont eux-mêmes des listes)

La bibliothèque NumPy est structurée autour d'une classe similaire, le tableau (anglais : "array") `numpy.ndarray` ou `numpy.array`.

Caractéristiques des tableaux `numpy` :

- contiennent plusieurs éléments qui sont forcement de même type, et toujours d'un type numérique (`int`, `float`, `complex` ...)
- taille fixe
- n-dimensionnel : vecteurs, matrices, tenseurs...
- optimisé pour le calcul numérique : plus rapide que les listes, beaucoup de fonctionnalité pour la manipulation efficace.

Créer un tableau

Exemples : Pour créer des objets de la classe `numpy.ndarray`

```
import numpy as np

sigma3 = np.array([[1, 0], [0, -1]])  # matrice [[ 1  0]
                                      #          [ 0 -1]]

s = np.array([1, 0], dtype=complex) # vect. [ 1.+0.j 0.+0.j]

nul2x3 = np.zeros((2, 3))  # [[ 0. 0. 0.]
                           #  [ 0. 0. 0.]]

iden = np.identity(3, dtype=int) # [[ 1 0 0]
                                 #  [ 0 1 0]
                                 #  [ 0 0 1]]

rng = np.arange(0.8, 2, 0.4)     # [ 0.8 1.2 1.6]
```

Créer un tableau

Un `numpy.ndarray` peut se créer

- en spécifiant les éléments dans une liste (ou liste de listes...) avec `numpy.array()`
- en spécifiant les dimensions par un tuple `(x, y, z...)` des `int`
  - `numpy.zeros()` crée un `ndarray` de zéros
  - `numpy.ones()` crée un `ndarray` de uns
  - `numpy.empty()` crée un `ndarray` sans initialiser les éléments
- cas spécial : la matrice d'identité en `n` dimensions, `numpy.identity(n)`
- tableaux 1-dimensionnels de nombres uniformement espacés :
  - `numpy.arange(debut, fin, pas)` : comme `range` mais avec des `float`
  - `numpy.linspace(debut, fin, N)` : `N` nombres équidistants entre `debut` et `fin`

Si besoin, spécifier le type de données des éléments avec l'argument `dtype` lors de la construction

Opérations arithmétiques sur les tableaux

Les opérations arithmétiques `+ = * / // %` entre les tableaux `numpy` sont définies par élément :

```
import numpy as np
sigma3 = np.array([[1, 0], [0, -1]], dtype=float)
print(sigma3 * np.array([[2., 3.] [4., 5.]])) # [[ 2.  0.]
                                              #  [ 0. -5.]]
```

Si les dimensions ne s'accordent pas, une opération arithmétique impliquant deux tableaux va produire un `ValueError`.

En revanche, il est toujours possible de ajouter/soustraire/multiplier/diviser par un scalaire :

```
print(sigma3 - 1)   # [[  0. -1.]
                    #  [ -1. -2.]]
```

Indicer et couper un tableau

On peut indicer un `numpy.ndarray` avec plusieurs indices selon ses dimensions :

```
sigma3 = np.array([[1, 0], [0, -1]], dtype=float)
print(sigma3[1, 1])   # "-1.0"
```

(avec la généralisation évidente pour des tableaux d-dimensionnels). On peut aussi les couper comme des listes Python :

```
# tous les éléments de la deuxième colonne:
print(sigma3[:, 1])   # "[ 0. -1.]"
# tous les éléments de la première ligne
print(sigma3[0, :])   # "[ 1. 0.]"
```

Couper des tableaux (array slicing), méthodes avancées

Créer le vecteur $0 \ldots 11$ et le réarranger dans une matrice 3×4 :

```
a = np.array(range(12)).reshape((3,4))
print(a)    # [[ 0  1  2  3]
            #  [ 4  5  6  7]
            #  [ 8  9 10 11]]
```

Avec l'opérateur `[i:j:k]` on accède aux éléments

- à partir de l'indice `i` (par défaut : début)
- jusqu'à l'indice `j` exclu (par défaut : fin)
- avec un pas de `k` (par défaut : 1)

Exemples :

```
6
print(a[1, ::2])   # [4 6] (2ème ligne, colonnes pairs)
print(a[1, 1::2])  # [5 7] (2ème ligne, colonnes impairs)
print(a[0, 1:3])   # [1 2] (1ère ligne, colonnes 1 et 2)
print(a[1:,:2])    # [[4 5]  (derniers 2 éléments
                   #  [8 9]]  des premières 2 colonnes)
```

Exercices

- Créer un tableau de nombres flottants 1.5, 2, 2.5 . . . 10.5, 11.
- Réarranger dans un tableau 4×5 avec la méthode `ndarray.reshape()`.
- De ce tableau, extraire les colonnes aux indices impairs.
- Extraire les lignes à partir de l'indice 2.
- De la dernière ligne, extraire les éléments aux indices pairs à partir de 2.

Multiplication matricielle avec les tableaux

Un tableau avec deux indices peut représenter une matrice. Un tableau avec un seul indice est un vecteur.
Les produits entre les matrices et vecteurs (produit scalaire entre deux vecteurs, action d'une matrice sur un vecteur, produit matriciel entre deux matrices) se calculent avec la fonction numpy.dot().

Exemples :

```
M = np.array([[1., 2., 4.], [2., -1., 0.], [5., -2., 1.]])
v = np.array([0., 1., 2.])
w = np.array([1., -1., 1.])

print(np.dot(v, w)) # produit scalaire v . w, résultat: 1.0

print(np.dot(M, v)) # matrice agît sur vecteur, M . v
                    #  résultat: [ 10. -1. 0.]

print(np.dot(M, M)) # produit matriciel M . M
                    #  résultat: [[ 25.  -8.   8.]
                    #             [  0.   5.   8.]
                    #             [  6.  10.  21.]]
```

Multiplication matricielle avec les tableaux

Exemple : Calcul des moyennes quantiques $\langle\psi|\sigma^{1,2,3}|\psi\rangle$ pour un système de deux niveaux

```
import numpy as np

sigma = np.array([[[ 0, 1], [1, 0]], # les matrices de Pauli
                  [[ 0, -1j], [1j, 0]],
                  [[ 1, 0], [0, -1]]], dtype = complex)

def norm(psi):  # la norme d'un vecteur complexe
    psic = np.conjugate(psi)
    return np.dot(psic, psi)**(1/2)

psi1 = complex(input("Entrer psi1: "))  # composantes de psi
psi2 = complex(input("Entrer psi2: "))
psi = np.array([psi1, psi2], dtype=complex)
psi /= norm(psi)              # normaliser le vecteur psi
psic = np.conjugate(psi)  # le vecteur conjugué complexe
vm = [np.dot(np.dot(psic, sigma[i]), psi) for i in range(3)]

print("Valeurs moyennes:\n <sigma1> = ", vm[0].real,
                        "\n <sigma2> = ", vm[1].real,
                        "\n <sigma3> = ", vm[2].real)
```

Méthodes utiles pour le calcul matriciel

La classe `numpy.ndarray` contient quelques autres champs et méthodes utiles pour manipuler des vecteurs et matrices : si `A` est un `numpy.ndarray`, alors

- `A.T` représente la transposée de `A`
- `A.trace()` calcule la trace $\sum_i A_{ii}$
- `A.conj()` calcule la matrice conjuguée complexe
- `A.max()` calcule l'élément maximal
- `A.sum()` calcule la somme des éléments
- ...

Pour chacune de ces méthodes il existe une fonction équivalente : `numpy.trace(A)`, `numpy.conjugate(A)`, `numpy.amax(A)`, `numpy.sum(A)`. De même, au lieu de `numpy.dot(A, B)` on peut écrire `A.dot(B)`.

Voir `https://docs.scipy.org/doc/numpy/reference/routines.html` pour documentation complète.

Exercices

- Réaliser un programme qui effectue le développement multipolaire en coordonnées cartésiennes pour une configuration de charges électriques donnée. (Voir TD7 pour les détails.)

Fonctions sur les tableaux

Les fonctions élémentaires `sin`, `cos`, `exp`, `log`, `sqrt` etc. des bibliothèques `math` et `cmath` existent aussi dans la bibliothèque `numpy`. Si on donne un tableau comme argument, la valeur de retour sera également un tableau avec les valeurs de fonction des éléments : "array broadcasting".

```
import numpy as np

x = np.array([-1, 0, 1])
print(np.arccos(x)) # "[ 3.14159265  1.57079633  0.]"
```

Equivalent (mais beaucoup moins vite et moins nette) :

```
import math

x = [-1, 0, 1]
acosx = [math.acos(t) for t in x]
print(acosx)
```

Fonctions sur les tableaux

Pour convertir une fonction ordinaire en fonction qui peut s'appliquer sur un tableau numpy, on utilise la fonction `numpy.vectorize`.

Exemple :

```
import numpy as np

# Une fonction ordinaire:
def f(x, y):  # retourne x si x>y et y-x sinon
    if x > y:
        return x
    return y - x

# La fonction vectorisée:
vf = np.vectorize(f)

x = np.array([1., 3., 7.])
vf(x, 4)  # array([ 3., 1., 7.])
```

Copier un tableau

Une copie par référence se fait avec l'opérateur d'affectation =, une copie profonde (comme `copy.deepcopy()`) avec `numpy.ndarray.copy()` :

```
a = sigma3          # permet d'accéder à sigma3
                    #  avec la novelle référence a
b = sigma3.copy()   # crée un nouvel objet
                    #  qui est une copie de sigma3
```

Importer et exporter des données

La fonction `numpy.loadtxt` permet d'importer des données d'un fichier.

```
# Fichier de données "donnees.dat"

3.14159 2.71828 0.57721   # commentaires seront ignorés
1.      -2.     3.e5
```

```
# Fichier du programme

import numpy as np

a = np.loadtxt("donnees.dat")
print(a) # [[ 3.1415900e+00  2.7182800e+00 5.7721000e-01]
         #  [ 1.0000000e+00 -2.0000000e+00 3.0000000e+05]]
```

Dans le fichier de données :

- éléments séparés par un ou plusieurs espaces blancs
- lignes blanches et commentaires # ignorés

Importer et exporter des données

La fonction `numpy.savetxt` permet d'enregistrer des données dans un fichier.

```
import numpy as np

a = np.arange(0.0, 4.0, 1.0) # le tableau [ 0., 1., 2., 3.]
np.savetxt("mydata.dat", a, header="Commentaire facultatif")
```

Fichier `mydata.dat` résultant :

```
# Commentaire facultatif
0.00000000000000000e+00
1.00000000000000000e+00
2.00000000000000000e+00
3.00000000000000000e+00
```

Visualisation avec matplotlib

Python dispose d'une bibliothèque très puissante pour créer des graphiques : la bibliothèque `matplotlib.pyplot`.

Usage typique pour tracer le graphe d'une fonction :

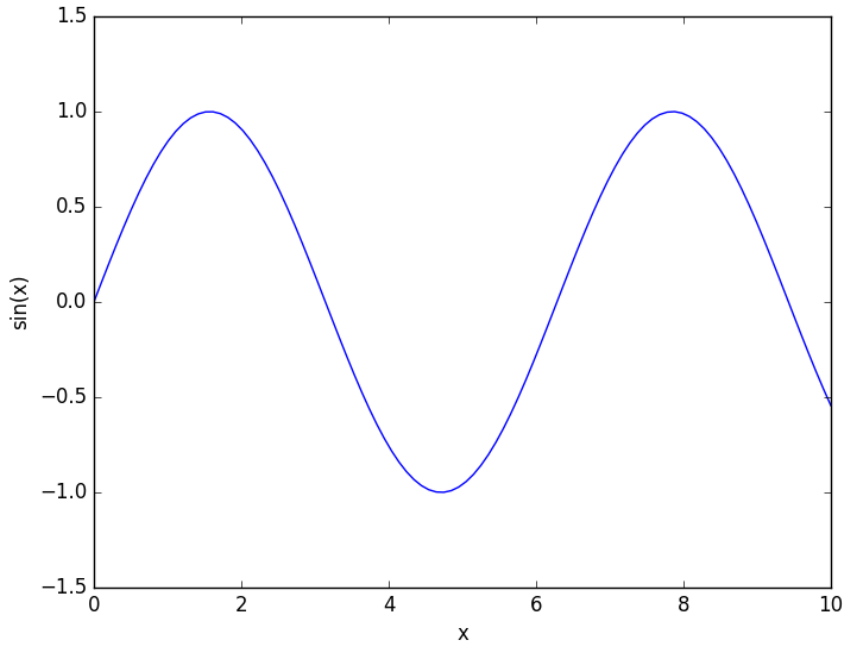
```
import numpy as np
import matplotlib.pyplot as plt

xpoints = np.linspace(0., 10., 100) # 100 pts entre 0 et 10
ypoints = np.sin(xpoints)           # valeurs de sin()

plt.plot(xpoints, ypoints) # tracer y en fonction de x
plt.ylim([-1.5, 1.5])      # pour y entre -1.5 et 1.5
plt.xlabel("x")            # étiquette de l'axe des x
plt.ylabel("sin(x)")       #    ... et de l'axe des y
plt.show()                 # afficher graphique
```

Visualisation avec matplotlib

Résultat :



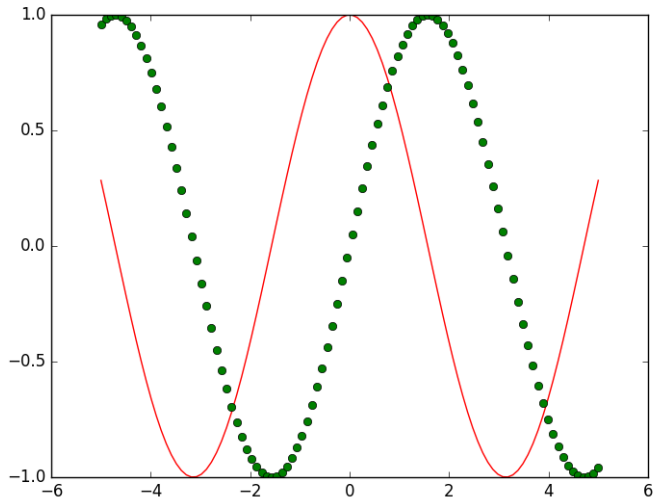
La fonction `matplotlib.pyplot.plot()`

Tracer des courbes ou des points : `matplotlib.pyplot.plot(x, y, m)`

- `x` = valeurs des x
- `y` = valeurs de fonction $y(x)$ à tracer
- optionnel : `m` = chaîne de caractères indiquant la couleur et le style du marqueur ou de la courbe, par exemple
  - `'r'`, `'g'`, `'b'` = rouge, vert, bleu (défaut)
  - `'-'`, `'--'`, `':'` = ligne solide (défaut), interrompue, pointillée
  - pour tracer des points individuels plutôt qu'une courbe :
    `'.'`, `','`, `'o'`, `'*'`, `'s'` = marqueur point, pixel, cercle, étoile, carré

Exemple :

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-5, 5, 100)
cosx = np.cos(x)
sinx = np.sin(x)
plt.plot(x, cosx, 'r')
plt.plot(x, sinx, 'go')
plt.show()
```



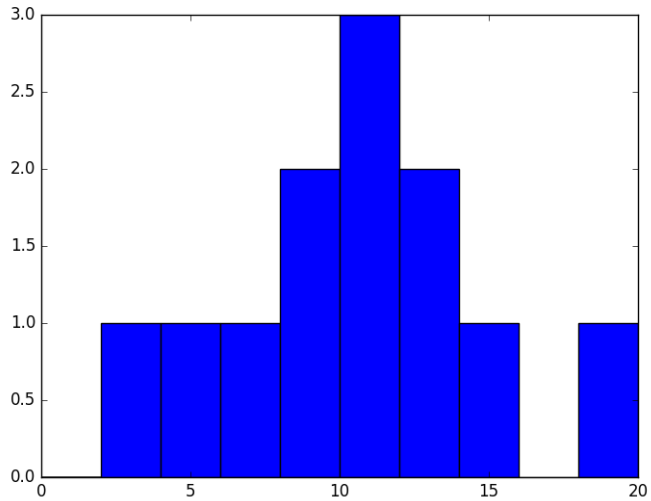
La fonction `matplotlib.pyplot.hist()`

Tracer des histogrammes : `matplotlib.pyplot.hist(x, bins, range)`

- `x` = valeurs à tracer
- optionnel : `bins` = nombre de barres
- optionnel : `range` = tuple `(x_minimal, x_maximal)`

Exemple :

```
import numpy as np
import matplotlib.pyplot as plt
notes = [13.5, 5.75, 10., 11.25,
         18., 7.5, 13., 8.75,
         10.5, 14., 9.25, 3.25]
plt.hist(notes, 10, (0, 20))
plt.show()
```



Tracer des fonctions de deux variables

Pour tracer une fonction $z(x, y)$ il faut d'abord créer un maillage (anglais : "meshgrid") pour représenter les binômes de coordonnées (x, y). Il convient de se servir de la fonction `numpy.meshgrid(x, y)`. Exemple :

```
import numpy as np

# 101 valeurs de x entre 0 et 10: [0.0 0.1 0.2 ... 10]
x = np.linspace(0, 10, 101)

# 10 valeurs de y,  3 <= y < 4: [3.0 3.1 3.2 ... 3.9]
y = np.arange(3.0, 4.0, 0.1)

X, Y = np.meshgrid(x, y)
```

Résultat : deux tableaux 10×101,

$$\mathtt{X} = \left.\begin{pmatrix} 0 & 0.1 & 0.2 & \dots & 10 \\ 0 & 0.1 & 0.2 & \dots & 10 \\ & & \dots & & \end{pmatrix}\right\} 10 \text{ lignes}, \quad \mathtt{Y} = \underbrace{\begin{pmatrix} 3 & 3 & \dots & 3 \\ 3.1 & 3.1 & \dots & 3.1 \\ \vdots & & & \vdots \\ 3.9 & 3.9 & \dots & 3.9 \end{pmatrix}}_{101 \text{ colonnes}}$$

La fonction `matplotlib.pyplot.imshow()`

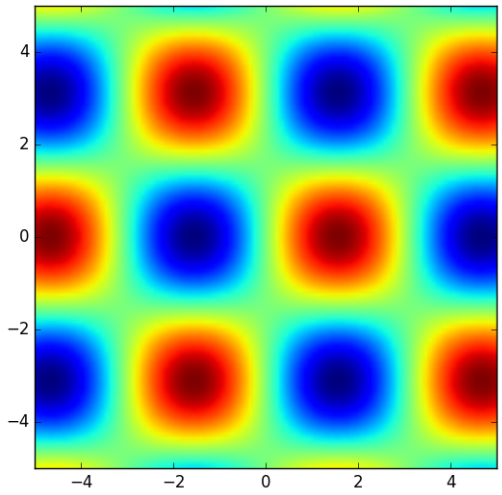
Tracer des cartes de chaleur (heat map) : `matplotlib.pyplot.imshow(z, extent)`

- `z` = tableau 2D avec les valeurs de fonction
- argument facultatif : `extent` = liste avec les `x` et `y` minimales et maximales

Exemple :

```
import numpy as np
import matplotlib.pyplot as plt

x = y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
z = np.sin(X) * np.cos(Y)
plt.imshow(z, extent=[-5,5,-5,5])
plt.show()
```



La fonction `matplotlib.pyplot.contour()`

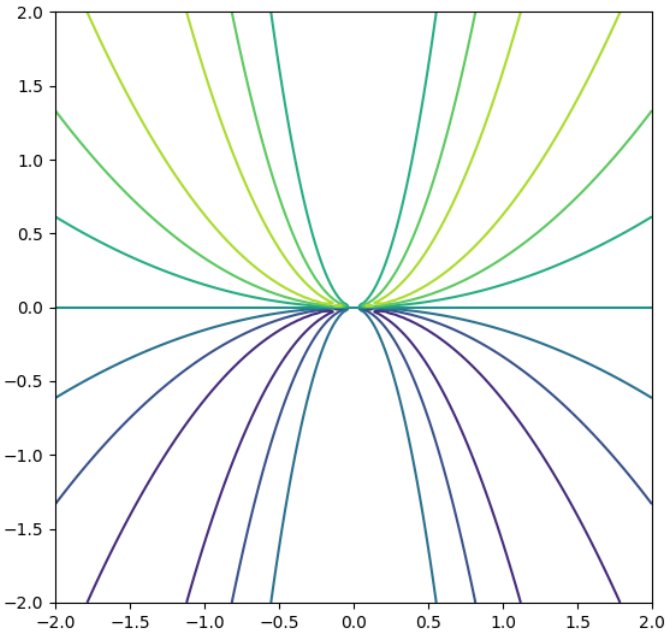
Tracer des courbes de niveau : `matplotlib.pyplot.contour(x, y, z)`

- `z` = tableau 2D avec les valeurs de fonction
- arguments facultatifs : `x`, `y` = tableaux avec les `x` et `y` correspondants

Exemple :

```
import numpy as np
import matplotlib.pyplot as plt

def f(x, y):
  if x == 0 and y == 0:
    return 0.0
  return x**2 * y / (x**4 + y**2)
vf = np.vectorize(f)
x = y = np.linspace(-2, 2, 200)
X, Y = np.meshgrid(x, y)
Z = vf(X, Y)
plt.contour(X, Y, Z)
plt.show()
```



Application III : Algèbre linéaire numérique

Dans ce chapitre

Généralités

- Systèmes d'équations linéaires
- Diagonalisation des matrices
- Regression non linéaire

Algorithmes

- La méthode de Gauss
- La décomposition LU
- La décomposition QR et l'algorithme QR
- Application exemplaire : Regression non linéaire avec l'algorithme de Gauss-Newton

Résoudre un système d'équations linéaires

On cherche une solution des n équations linéaires à n inconnues x_n

$$
\begin{aligned}
a_{11}\,x_1 + a_{12}\,x_2 + \ldots + a_{1n}\,x_n &= b_1 \\
a_{21}\,x_1 + a_{22}\,x_2 + \ldots + a_{2n}\,x_n &= b_2 \\
&\ldots \\
a_{n1}\,x_1 + a_{n2}\,x_2 + \ldots + a_{nn}\,x_n &= b_n
\end{aligned}
$$

L'algorithme de Gauss

L'algorithme de Gauss est une méthode pour résoudre ces systèmes d'équations linéaires. Le système peut être écrit comme une équation matricielle :

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

On considère la matrice $n \times (n+1)$ suivante :

$$M = \begin{pmatrix} a_{11} & \cdots & a_{1n} & b_1 \\ \vdots & & \vdots & \vdots \\ a_{n1} & \cdots & a_{nn} & b_n \end{pmatrix}$$

Observation : La solution $(x_1, \ldots, x_n)$ du système n'est pas changée par les transformations élémentaires :

- échange de deux lignes de M
- multiplication d'une ligne de M par un nombre $\neq 0$
- ajout d'une ligne de M à une autre

L'algorithme de Gauss

En effectuant une séquence de transformations élémentaires, on transforme M en forme triangulaire supérieure :

$$M' = \begin{pmatrix} a'_{11} & a'_{12} & \cdots & \cdots & a'_{1,n-1} & a'_{1n} & b'_1 \\ 0 & a'_{22} & \cdots & \cdots & a'_{2,n-1} & a'_{2n} & b'_2 \\ 0 & 0 & \ddots & \cdots & a'_{3,n-1} & a'_{3n} & b'_3 \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & a'_{n-1,n-1} & a'_{n-1,n} & b'_{n-1} \\ 0 & 0 & \cdots & 0 & 0 & a'_{nn} & b'_n \end{pmatrix}$$

Ici tous les a'_{ij} sont zéro si $i > j$.

En pratique, il convient de combiner les deux dernières types de transformations élémentaires. Les transformations à appliquer sont alors :

- échanger deux lignes,
- ajouter un multiple d'une ligne à une autre.

L'algorithme de Gauss

En détail, pour transformer M en forme triangulaire supérieure :

1. Si la matrice ne contient qu'une seule ligne, rien à faire : terminer.
2. S'il y a au moins un coefficient non nul dans la première colonne :
   - On appelle a_{11} coefficient pivot. Si $a_{11} = 0$, échanger la première ligne avec une ligne dont le premier coefficient est $\neq 0$.
   - Éliminer tous les coefficients a_{k1} de la première colonne à part le pivot a_{11} : Ajouter $(-a_{k1}/a_{11})$-fois la première ligne à la k-ème ligne.
3. Répéter à partir de 1. avec la sous-matrice que l'on obtient en supprimant la première ligne et la première colonne de M.

En pratique, on choisit souvent comme pivot le plus grand coefficient de la première colonne, pour minimiser les erreurs d'arrondi. Pour nous il suffira de choisir un élément quelconque (non nul).

L'algorithme de Gauss

On a maintenant transformé M en forme triangulaire supérieure,

$$M' = \begin{pmatrix} a'_{11} & a'_{12} & \cdots & a'_{1,n-1} & a'_{1n} & b'_1 \\ 0 & a'_{22} & \cdots & a'_{2,n-1} & a'_{2n} & b'_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a'_{n-1,n-1} & a'_{n-1,n} & b'_{n-1} \\ 0 & 0 & \cdots & 0 & a'_{nn} & b'_n \end{pmatrix}$$

sans avoir changé la solution $(x_1, \ldots, x_n)$. Ensuite on calcule successivement les x_i :

- $x_n = b'_n / a'_{nn}$
- $x_{n-1} = (b'_{n-1} - a'_{n-1,n} x_n)/a'_{n-1,n-1}$, avec x_n déjà connu
- ...
- $x_i = \left(b'_i - \sum_{k>i} a'_{ik} x_k\right) / a'_{ii}$, avec les x_k déjà connus
- ...
- $x_1 = \left(b'_1 - \sum_{k>1} a'_{1k} x_k\right) / a'_{11}$, avec les x_k déjà connus

Si un des a'_{ii} est zéro, il n'y a pas de solution, sauf si le numérateur $b'_i - \sum_{k>i} a'_{ik} x_k$ est aussi zéro (en ce cas la solution pour x_i n'est pas unique).

L'algorithme de Gauss

Exemple : On cherche la solution $\vec{x}$ du système d'équations $A\vec{x} = \vec{b}$ avec

$$A = \begin{pmatrix} 2 & 4 & -2 & 1 \\ 1 & 2 & 3 & 4 \\ -1 & 1 & 2 & 2 \\ 0 & 3 & -2 & 0 \end{pmatrix}, \qquad \vec{b} = \begin{pmatrix} 0 \\ -3 \\ 2 \\ 0 \end{pmatrix}$$

- Au début :

$$M = \begin{pmatrix} 2 & 4 & -2 & 1 & 0 \\ 1 & 2 & 3 & 4 & -3 \\ -1 & 1 & 2 & 2 & 2 \\ 0 & 3 & -2 & 0 & 0 \end{pmatrix}$$

- Première colonne : le pivot est 2. On ajoute donc $(-1/2)$-fois la première ligne à la deuxième et $(1/2)$-fois la première ligne à la troisième. Pour la quatrième ligne il n'y a rien à faire.

L'algorithme de Gauss

$$M' = \begin{pmatrix} 2 & 4 & -2 & 1 & 0 \\ 0 & 0 & 4 & \frac{7}{2} & -3 \\ 0 & 3 & 1 & \frac{5}{2} & 2 \\ 0 & 3 & -2 & 0 & 0 \end{pmatrix}$$

- Deuxième colonne : on ne peut pas prendre 0 comme pivot. Alors d'abord on échange la deuxième ligne ave la troisième :

$$M' = \begin{pmatrix} 2 & 4 & -2 & 1 & 0 \\ 0 & 3 & 1 & \frac{5}{2} & 2 \\ 0 & 0 & 4 & \frac{7}{2} & -3 \\ 0 & 3 & -2 & 0 & 0 \end{pmatrix}$$

  Puis, rien à faire pour la troisième ligne. Ajouter (-1)-fois la deuxième à la quatrième pour éliminer M'_{42}.

L'algorithme de Gauss

$$M' = \begin{pmatrix} 2 & 4 & -2 & 1 & 0 \\ 0 & 3 & 1 & \frac{5}{2} & 2 \\ 0 & 0 & 4 & \frac{7}{2} & -3 \\ 0 & 0 & -3 & -\frac{5}{2} & -2 \end{pmatrix}$$

- Troisième colonne : le pivot est 4. Ajouter (3/4) de la troisième ligne à la quatrième :

$$M' = \begin{pmatrix} 2 & 4 & -2 & 1 & 0 \\ 0 & 3 & 1 & \frac{5}{2} & 2 \\ 0 & 0 & 4 & \frac{7}{2} & -3 \\ 0 & 0 & 0 & \frac{1}{8} & -\frac{17}{4} \end{pmatrix}$$

Maintenant M' est triangulaire supérieure et on peut calculer la solution $\vec{x}$:

- $\frac{1}{8}\, x_4 = -\frac{17}{4} \qquad \Rightarrow \; x_4 = -34$
- $4\, x_3 + \frac{7}{2}\, x_4 = 4\, x_3 - 119 = -3 \qquad \Rightarrow \; x_3 = 29$
- $3\, x_2 + x_3 + \frac{5}{2}\, x_4 = 3\, x_2 + 29 - 85 = 2 \qquad \Rightarrow \; x_2 = \frac{58}{3}$
- $2\, x_1 + 4\, x_2 - 2\, x_3 + x_4 = 2\, x_1 + \frac{232}{3} - 58 - 34 = 0 \qquad \Rightarrow \; x_1 = \frac{22}{3}$

L'algorithme de Gauss

Exercices

Dérouler à la main l'algorithme de Gauss pour les systèmes d'équations suivants :

- $$\begin{aligned} 5x_1 + 3x_2 - x_3 &= 0 \\ -2x_1 + 2x_2 - 4x_3 &= 1 \\ -x_2 + x_3 &= 3 \end{aligned}$$
- $$\begin{aligned} 2\,x_2 - 2\,x_3 + 2\,x_4 &= 4 \\ -x_1 + 3\,x_2 + 4\,x_3 + 2\,x_4 &= 2 \\ x_1 - x_2 - 3\,x_4 &= 2 \\ x_1 + x_2 - 2\,x_3 &= 0 \end{aligned}$$

Exercices

- Réaliser une fonction `joindre(A, b)` dont les arguments sont une matrice $n \times n$ `A` (avec les coefficients a_{ij}) et un vecteur `b` (avec les composantes b_i). La valeur de retour sera la matrice `M` définie par

$$M = \begin{pmatrix} a_{11} & \cdots & a_{1n} & b_1 \\ \vdots & & \vdots & \vdots \\ a_{n1} & \cdots & a_{nn} & b_n \end{pmatrix}.$$

- Réaliser deux fonctions `echanger(M, i, j)` et `ajouter_multiple(M, i, j, r)` pour effectuer les deux types de transformations nécessaires pour l'algorithme de Gauss : l'échange des lignes `i` et `j` de la matrice `M`, et l'addition de `r`-fois la `i`-ème ligne à la `j`-ème.
- Réaliser une fonction `triangulariser(M)` qui transforme une matrice $n \times (n+1)$ en forme triangulaire supérieure.
- Réaliser une fonction `gauss(A, b)` dont les arguments sont les mêmes que celles de la fonction `joindre` et qui retourne le vecteur des solutions $(x_1, \ldots, x_n)$. S'il n'y a pas de solution, elle produira un `ValueError`. Si la solution n'est pas unique, elle retournera une des possibles solutions.
- Eprouver votre code dans un script Python. Verifier en particulier votre solution de l'exercice précédente

La décomposition LU

Soit A une matrice carrée. On peut représenter les opérations élémentaires sur A par des multiplications avec certaines matrices spéciales.

Ajouter r fois la j-ème à l'i-ème ligne ($i \neq j$)

$$A \rightarrow T_{(i,j;r)}A$$

$T_{(i,j;r)}$ = la matrice identité avec le (i,j) coefficient remplacé par r :

$$T_{(i,j;r)} = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & r & & \\ & & & \ddots & \\ & & & & 1 \end{pmatrix}$$

La décomposition LU

Soit A une matrice carrée. On peut représenter les opérations élémentaires sur A par des multiplications avec certaines matrices spéciales.

Échange de l'i-ème et la j-ème ligne :

$$A \rightarrow P_{(i,j)}A$$

$P_{(i,j)}$ = la matrice identité avec les lignes i et j échangées :

$$P_{(i,j)} = \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & 0 & & & 1 & \\ & & & & 1 & & & \\ & & & & & \ddots & & \\ & & & & & & 1 & \\ & & & 1 & & & 0 & \\ & & & & & & & 1 \end{pmatrix}$$

La décomposition LU

La décomposition LU d'une matrice $n \times n$ A est la décomposition

$$PA = LU$$

- U est une matrice $n \times n$ triangulaire supérieure
  (zéro au-dessous de la diagonale principale)
- L est une matrice $n \times n$ triangulaire inférieure
  (zéro au-dessus de la diagonale principale)
- P est une matrice de permutation $n \times n$
  (un produit des matrices du type $P_{(i,j)}$).

La matrice U est le résultat de l'application de l'algorithme de Gauss à la matrice A. En prenant note des transformations effectuées pendant le déroulement de l'algorithme, on peut aussi déterminer L et P.

La décomposition LU

Rappel : pour éliminer la j-ème colonne, l'algorithme de Gauss procède en deux pas,

- (Potentiellement) un échange des lignes $P_{(k,j)}$:
  - Si $a_{jj} = 0$, on cherche une autre ligne a_{k*} avec $a_{kj} \neq 0$ et on l'échange avec la j-ème ligne. Opération réprésenté par $P_{(k,j)}$ avec $k > j$.
  - Si déjà $a_{jj} \neq 0$, pas besoin de changer le pivot. Opération "nulle" réprésentée par $P_{(j,j)} = \mathbb{1}$.
- Ensuite, $(n - j - 1)$ opérations du type $T_{(i,j;r)}$ (où $i = j + 1 \ldots n$) :
  Ajout de r fois de la ligne j du pivot à la ligne i)

Ces dernières peuvent être regroupées dans une matrice de Frobenius $F_{(j;r_{j+1}\ldots r_n)}$ données par $F_{(j;r_{j+1}\ldots r_n)} = T_{(j+1,j;r_{j+1})}\, T_{(j+2,j;r_{j+2})} \cdots T_{(n,j;r_n)}$, ou explicitement (→ exercices)

$$F_{(j;r_{j+1}\ldots r_n)} = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ & & & r_{j+1} & 1 & & \\ & & & r_{j+2} & & \ddots & \\ & & & \vdots & & & \\ & & & r_n & & & 1 \end{pmatrix}$$

La décomposition LU

On peut alors représenter les transformations de l'algorithme de Gauss schématiquement comme suit :

$$\boxed{F_{(n-1;*)}\, P_{(*,n-1)}\, F_{(n-2;*)} \cdots P_{(*,2)}\, F_{(1;*)}\, P_{(*,1)}\, A = U}$$

- A est la matrice à transformer
- U est le résultat de l'algorithme de Gauss, une matrice triangulaire supérieure
- Les $P_{(i,j)}$ correspondent aux changements de pivot ($P_{(i,j)} = \mathbb{1}$ où pas de changement nécessaire)
- Les $F_{(j;r_{j+1}\ldots r_n)}$ sont les matrices de Frobenius pour l'élimination de la j-ème colonne

On insère $\mathbb{1}$ dans l'équation ci-dessus, en utilisant $P_{(i,j)}P_{(i,j)} = \mathbb{1}$:

$$\boxed{\begin{aligned} &F_{(n-1;*)}\, P_{(*,n-1)}\, F_{(n-2;*)} \cdots P_{(*,2)}\, F_{(1;*)}\, P_{(*,2)}\, P_{(*,3)} \cdots P_{(*,n-1)} \cdot \\ &\quad \cdot P_{(*,n-1)} \cdots P_{(*,3)}\, P_{(*,2)}\, P_{(*,1)}\, A = U \end{aligned}}$$

On peut montrer : La première ligne

$$\hat{L} = F_{(n-1;*)}\, P_{(*,n-1)}\, F_{(n-2;*)} \cdots P_{(*,2)}\, F_{(1;*)}\, P_{(*,2)}\, P_{(*,3)} \cdots P_{(*,n-1)}$$

est triangulaire inférieure.

La décomposition LU

On est arrivé à la représentation suivante de la triangularisation de la matrice A :

$$\boxed{\hat{L} P_{(*,n-1)} \cdots P_{(*,3)} P_{(*,2)} P_{(*,1)} A = U}$$

où

$$\hat{L} = F_{(n-1;*)} P_{(*,n-1)} F_{(n-2;*)} \cdots P_{(*,2)} F_{(1;*)} P_{(*,2)} P_{(*,3)} \cdots P_{(*,n-1)}$$

est une matrice triangulaire inférieure.

L'inverse d'une matrice triangulaire inférieure est également triangulaire inférieure. En définissant

$$\begin{aligned} L &\equiv \hat{L}^{-1} \\ P &\equiv P_{(*,n-1)} \cdots P_{(*,2)} P_{(*,1)} \end{aligned}$$

et en multipliant avec L à la gauche aux deux cotés on arrive à la forme souhaitée de la décomposition LU :

$$\boxed{PA = LU}$$

.

La décomposition LU

Pour calculer la décomposition LU, $PA = LU$:

- $U =$ le résultat de l'algorithme de Gauss appliqué à A
- $P =$ le produit de tous $P_{(i,j)}$ (échanges de lignes) qui apparaissent pendant le déroulement de la méthode de Gauss. Démarrer l'algorithme de Gauss avec $P = \mathbb{1}$, et pour tout échange de lignes effectué, multiplier P à gauche par $P_{(i,j)}$.
- $L = \hat{L}^{-1} = P\, P_{(*,1)}\, F^{-1}_{(1;*)}\, P_{(*,2)}\, F^{-1}_{(2;*)} \cdots P_{(*,n-1)}\, F^{-1}_{(n-1;*)}$
  avec les $F_{(i;*)}$ correspondant aux éliminations de colonnes avec la méthode de Gauss. Démarrer l'algorithme de Gauss avec $L = \mathbb{1}$; pour tous échanges de lignes, multiplier L à droite par $P_{(i,j)}$; pour toute colonne, multiplier à droite par le $F^{-1}_{(i;*)}$ correspondant ; enfin multiplier à gauche par P.

La décomposition LU

Exemple : On cherche la décomposition LU de la matrice

$$A = \begin{pmatrix} 2 & 4 & -2 & 1 \\ 1 & 2 & 3 & 4 \\ -1 & 1 & 2 & 2 \\ 0 & 3 & -2 & 0 \end{pmatrix}$$

- Au début : $U = A$, $L = \mathbb{1}$, $P = \mathbb{1}$

La décomposition LU

- Première colonne :
  pas de changement de pivot, $P_{(i,j)} = P_{(1,1)} = \mathbb{1}$, $P \leftarrow P$, $L \leftarrow L$
  ajouter $(-1/2)$(première ligne) à (deuxième ligne) :

$$L \leftarrow L \begin{pmatrix} 1 & & & \\ \frac{1}{2} & 1 & & \\ & & 1 & \\ & & & 1 \end{pmatrix}$$

  ajouter $(1/2)$(première ligne) à (troisième ligne) :

$$L \leftarrow L \begin{pmatrix} 1 & & & \\ & 1 & & \\ -\frac{1}{2} & & 1 & \\ & & & 1 \end{pmatrix}$$

  rien à faire pour quatrième ligne : $L \leftarrow L$.
  Maintenant

$$L = \begin{pmatrix} 1 & & & \\ \frac{1}{2} & 1 & & \\ -\frac{1}{2} & & 1 & \\ & & & 1 \end{pmatrix}, \qquad P = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{pmatrix}.$$

La décomposition LU

- Deuxième colonne : Maintenant

$$U = \begin{pmatrix} 2 & 4 & -2 & 1 \\ 0 & 0 & 4 & \frac{7}{2} \\ 0 & 3 & 1 & \frac{5}{2} \\ 0 & 3 & -2 & 0 \end{pmatrix}$$

$U_{22} = 0$: il faut changer le pivot. On échange donc la deuxième et la troisième ligne :

$$U = \begin{pmatrix} 2 & 4 & -2 & 1 \\ 0 & 3 & 1 & \frac{5}{2} \\ 0 & 0 & 4 & \frac{7}{2} \\ 0 & 3 & -2 & 0 \end{pmatrix}$$

$$P_{(i,j)} = P_{(2,3)} = \begin{pmatrix} 1 & & & \\ & 0 & 1 & \\ & 1 & 0 & \\ & & & 1 \end{pmatrix}, \quad P \leftarrow P_{(2,3)}P, \quad L \leftarrow LP_{(2,3)}.$$

La décomposition LU

Maintenant

$$L = \begin{pmatrix} 1 & & & \\ \frac{1}{2} & 0 & 1 & \\ -\frac{1}{2} & 1 & 0 & \\ & & & 1 \end{pmatrix}, \qquad P = \begin{pmatrix} 1 & & & \\ & 0 & 1 & \\ & 1 & 0 & \\ & & & 1 \end{pmatrix}.$$

$$U = \begin{pmatrix} 2 & 4 & -2 & 1 \\ 0 & 3 & 1 & \frac{5}{2} \\ 0 & 0 & 4 & \frac{7}{2} \\ 0 & 3 & -2 & 0 \end{pmatrix}$$

Rien à faire pour la troisième ligne : $L \leftarrow L$.
Ajouter (-1) fois (deuxième ligne) à (quatrième ligne) :

$$L \leftarrow L \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & 1 & & 1 \end{pmatrix} \qquad \text{donc} \qquad L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 1 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

La décomposition LU

- Troisième colonne : Maintenant

$$U = \begin{pmatrix} 2 & 4 & -2 & 1 \\ 0 & 3 & 1 & \frac{5}{2} \\ 0 & 0 & 4 & \frac{7}{2} \\ 0 & 0 & -3 & -\frac{5}{2} \end{pmatrix}$$

pas de changement de pivot : $P_{(i,j)} = P_{(3,3)} = \mathbb{1}$, $P \leftarrow P$, $L \leftarrow L$
ajouter (3/4)(troisième ligne) à (quatrième ligne) :

$$L \leftarrow L \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -\frac{3}{4} & 1 \end{pmatrix} \quad \text{donc} \quad L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 1 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 \\ 0 & 1 & -\frac{3}{4} & 1 \end{pmatrix}$$

- Finalement : $L \leftarrow PL$.

La décomposition LU

Enfin :

$$U = \begin{pmatrix} 2 & 4 & -2 & 1 \\ 0 & 3 & 1 & \frac{5}{2} \\ 0 & 0 & 4 & \frac{7}{2} \\ 0 & 0 & 0 & \frac{1}{8} \end{pmatrix}$$

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 \\ \frac{1}{2} & 0 & 1 & 0 \\ 0 & 1 & -\frac{3}{4} & 1 \end{pmatrix}$$

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Exercices

- Montrer que la matrice de Frobenius $F_{(j;r_{j+1}\dots r_n)}$ est le produit des matrices $T_{(j+1,j;\,r_{j+1})}\cdots T_{(n,j;\,r_n)}$.
- Montrer que l'inverse de la matrice de Frobenius $F_{(j;r_{j+1}\dots r_n)}$ est la matrice de Frobenius $F_{(j;-r_{j+1}\dots -r_n)}$.
- Réaliser deux fonctions `P(i, j)` et `T(i, j, r)` qui retournent respectivement la matrice $P_{(i,j)}$ et $T_{(i,j;r)}$.
- En modifiant votre code de la fonction `triangulariser()`, réaliser une fonction `decomp_LU(M)` qui transforme son objet en forme trianguaire supérieure et qui retourne deux tableaux, les matrices L et P de la décomposition LU.
- Trouver des simples algorithmes pour calculer le déterminant d'une matrice triangulaire supérieure, d'une matrice triangulaire inférieure et d'une matrice de permutation.
- Réaliser une fonction `det(M)` qui retourne le déterminant d'une matrice carrée.

La décomposition LU : Applications

Une fois calculée la décomposition LU d'une matrice carrée A, on peut facilement resoudre tous systèmes d'équations de la forme $A\vec{x} = \vec{b}$ pour un $\vec{b}$ quelconque :

$$A\vec{x} = \vec{b} \Leftrightarrow LU\vec{x} = P\vec{b} \Leftrightarrow L\vec{y} = \vec{b}' \quad \text{avec } \vec{y} = U\vec{x},\ \vec{b}' = P\vec{b}$$

- Déterminer $\vec{y}$ à partir de $L\vec{y} = \vec{b}'$:

$$\begin{aligned} y_1 &= b'_1/L_{11} \\ y_2 &= (b'_2 - L_{21}y_1)/L_{22} \\ &\vdots \\ y_n &= (b'_n - \sum_{k<n} L_{nk}y_k)/L_{nn} \end{aligned}$$

- Calculer $\vec{x}$ à partir de $U\vec{x} = \vec{y}$ (voir dernière étape de la méthode de Gauss) :

$$\begin{aligned} x_n &= y_n/U_{nn} \\ x_{n-1} &= (y_{n-1} - U_{n-1,n}x_n)/U_{n-1,n-1} \\ &\vdots \\ x_1 &= (y_1 - \sum_{k>1} U_{1k}x_k)/U_{11} \end{aligned}$$

La décomposition LU : Applications

```
def resoudre(L, U, P, b): # trouve x dans LUx = Pb
    n = U.shape[0] # les matrices sont n fois n
    bprime = P.dot(b)       # le vecteur b' = Pb
    y = np.zeros(n)      # le vecteur y
    for i in range(n):  # initialiser y
        sig = 0
        for k in range(i): # k entre 0 et i-1
            sig += L[i, k] * y[k]
        y[i] = (bprime[i] - sig)/L[i, i]
    x = np.zeros(n)      # le vecteur x
        for i in range(n-1, -1, -1): # i entre 0 and n-1,
            sig = 0                  #          à l'envers
            for k in range(i+1, n):    # k entre i+1 et n-1
                sig += U[i, k] * x[k]
            x[i] = (y[i] - sig)/U[i, i]
    return x
```

La décomposition LU : Applications

Calcul de l'inverse : La décomposition LU permet de calculer le vecteur $\vec{x}$ dans le système d'équations $A\vec{x} = \vec{b}$. En appliquant cette méthode n fois avec $\vec{b} =$ les n vecteurs de colonne d'une matrice B, on obtient n vecteurs de solution : les vecteurs de colonne d'une matrice X qui vérifient l'équation matricielle

$$AX = B\,.$$

Si $B = \mathbb{1}$, alors $X = A^{-1}$ l'inverse de la matrice A.

```
def inverse(A): # trouve l'inverse de A
    U = A.copy()    # pour ne pas changer A
    n = U.shape[0]
    P, L = decomp_LU(U)
    Ainv = np.zeros((n, n))
    for i in range(n):
        e_i = np.zeros(n)
        e_i[i] = 1
        Ainv_i = resoudre(L, U, P, e_i)
        for k in range(n):
            Ainv[k, i] = Ainv_i[k]
    return Ainv
```

L'efficacité de la méthode de Gauss et la décomposition LU

- La complexité en temps de la méthode de Gauss et de sa variante la décomposition LU est $\mathcal{O}(n^3)$ pour une matrice $n \times n$.
- Pareil pour calculer l'inverse et le déterminant avec la décomposition LU.
- D'autres algorithmes, souvent plus efficaces, existent pour les matrices de forme spéciale (matrices creuses. . .).

Remarques concernant notre implémentation :

- Pour minimiser le besoin de mémoire et du temps de calcul, il convient de ne pas paramétriser les permutations avec une matrice P mais avec un vecteur $\vec{p}$ qui contient les mêmes informations.
- Pour minimiser le temps du calcul, on ne construira pas L par une séquence de multiplications de matrices (qui coûtent chères, $\sim n^3$ opérations) mais plus directement.
- D'autres optimisations sont possibles pour les applications en pratique.

Les valeurs propres et les vecteurs propres d'une matrice

Soit A une matrice réelle $n \times n$ qui est symétrique : $A^T = A$.

Theorème : Il existe une matrice orthogonale S tel que $S^T A S \equiv D$ est une matrice diagonale.

(Rappel : S orthogonale $\overset{def.}{\Leftrightarrow}$ $S^T S = \mathbb{1}$; D diagonale $\overset{def.}{\Leftrightarrow}$ D zéro hors la diagonale principale .)

Dans ce cas les coefficients de la diagonale principale de D sont les valeurs propres de A, et les colonnes de S sont les vecteurs propres de A.

(Rappel : $\lambda \in \mathbb{R}$ valeur propre de $A \overset{def.}{\Leftrightarrow} \exists\ \vec{v} \in \mathbb{R}^n$ non nul, le *vecteur propre*, tel que $A\vec{v} = \lambda\vec{v}$.)

Problème :

Soit A une matrice symétrique donnée. On cherche ses valeurs propres et ses vecteurs propres correspondants. Équivalent : on cherche S et D.

L'algorithme QR

Un algorithme classique pour trouver les valeurs propres et les vecteurs propres d'une matrice est l'algorithme QR. Ici on va discuter une version simple. Des algorithmes plus modernes sont généralement plus efficaces et plus stables, mais aussi plus compliqués.

L'algorithme QR repose sur la décomposition QR d'une matrice carrée A,

$$A = QR$$

où Q est une matrice orthogonale et R est une matrice triangulaire supérieure. On définit la suite A_k par

$$A_0 = A\,, \qquad A_{n+1} = R_n Q_n$$

avec $A_n = Q_n R_n$ la décomposition QR de A_n. On peut montrer que cette suite converge, sous certains conditions, vers une matrice triangulaire dont les coefficients diagonaux sont les valeurs propres de A. La matrice S est le produit de toutes les matrices Q_n.

L'algorithme QR

Algorithme :

1. Démarrer avec $S = \mathbb{1}$.
2. Si A est triangulaire supérieure, terminer. Valeurs propres = coefficients sur la diagonale principale de A, vecteurs propres = colonnes de S.
3. Trouver la décomposition QR de A, $A = QR$.
4. Répéter à partir de (2) avec $A \leftarrow RQ$ et $S \leftarrow SQ$.

La décomposition QR

Pour implémenter l'algorithme QR, il faut savoir calculer la décomposition QR d'une matrice A. Il y a plusieurs méthodes ; un simple algorithme est la méthode modifiée de Gram-Schmidt :

- L'idée est de orthonormaliser les colonnes de A par une suite de transformations qui peuvent être representées par des matrices triangulaires supérieures. Si alors $\vec{v}_i$ et $\vec{v}_j$ sont deux colonnes différentes, on souhaite que $\vec{v}_i \cdot \vec{v}_j = 0$ et $\vec{v}_i \cdot \vec{v}_i = \vec{v}_j \cdot \vec{v}_j = 1$.
- On définit la projection de $\vec{v}$ sur $\vec{u}$:

$$\mathrm{pr}_{\vec{u}}\vec{v} \equiv \frac{\vec{v} \cdot \vec{u}}{||\vec{u}||^2}\vec{u}$$

- Pour tout vecteur de colonne $\vec{v}_i$:
  - Normaliser, $\vec{v}_i \leftarrow \vec{v}_i/||\vec{v}_i||$ (si $\vec{v}_i$ n'est pas zéro)
  - Soustraire de chacune des colonnes derrière $\vec{v}_i$ sa projection sur $\vec{v}_i$, $\vec{v}_j \leftarrow \vec{v}_j - \mathrm{pr}_{\vec{v}_i}\vec{v}_j \quad \forall\, j > i$
  - Maintenant $\vec{v}_i$ est orthogonal à tout $\vec{v}_j$ avec $j > i$ (et à toutes ses combinaisons linéaires)
- Ainsi on obtient les vecteurs de colonne de Q. Poser $R = Q^{-1}A = Q^T A$.

La décomposition QR

Exemple (Méthode modifiée de Gram-Schmidt) :

$$A = Q = \begin{pmatrix} 2 & 0 & 1 \\ 2 & 4 & -1 \\ -1 & 2 & 3 \end{pmatrix}$$

Normaliser la première colonne : $\sqrt{2^2 + 2^2 + (-1)^2} = 3$, alors

$$Q \leftarrow \begin{pmatrix} \frac{2}{3} & 0 & 1 \\ \frac{2}{3} & 4 & -1 \\ -\frac{1}{3} & 2 & 3 \end{pmatrix}$$

La produit scalaire de la 1ère avec la 2ème colonne est $\frac{2}{3} \times 0 + \frac{2}{3} \times 4 + (-\frac{1}{3}) \times 2 = 2$. Il faut donc soustraire $2\times$(1ère colonne) de la 2ème. De même, le produit scalaire de la 1e colonne avec la 3ème est $\frac{2}{3} \times 1 + \frac{2}{3} \times (-1) + (-\frac{1}{3}) \times 3 = -1$, on soustrait alors $(-1)\times$(1ère colonne) de la 3ème.

$$Q \leftarrow \begin{pmatrix} \frac{2}{3} & -\frac{4}{3} & \frac{5}{3} \\ \frac{2}{3} & \frac{8}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{8}{3} & \frac{8}{3} \end{pmatrix}.$$

Maintenant la première colonne est normalisée et orthogonale aux deux autres.

La décomposition QR

Exemple (Méthode modifiée de Gram-Schmidt) :

Normaliser la deuxième colonne : $\sqrt{(-4/3)^2 + (8/3)^2 + (8/3)^2} = 4$, alors

$$Q \leftarrow \begin{pmatrix} \frac{2}{3} & -\frac{1}{3} & \frac{5}{3} \\ \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{2}{3} & \frac{8}{3} \end{pmatrix}$$

La produit scalaire de la 2ème avec la 3ème colonne est 2. Il faut donc soustraire la 2ème de la 3ème.

$$Q \leftarrow \begin{pmatrix} \frac{2}{3} & -\frac{1}{3} & 2 \\ \frac{2}{3} & \frac{2}{3} & -1 \\ -\frac{1}{3} & \frac{2}{3} & 2 \end{pmatrix}$$

Maintenant les premières deux colonnes sont normalisées et orthogonales à eux-mêmes et à la troisième.

Il ne reste que normaliser cette dernière :

$$Q \leftarrow \begin{pmatrix} \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{2}{3} & \frac{2}{3} \end{pmatrix}$$

La décomposition QR

Exemple (calcul de R) :

On a

$$Q^{-1} = Q^T = \begin{pmatrix} \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} \end{pmatrix}$$

et alors

$$R = Q^{-1}A = \begin{pmatrix} \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} \end{pmatrix} \begin{pmatrix} 2 & 0 & 1 \\ 2 & 4 & -1 \\ -1 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 3 & 2 & -1 \\ 0 & 4 & 1 \\ 0 & 0 & 3 \end{pmatrix}$$

La décomposition QR

Deux fonctions auxiliaires :

```
import numpy as np

# pour calculer la norme de v:
def norme(v):
    return np.sqrt(np.sum(v**2))

# pour calculer la projection de w sur v:
def projeter(v, w, epsilon=1.0E-10):
    n = v.shape[0]
    vnorme = norme(v)
    if vnorme < epsilon:  # proj. sur vect. nul = vect. nul
        return np.zeros(n)
    prod = np.dot(v, w)   # le produit scalaire entre v et w
    res = np.copy(v)      # le vecteur résultat
    res *= prod
    res /= vnorme**2
    return res
```

La décomposition QR

Orthogonalisation de Gram-Schmidt :

```
def gram_schmidt(A, epsilon=1.0E-10):
  n = A.shape[0]
  resultat = np.copy(A)  # on y mettra le résultat
  for i in range(n):     # Gram-Schmidt modifiée:
    v = resultat[:, i]  # pour tout vecteur de colonne v:
    vnorm = norme(v)
    if vnorm >= epsilon: #  normaliser (sauf si nul)
      v /= vnorm
    for j in range(i+1, n): # de toute colonne après v:
                            # soustraire sa projection sur v
      resultat[:, j] -= projeter(v, resultat[:, j], epsilon)
  return resultat
```

Décomposition QR :

```
def decomp_QR(A, epsilon=1.0E-10):
    Q = gram_schmidt(A, epsilon)
    R = np.dot(Q.T, A)
    return Q, R
```

L'algorithme QR

Exercices

Réaliser une fonction `diagonaliser(A, epsilon=1.0E-5, maxit=30)` qui calcule les valeurs propres et les vecteurs propres de son argument la matrice `A`. Servez-vous des fonctions déjà réalisées ainsi que de la fonction `decomp_QR` ci-dessus. La fonction testera d'abord si `A` est une matrice symétrique. Sinon elle termine avec un `ValueError`. Puis elle déroulera l'algorithme QR jusqu'à ce que RQ soit triangulaire supérieure à une erreur de $\mathcal{O}(\texttt{epsilon})$ près (trouvez une condition appropriée). Si elle dépasse `maxit` itérations, elle terminera avec un `RuntimeError`. Enfin elle retournera un tableau qui contient les vecteurs propres et un autre qui contient les valeurs propres.

Algèbre linéaire avec SciPy

La bibliothèque SciPy contient des méthodes pour l'algorithme de Gauss, la décomposition LU, la décomposition QR et de nombreuses autres : décomposition en valeurs singulières, décomposition de Cholesky, fonctions matricielles, méthodes optimisées pour des matrices spéciales...

Pour les vraies applications dans la physique numérique, il est préférable de se servir de ces méthodes optimisées au lieu de nos implémentations du cours « fait maison ».

On les trouve dans la sous-bibliothèque `scipy.linalg` :

```
import numpy as np
import scipy.linalg as la
A = np.array([[5, 3, -1], [3, 2, -4], [-1, -4, 0]])
Ainv = la.inv(A)            # inverse
d = la.det(A)               # déterminant
vals, vecs = la.eig(A)      # valeurs/vecteurs propres
Pinv, L, U = la.lu(A)       # déc. LU: Pinv^(-1) A = L U
Q, R = la.qr(A)             # décomposition QR
```

Le travail de calcul dans cette bibliothèque reste sur des routines en C, C++ et FORTRAN optimisées qui sont beaucoup plus vite que des routines en Python.

Application exemplaire : la regression non linéaire

Description du problème :

Soient $(x_1, y_1) \ldots (x_n, y_n)$ des données (par exemple des données expérimentales) et $f(x; \alpha_1, \ldots \alpha_p)$ une famille de fonctions qui dépend des paramètres inconnus $\alpha_1, \ldots \alpha_p$. On cherche les valeurs des paramètres telles que la fonction f correspondante décrit le mieux les données, c.à.d. $f(x_i; \vec{\alpha}) \approx y_i$.

Exemple : L'activité d'une source radioactive en fonction du temps est décrite par la fonction

$$f(t; \alpha_1, \alpha_2) = \alpha_1 \, e^{-\alpha_2 t}$$

où α_1 est l'activité initiale et α_2 est la durée de vie moyenne de la substance. On mésure

temps [h]	activité [10^{15} Bq]
0	2.02
5	1.95
10	1.93
20	1.88
40	1.73
80	1.50

Qu'est-ce que l'on peut conclure pour les valeurs de α_1 et α_2 ?

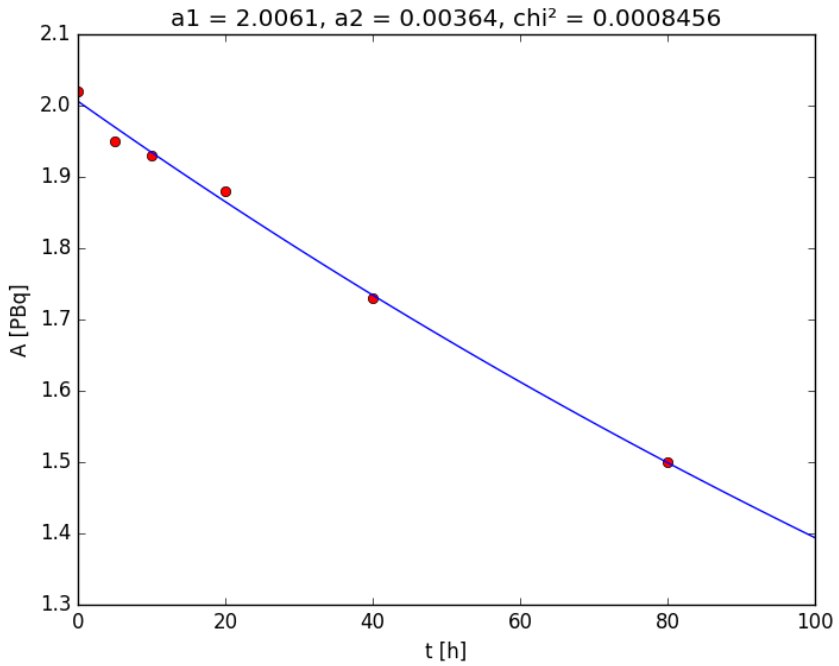
Regression non linéaire : Méthode des moindres carrés

Méthode standard : Méthode des moindres carrés.

On cherche les valeurs $\alpha_1 \dots \alpha_p$ telles que χ^2, défini par

$$\chi^2 \equiv \sum_{i=1}^{n} r_i(\alpha_1 \dots \alpha_p)^2 \qquad \text{où } r_i(\alpha_1, \dots \alpha_p) \equiv f(x_i;\, \alpha_1 \dots \alpha_p) - y_i$$

est minimisé. Ces valeurs donnent le meilleur ajustement des paramètres aux données.



Généralisations : incertitudes variables (nécessite pondérations des contributions respectives à χ^2), plusieurs composantes de x (le principe reste le même), ...

Regression non linéaire : Méthode des moindres carrés

Comment minimiser χ^2 en fonction de $\alpha_1 \ldots \alpha_p$?

Problème d'optimisation non linéaire. Plusieurs approches possibles.

- Par exemple, par la méthode de Newton généralisée. Un minimum de χ^2 est un zéro de $\nabla\chi^2$ que l'on peut chercher avec la généralisation de la méthode de Newton à plusieurs variables.
- Ici on va discuter une variante simplifiée, l'algorithme de Gauss-Newton.
- (Très commun en pratique : une variante plus complexe, l'algorithme de Levenberg-Marquardt.)

Regression non linéaire : L'algorithme de Gauss-Newton

- Méthode de Newton (rappel) : pour trouver un zéro de $g(x)$, on itère

$$x \leftarrow x - \frac{g(x)}{g'(x)}$$

- Pour trouver un point critique (et donc un minimum potentiel) de $g(x)$, on cherche un zéro de $g'(x)$: on itère

$$x \leftarrow x - \frac{g'(x)}{g''(x)}$$

- Généralisation à plusieurs variables : soit $g(\vec{\alpha})$ une fonction de p variables $\vec{\alpha}$. Pour trouver un point critique, itérer

$$\vec{\alpha} \leftarrow \vec{\alpha} - H^{-1}(\vec{\alpha})\vec{\nabla} g(\vec{\alpha})$$

où la matrice hesséenne $H(\vec{\alpha})$ est la généralisation de la dérivée seconde pour plusieurs variables,

$$H = \begin{pmatrix} \frac{\partial^2 g}{\partial \alpha_1^2} & \frac{\partial^2 g}{\partial \alpha_1 \partial \alpha_2} & \frac{\partial^2 g}{\partial \alpha_1 \partial \alpha_3} & \cdots & \frac{\partial^2 g}{\partial \alpha_1 \partial \alpha_p} \\ \frac{\partial^2 g}{\partial \alpha_2 \partial \alpha_1} & \frac{\partial^2 g}{\partial \alpha_2^2} & \frac{\partial^2 g}{\partial \alpha_2 \partial \alpha_3} & \cdots & \frac{\partial^2 g}{\partial \alpha_2 \partial \alpha_p} \\ \vdots & & & & \vdots \\ \frac{\partial^2 g}{\partial \alpha_p \partial \alpha_1} & \frac{\partial^2 g}{\partial \alpha_p \partial \alpha_2} & \frac{\partial^2 g}{\partial \alpha_p \partial \alpha_3} & \cdots & \frac{\partial^2 g}{\partial \alpha_p^2} \end{pmatrix}$$

Regression non linéaire : L'algorithme de Gauss-Newton

L'algorithme de Gauss-Newton évite le calcul de H en profitant du fait que la fonction à minimiser est une somme des carrés :

$$\chi^2(\alpha_1 \dots \alpha_p) = r_1(\vec{\alpha})^2 + r_2(\vec{\alpha})^2 + \dots + r_n(\vec{\alpha})^2 = \vec{r} \cdot \vec{r}$$

avec les n résidus

$$r_i(\vec{\alpha}) = f(x_i; \vec{\alpha}) - y_i$$

Alors

$$\vec{\nabla}\chi^2 = \begin{pmatrix} 2\, r_1 \frac{\partial r_1}{\partial \alpha_1} + 2\, r_2 \frac{\partial r_2}{\partial \alpha_1} + \dots + 2\, r_n \frac{\partial r_n}{\partial \alpha_1} \\ 2\, r_1 \frac{\partial r_1}{\partial \alpha_2} + 2\, r_2 \frac{\partial r_2}{\partial \alpha_2} + \dots + 2\, r_n \frac{\partial r_n}{\partial \alpha_2} \\ \vdots \\ 2\, r_1 \frac{\partial r_1}{\partial \alpha_p} + 2\, r_2 \frac{\partial r_2}{\partial \alpha_p} + \dots + 2\, r_n \frac{\partial r_n}{\partial \alpha_p} \end{pmatrix} = 2\, \vec{r}\, J$$

Ici J est la matrice jacobienne $n \times p$

$$J_{ij} = \frac{\partial r_i}{\partial \alpha_j}\,.$$

De plus,

$$H_{ij} = \frac{\partial}{\partial \alpha_i} \left(\nabla \chi^2\right)_j = \frac{\partial}{\partial \alpha_i} \left(2\, \vec{r}\, J\right)_j = 2 \left(J^T\, J\right)_{ij} + 2\vec{r} \cdot \frac{\partial^2 \vec{r}}{\partial \alpha_i \partial \alpha_j} \approx 2 \left(J^T\, J\right)_{ij}$$

(approximation au premier ordre en dérivées).

Regression non linéaire : L'algorithme de Gauss-Newton

On a trouvé

$$\vec{\nabla}\chi^2 = 2\,\vec{r}\,J$$

et

$$H \approx 2\left(J^T J\right)$$

où $\vec{r}$ est le vecteur à n composantes des résidus (fonctions des paramètres α_i)

$$r_i(\vec{\alpha}) = f(x_i;\vec{\alpha}) - y_i$$

et J est la matrice jacobienne $n \times p$ de $\vec{r}$ par rapport à $\vec{\alpha}$,

$$J = \frac{\partial \vec{r}}{\partial \vec{\alpha}}\,.$$

L'itération de la méthode de Newton pour trouver un point critique de $\chi^2 = \vec{r}\cdot\vec{r}$ devient

$$\boxed{\vec{\alpha} \leftarrow \vec{\alpha} - (J^T J)^{-1}\vec{r}\,J}$$

Méthode de Gauss-Newton.

Regression non linéaire : L'algorithme de Gauss-Newton

En pratique :

- Calculer analytiquement les dérivées $\frac{\partial f}{\partial \alpha_j}$.
- Commencer avec un ensemble de paramètres $\vec{\alpha}$ au choix.
- Mettre à jour les $r_i = f(x_i; \vec{\alpha}) - y_i$ et les $J_{ij} = \frac{\partial f}{\partial \alpha_j}(x_i; \vec{\alpha})$.
- Remplacer $\vec{\alpha} \leftarrow \vec{\alpha} - (J^T J)^{-1} \vec{r} J$.
- Itérer ces dernières deux étapes jusqu'à la convergence.
  S'arrêter lorsque $||\vec{\alpha}_{\text{présent}} - \vec{\alpha}_{\text{précédent}}|| < \epsilon$.

Remarque : Pour améliorer la stabilité numérique et l'efficacité, il convient d'éviter le calcul de l'inverse de la matrice $J^T J$. Ainsi, au lieu de l'itération $\vec{\alpha}_{m+1} = \vec{\alpha}_m - (J^T J)^{-1} \vec{r} J$ on déterminera $\vec{\alpha}_{m+1}$ par la méthode de Gauss comme solution du système linéaire

$$J^T J \vec{\alpha}_{m+1} = J^T J \vec{\alpha}_m - \vec{r} J$$

(mathématiquement équivalent après multplication par $J^T J$ aux deux côtés).

Regression non linéaire : L'algorithme de Gauss-Newton

Ce code suppose que les fonctions $\frac{\partial f}{\partial \alpha_j}$ sont données dans une liste `gradf`.

```
import numpy as np
import gauss   # pour la fonction gauss() de l'exo 8.1

def gauss_newton(x, y, f, gradf, alpha0, epsilon=1.E-4):
  alpha = np.copy(alpha0) # les paramètres à ajuster
  oldalpha = alpha + 1.   # ses anciennes valeurs, pour
                          #          tester convergence

  while np.sqrt(np.sum((oldalpha - alpha)**2)) > epsilon:
    r = f(x, alpha) - y   # les résidus
    J = np.array([df(x, alpha) for df in gradf]).T
    JTJ = J.T.dot(J)      # la matrice J^T J
    oldalpha = alpha
    alpha = gauss.gauss(JTJ, JTJ.dot(alpha) - r.dot(J))

  chi2 = np.sum(r**2)     # chi^2 après minimisation

  return alpha, chi2
```

Algorithme de Gauss-Newton : Exemple d'application

Par exemple, pour le problème de la désintégration radioactive du début : on a

$$f(t;\vec{\alpha}) = \alpha_1\, e^{-\alpha_2 t}$$

et donc

$$\frac{\partial f}{\partial \alpha_1}(t;\vec{\alpha}) = e^{-\alpha_2 t}, \quad \frac{\partial f}{\partial \alpha_2}(t;\vec{\alpha}) = -t\alpha_1\, e^{-\alpha_2 t} = -t\, f(t;\vec{\alpha})\,.$$

Code :

```
def f(t, alpha):
    return alpha[0] * np.exp(-alpha[1] * t)

gradf = [ lambda t, alpha: np.exp(-alpha[1] * t),
          lambda t, alpha: -t * f(t, alpha) ]

x = np.array([0., 5., 10., 20., 40., 80.])
y = np.array([2.02, 1.95, 1.93, 1.88, 1.73, 1.50])

alpha,chi2 = gauss_newton(x, y, f, gradf, np.array([1.,0.]))
```

Remarque : Ce problème particulier peut aussi se traiter (plus facilement) comme problème de régression *linéaire* car $\ln f = \ln \alpha_1 - \alpha_2 t$; il faut alors n'ajuster qu'une droite aux logarithmes des valeurs mesurées.

Algorithme de Gauss-Newton

Exercices

Réaliser un programme qui détermine les paramètres N, m_Z et Γ_Z par un ajustement de la fonction de Breit-Wigner

$$\sigma(E) = \frac{N^2\, E^2}{(m_Z^2 c^4 - E^2)^2 + m_Z^2 c^4 \Gamma_Z^2}$$

aux données de l'expérience OPAL pour le processus de diffusion $e^+e^- \to \mu^+\mu^-$.

La bibliothèque SciPy

Dans ce chapitre

Python

- Aperçu de la bibliothèque SciPy pour la programmation scientifique avec Python
- Le mode interactif de Python avec l'interface Jupyter

SciPy

La bibliothèque `SciPy` (`http://www.scipy.org`) est une extension de NumPy qui fournit de la fonctionnalité supplémentaire pour la programmation scientifique et en particulier pour la physique numérique.

Une grande partie des routines de SciPy prend appui sur les bibliothèques de logiciels d'analyse numérique du site `Netlib` (`http://www.netlib.org/`) dont la plupart est implémentée en `C` ou `FORTRAN`. Pour cette raison les fonctions de SciPy sont typiquement beaucoup plus efficaces qu'une fonction équivalente entièrement implémentée en Python. De plus, les bibliothèques de `Netlib` ont été testés et debougés pendant des dizaines d'années, alors elles sont relativement fiables.

SciPy

On a déjà rencontré la sous-bibliothèque `scipy.linalg` contenant des routines d'algèbre linéaire numérique. D'autres modules sont :

- `scipy.constants` qui contient les valeurs numériques de nombreuses constantes physiques
- `scipy.special` qui d'définit des fonctions speciales comme les fonctions d'Airy, de Bessel (et leurs intégrales et dérivées), des polynômes orthogonales, la fonction Gamma et les fonctions liées, les fonctions elliptiques, hypergéométriques ...
- `scipy.integrate` pour le calcul numérique des intégrales des fonctions numériques, ainsi que pour la solution numérique des systèmes d'équations différentielles avec des conditions initiales
- `scipy.optimize` pour la maximisation ou minimisation numérique des fonctions à plusieurs variables et pour la recherche des zéros
- `scipy.sparse` pour des méthodes du calcul matriciel et l'algèbre linéaire optimisées pour les matrices creuses

SciPy

- `scipy.interpolate` pour l'interpolation entre des points de données discrètes
- `scipy.fftpack` pour le calcul des transformations de Fourier
- `scipy.signal` pour des méthodes de traitement de signal
- `scipy.stats` pour des méthodes de statistique
- . . . et de nombreux autres.

Python en mode interactif avec Jupyter

Ici on ne va pas systématiquement discuter toute la fonctionnalité de SciPy. Nous allons plutôt découvrir quelques aspects sélectionnés de l'utilisation de SciPy avec un nouvel interface interactif : le notebook graphique `Jupyter`.

- Pour travailler avec les fichiers d'exemples, télécharger les trois fichiers `LennardJones.ipynb`, `Membrane.ipynb` et `Corde.ipynb` de la page `Moodle` du cours.
- Pour lancer l'interface : dans une console, entrer `jupyter notebook`. Sélectionner p.ex. le fichier `LennardJones.ipynb`.
- Le contenu du fichier est divisé en cellules : il y a des cellules de text avec des explications et des cellules de code source. Pour exécuter le code dans une cellule, la marquer et entrer `[Shift] + [Enter]`.