

Programmation pour la physique

HAP608P, Faculté des Sciences de Montpellier, 2026
Felix Brümmer (felix.bruemmer@umontpellier.fr)

- 1 Introduction
- 2 Les types de données
- 3 Les structures de contrôle
- 4 Les fonctions
- 5 NumPy et graphisme
- 6 Recherche des zéros
- 7 Algèbre linéaire numérique
- 8 Ajustement
- 9 Equations différentielles ordinaires

Introduction

Dans ce chapitre

Généralités

- Programmation scientifique et physique numérique
- Aperçu du langage Python

Python

- Instructions

Pourquoi physique numérique ?



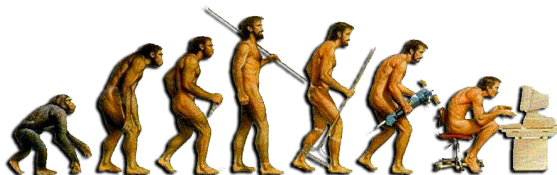
L'ordinateur est un outil indispensable en physique :

- **Solution numérique** des équations qui décrivent les systèmes physiques
- **Simulation** des systèmes complexes
- Assistance aux **calculs analytiques**
- **Pilotage** et **surveillance** des manip
- **Analyse** et **traitement des données**
- **Visualisation** des résultats, **rédaction** des publications scientifiques

• ...

Pourquoi apprendre à programmer ?

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



Programmation et intelligence artificielle générative

Q : ChatGPT, Copilot, Gemini, Claude... produisent du code vraisemblablement meilleur que le mien. Vaut-il toujours la peine d'apprendre à programmer ?

- L'IA est devenu un outil très apprécié en programmation. Mais il ne faut pas y faire confiance aveuglement.
- L'IA produira un ébauche de programme qu'un programmeur humain peut ensuite perfectionner, pourvu qu'il maîtrise bien la programmation.
- Il faut avoir appris le calcul mental avant de se servir de la calculatrice
- Il faut avoir appris à écrire correctement avant de se servir de l'autocorrection
- Il faut avoir appris à programmer avant d'utiliser l'IA

I'd like to implement the game of Tic Tac Toe in Python, with a text-based interface. Can you do this for me?

⌕ Tic Tac Toe

```
1 def print_board(board):
2     """Prints the Tic Tac Toe board."""
3     for row in board:
4         print(" | ".join(row))
5         print("-" * 5)
6
```

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 syntaxiques** que les commandes doivent suivre afin qu'elles puissent être traduites en instructions au système d'exploitation (ou directement au matériel informatique).

Cette traduction aura lieu soit au fur et à mesure lors de l'exécution du programme (**interprétation**) soit une seule fois avant l'exécution (**compilation**).

Langages de programmation

Afficher "Hello World!" avec Python

```
print("Hello World!")
```

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

```
#include <iostream>
using namespace std;

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

! Afficher "Hello World!" avec Fortran 90

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



Pourquoi Python ?

- très courant : le standard universel en programmation scientifique pour la plupart des tâches
- facile à apprendre, syntaxe simple et intuitive \Rightarrow on peut se concentrer sur le programme sans être encombré par les pièges du langage
- polyvalent, multiples domaines d'application
- beaucoup des bibliothèques incluses \Rightarrow vaste fonctionnalité
- haut niveau d'abstraction (pas de manipulation directe du matériel informatique)
- moderne :
 - programmation orientée objet
 - programmation fonctionnelle
 - typage dynamique
 - gestion automatique de mémoire

Calcul symbolique / calcul formel

[illegible]

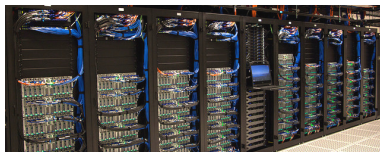
- Manipulation automatisée des objets mathématiques au niveau **symbolique**, c.à.d. sans forcément une application numérique : Algèbre, analyse, arithmétique. . .
- Les systèmes de calcul formel incluent typiquement des **langages de programmation complets** pour gérer le flot d'exécution. Pas seulement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

* propriétaire

- * basé sur Python

Quelques domaines de la programmation scientifique

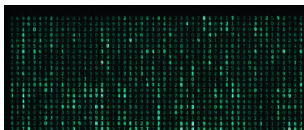
Calcul numérique à haute performance



- 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 degrés de liberté. . .
- Domaine classique de l'**analyse numérique**.
- Peut être **très demandeur** côté puissance de calcul : **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
- Interface : soit éditeur + console avec ses outils (compilateur, lieur, débogueur), soit environnement de développement intégré
- Exemples : FORTRAN, C, C++

Quelques domaines 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 (figures et animations). . .
- Besoin surtout de fonctionnalités diverses (polyvalence)
- Typiquement domaine des langages de programmation **interprétés**.
- Interface :
 - soit fichiers de script (créés dans un éditeur)
 - 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**

Développement des logiciels et outils auxiliaires



- Programmation des interfaces, de l'environnement graphique, du web, des bases de données, du système d'exploitation. . .
- Interférences avec d'autres secteurs du développement des logiciels
- Domaine des **langages universels**, interprétés ou compilés selon l'objectif
- 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**. Un autre logiciel auxiliaire, l'**éditeur de liens**, peut assister afin d'intégrer les composantes du programme et les bibliothèques externes dans le fichier exécutable.

Forces :

- Optimisation automatique
- Rapidité, efficacité augmentée
- Manque de transparence : plus facile de cacher les détails du fonctionnement
→ logiciels commerciaux

Faiblesses :

- Après toute modification du code source il faut recompiler.
- Manque de **portabilité** du code compilé
- Pas de possibilité d'exécuter seulement une partie d'un programme
- Manque de transparence : moins facile de savoir les détails du fonctionnement

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 (débugage)
- Portabilité : les seuls fichiers de programme sont celles du code source, qui sont les mêmes sur tout système

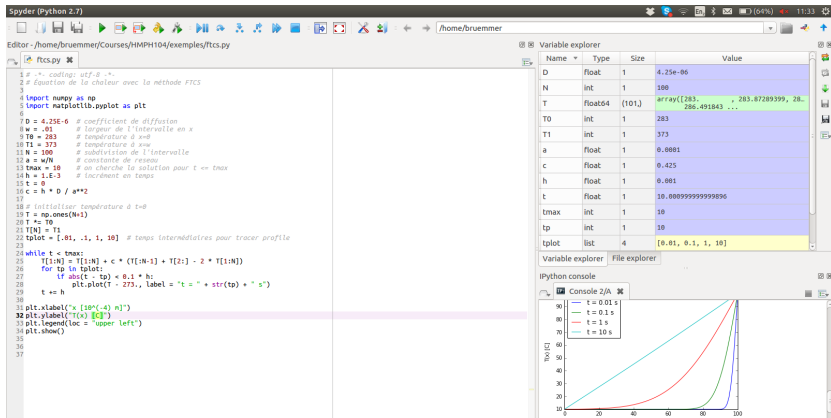
Faiblesses :

- Efficacité généralement **inférieure** à un programme compilé et optimisé.
Mais un programme interprété peut utiliser des **bibliothèques compilées** pour les parties les plus lourdes en calcul.

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

1. Fichier de code classique / script :

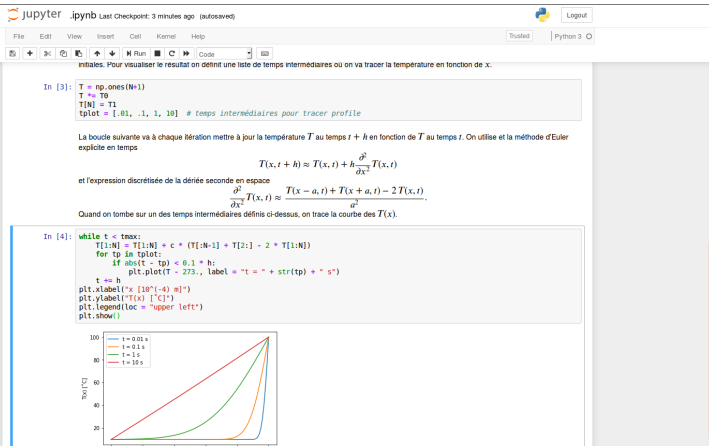
- Créé avec un **éditeur de texte**, éventuellement partie d'un **environnement de développement intégré**.
- Format : **fichier de texte**, ne contient que le code de source (et des commentaires).
- Le code est enregistré dans un ou plusieurs fichiers que l'on passe ensuite au compilateur ou à l'interpréteur.



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



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 appropriés à la programmation procédurale :
FORTRAN, C, **Python**

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 héréditaires.
- Structures plus abstraites.
- Programmes moins linéaires, favorisant la modularité
- Mieux adaptée aux grands projets de plusieurs contributeurs.
- Exemples de langages bien adaptés à la programmation orientée objet : C++, Java, **Python**



Le langage Python

- est un langage **interprété**
Plus précisément, 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.**

Ce cours porte sur des sujets en programmation scientifique et en analyse numérique :

- Révision de la programmation procédurale avec Python
- Calcul matriciel avec NumPy, graphisme
- Méthodes de recherche de zéros
- Méthodes de l'algèbre linéaire numérique
- Ajustement
- Résolution numérique d'équations différentielles ordinaires
- Introduction à la programmation orientée objet

Prérequis pour le suivre avec profit :

- Connaissances en programmation (UEs d'informatique en L1-L2)...
- ...en physique (mécanique, électrodynamique, physique quantique)...
- ...et en mathématiques (nombres complexes, analyse réelle, algèbre linéaire).

Matières à réviser indépendamment si besoin !

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://www.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++)

Vous trouverez sur Moodle :

- Ces notes de cours
- Tous les exemples de code apparaissant ci-dedans : répertoire `Exemples/`
- Toutes les fiches d'exercices

Exécuter son code avec Python

- Rédiger votre code de source avec l'aide de votre éditeur de texte préféré
- L'enregistrer dans un fichier, par exemple `exemple.py`
- Exécuter le script avec l'interpréteur Python :
Entrer `python3 exemple.py` par la console (dans le dossier où se trouve le fichier)

Raccourci : Des environnements de développement intégrés (comme `spyder`) permettent de rédiger le code et de l'exécuter directement par un clic dans l'interface graphique.

Il faudra pourtant que vos programmes soient **autonomes** (ne dépendent pas des fonctionnalités de `spyder` pour générer ses résultats) !

Un premier programme en Python

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

print("Hello World!")
```

Dans cet exemple il y a

- 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 en économiser** !
- une ligne blanche, ignorée par l'interpréteur
- une **instruction** : un appel de la fonction `print()` qui fait apparaître sur l'écran la chaîne de caractères entre les parenthèses

Exercice

Exécuter ce script (HelloWorld.py)

Format et interprétation du code source

- Chaque ligne du script (à part les lignes blanches et celles ne contenant 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.

```
print("Oups!") # erreur: la bonne fonction s'appelle print()
#
# Le programme va s'arrêter avec le message
# "NameError: name 'print' is not defined"
# qui indique que 'print' 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 incluant une parenthèse (ou un crochet [ou une accolade { **non fermé** 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.")
```

- Une ligne terminée par un **anti-slash** \ se poursuit sur la ligne suivante.
- Une ligne peut contenir **plusieurs instructions** séparées par des **point-virgules** ;

```
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.

```
import this
```

Préfère :

la beauté à la laideur,
l'explicite à l'implicite,
le simple au complexe
et le complexe au compliqué,
le déroulé à l'imbriqué,
l'aéré au compact.

Prends en compte la lisibilité.

Les cas particuliers ne le sont jamais assez pour violer les règles.

Mais, à la pureté, privilégie l'aspect pratique.

[...]

→ T. Peters, 1999

Les instructions : Erreurs fréquentes

- À la différence de Python 2, `print()` est une **fonction** en Python 3 — il faut **impérativement** mettre les **parenthèses** `()`

```
print("ça marche")           # ça marche
print "ça 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 : **Examiner le 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 produite.
- Contrairement à beaucoup d'autres langages de programmation, des espaces blancs au début d'une ligne (**indentation**) ont une **signification syntaxique**. Il ne **faut pas commencer une ligne avec des espaces blancs** (sauf si l'objectif est de créer un bloc ; voir le chapitre "Structures de contrôle").

Les types de données

Python

- Variables et affectations
- Types de données numériques
- Opérations arithmétiques
- Types de données séquentiels
- Chaînes de caractères
- Saisie du clavier

Variables, types et 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
ma_liste = ["lundi", "mardi", "mercredi"]
```

- Ici `phrase`, `nombre`, `somme`, `nombre3` et `ma_liste` désignent des variables.
- Toute variable est d'un **type** qui est déterminé par le format utilisé dans l'affectation. Ici '`phrase`' est du type **str** (chaîne des caractères), '`nombre`' et '`somme`' sont du type **int** (nombres entiers), '`nombre3`' est du type **float** (nombre flottant) et '`ma_liste`' est du type **list** (liste d'objets).
- Une fois initialisée, 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 comme 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 comme 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 probablement créer des bogues).
- **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
<code>int</code>	nombre entier sans limite de taille théorique
<code>float</code>	nombre flottant, précision 64 bit soit ≈ 15 décimales
<code>complex</code>	nombre flottant complexe correspondant à deux <code>float</code>

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 à virgule flottante** 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 + 5.E-3J   # le nombre complexe -2.5 + 0.005 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

re_z = 1.7           # un float
im_z = -2.8           # un deuxième float
z = complex(re_z, im_z) # le nombre complexe 1.7 - 2.8 i
```

Les types de données numériques : `complex`

Pour obtenir la partie réelle ou imaginaire d'un nombre complexe :

```
z = -1 + 2.3J  
re_z = z.real # donne -1.0  
im_z = z.imag # donne 2.3
```


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 valeur 8
y = x / 5      # y est du type float la valeur 1.6
z = x // 5     # z est du type int avec 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 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 à gauche du signe `=`

Cela permet des instructions comme

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

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

```
x = y = 7      # x et y sont du type int avec valeur 7
```

Les affectations

Combinaison des opérations arithmétiques avec des affectations, pour changer la valeur d'une variable :

```
x = 2      # x est initialement du type int avec valeur 2

x += 1     # ajouter 1 à x: x devient 3
           # (équivalent: x = x + 1)

x *= 3     # multiplier x par 3: x devient 9
           # (équivalent: x = x * 3)

x -= 1     # soustraire 1 de x: x devient 8
           # (équivalent: x = x - 1)

x /= 4     # diviser x par 4: x devient 2.0
           # (équivalent: x = x / 4)
```

Les types séquentiels (“sequence types”)

type	description
<code>str</code>	chaîne de caractères
<code>list</code>	liste muable d'objets
<code>tuple</code>	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 (HAP608P)'
```

- On peut convertir toute 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 en notation Python.

```
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)  # affiche "J'ai 12 ans."
```

Avec `[]` on accède aux caractères séparés. **Attention : l'indice du premier caractère est toujours 0 !**

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

L'opérateur `[a:b]` retourne la **sous-chaîne** de caractères à partir de l'indice `a` (inclus) jusqu'à l'indice `b` (exclus). 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])    # affiche "'a"
print(phrase[5:])     # affiche "12 ans"
print(phrase[:6])     # affiche "J'ai 1"
```

Le caractère d'échappement dans une chaîne de caractères

Dans une chaîne des caractères, la barre oblique inversée ou anti-slash `\` a une rôle spéciale : c'est le **caractère d'échappement**.

- Dans un string littéral entouré des apostrophes `'`, les caractères `"` sont traités comme des caractères réguliers et vice-versa. Si on veut inclure le caractère `'` (ou `"`) dans un string littéral qui commence et finit avec `'` (ou `"`), il doit être précédé par un `\` :

```
print("J'ai 12 ans")  
print('J\'ai 12 ans') # même résultat  
print("\"Infâme !\" s'exclama Bastien.")
```

- 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 récupérer 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 + " font " + str(i**2))
```


Les types séquentiels : `list`

Le type `list` représente une **liste d'objets**. Les listes sont délimitées par des **crochets** `[]`. Leurs éléments sont séparés par des virgules `,`.

```
premiers_nombres_premiers = [2, 3, 5, 7, 11]
hiver = ["décembre", "janvier", "février"]
```

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

```
liste_bizarre = [2, 2.0, 2 + 0J, "deux"]
liste_nulle = [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 **fusionner** des listes :

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

Avec [] on accède aux éléments individuels. **Attention : l'indice du premier 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 une copie de toute la liste mois2
```

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 délimité par des **parenthèses** () avec les éléments séparés par des virgules ,. On peut supprimer les parenthèses si pas d'ambiguïté. Exemples :

```
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
```

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    # calculer 5 * 3 et oublier le résultat
```

- Penser à **convertir** ses variables au bon type :

```
age = input("Quel est ton age ? ") # input() retourne un str
naissance = 2025 - 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 variables et les types de données : Erreurs fréquentes

- Notation scientifique : le E signifie “ $\times 10$ à la puissance de”. Alors pour convertir des nanomètres en mètres :

```
nm = 10E-9      # faux
nm = 10**-9     # pas strictement faux mais à éviter
nm = 1E-9       # correct
```

- Python utilise le **point décimal**, pas la virgule décimale. Alors

```
pi = 3.14        # pi est un float de valeur 3.14
pas_pi = 3,14    # pas_pi est un tuple de deux entiers 3 et 14
```

Les structures de contrôle

Python

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

Les blocs d'instructions

Un **bloc** est une séquence de lignes d'instructions distinguées par leur **indentation** (décalage par rapport aux lignes qui les entourent). Une ou plusieurs lignes consécutives décalées au même niveau constituent un bloc. Un bloc est toujours précédé par une **ligne d'en-tête** qui se termine avec un deux-points :

```
...  
LIGNE EN-TETE:                # introduit un bloc  
    INSTRUCTION 1  
    INSTRUCTION 2  
    ...  
    DERNIERE INSTRUCTION      # ici le bloc se termine  
INSTRUCTION SUIVANTE          # <- ne fait plus partie du 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**.

Les blocs d'instructions

Un bloc peut en contenir d'autres :

```
...  
LIGNE EN-TETE:      # ici commence un bloc  
    INSTRUCTION  
    INSTRUCTION  
    ...  
    LIGNE EN-TETE: # ici commence un sous-bloc  
        INSTRUCTION  
        INSTRUCTION  
        ...  
        DERNIERE_INSTRUCTION # fin du sous-bloc  
    INSTRUCTION      # le 1er bloc se poursuit  
    ...  
    DERNIERE INSTRUCTION # fin du 1er bloc  
INSTRUCTION HORS BLOC  
...
```

De même pour les sous-sous-blocs etc.

La structure conditionnelle

La **structure conditionnelle** (ou structure **if**) prend la forme suivante :

```
if CONDITION: # ligne d'en-tête caractérisée par mot clé if
    INSTRUCTION 1    # bloc à exécuter si CONDITION vérifiée,
    INSTRUCTION 2    # à sauter sinon
    ...
    DERNIERE INSTRUCTION # ici le bloc se termine
CONTINUER_ICI    # en tout cas le programme arrive 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.
- Sinon le programme saute le bloc et continue directement à **CONTINUER_ICI**.
- Les deux-points : dans la ligne d'en-tête font partie de la structure et ne doivent pas être omis

La structure conditionnelle

Exemples :

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

```
print("Tu veux savoir un secret?")
reponse1 = input("Entre 'o' si oui: ")
if reponse1 == "o":
    reponse2 = input("T'es sur? Entre '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'agit 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). (On peut même directement utiliser la valeur numérique correspondante dans une structure conditionnelle au lieu de la condition – normalement déconseillé car peu lisible.)

Les opérateurs de comparaison :

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

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

<code>not a</code>	True si a est False et vice-versa
<code>a and b</code>	True si a est True et b est True, False autrement
<code>a or b</code>	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
"gorille" 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 éléments
y = [1, 2]      # une autre liste avec les mêmes éléments
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 si la condition `CONDITION` est `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
    ...
...
```

La boucle `while`

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

```
while CONDITION:
    FAIRE_QUELQUE_CHOSE # ce bloc est répété tant que
    ...                 # CONDITION est 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("La moitié de", i, "est", i // 2)
```

La boucle while

Deuxième exemple : Conjecture de Collatz.

On définit la suite (n_k) par un $n_0 \in \mathbb{N}$ et la règle de récurrence

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

Conjecture : Pour toute valeur de départ n_0 on va ultérieurement tomber sur $n_i = 1$ (et puis $n_{i+1} = 4$, $n_{i+2} = 2$, $n_{i+3} = 1$ etc.)

En supposant que la conjecture soit vraie (sinon : boucle infinie, le programme ne terminera jamais!), on calcule le nombre minimal d'itérations i pour tomber sur $n_i = 1$, avec n_0 fourni par l'utilisateur :

```
n = int(input("Entrez n0: "))
i = 0    # compteur d'itérations
while n != 1: # ne termine que si la conjecture est vraie !
    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("Tombé sur 1 après", i, "itérations.")
```

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 $(0, 1, 2, \dots, y-1)$
- `range(x, y)` retourne $(x, x+1, x+2, \dots, y-1)$
- `range(x, y, s)` retourne $(x, x+s, x+2s, \dots, x+ns)$ avec $x+ns < y$ maximal

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

```
for x in range(ITER):  
    FAIRE_QUELQUE_CHOSE    # bloc répété 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 les caractères :

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

Boucles `for` imbriquées :

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

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!")
```

Les structures de contrôle : Erreurs fréquentes

- Deux-points oubliés après `if`, `while`, `for` etc.
- Pour l'indentation des blocs :
Ne jamais mélanger les espaces et les tabulatrices.
Conseillé : éviter 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) tant que `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 !") # Jamais atteint car i ne change pas
```

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 de fonctions

On a déjà employé quelques fonctions intégrées dans Python comme `print()`, `input()` et `range()`. En général, une **fonction** est une partie du programme qui

- peut être **appelée** avec un ou plusieurs paramètre(s) dit **argument(s)**
- effectue une **tâche** en fonction de ces arguments
- peut **retourner** une valeur

Par exemple, comme nous l'avons vu, la fonction `range()` prend entre 1 et 3 arguments et retourne un n -uplet de nombres entiers.

Autres exemples :

fonction	argument(s)	tâche	valeur de retour
<code>print()</code>	plusieurs	affichage sur l'écran	aucune
<code>input()</code>	un str	affiche son argument, attend saisie du clavier	le str entré par le clavier
<code>int()</code>	un nombre ou str qui peut être converti en int	convertit son argument en int	le résultat de la conversion
<code>len()</code>	une séquence	compte le nombre d'éléments	le nombre d'éléments dans la séquence

Nouvelles fonctions

Voici un exemple d'une **définition d'une fonction originale** `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() avec l'argument
             # x=5, affecte la valeur de retour à a
print(a)    # affiche "125"
```

Définir une fonction

La syntaxe pour une définition de 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 prendre un nombre quelconque d'**arguments** ARG1, ARG2 etc. (mais il est également possible de définir des fonctions sans arguments)
- Une fois la fonction définie, on l'appelle avec la commande
 NOM_DE_FONCTION(VAL1, VAL2, ...)
où VAL1, VAL2 etc. sont les valeurs à substituer pour les arguments ARG1, ARG2 etc. pendant cet appel.
La valeur de l'expression d'appel devient la valeur de retour de la fonction.
- Le bloc suivant la ligne d'en-tête, caractérisé par le mot clé **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, des appels de fonctions. . .

Valeurs par défaut des arguments

Il est possible de spécifier des **valeurs par défaut** pour tous les arguments ou une partie :

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

Si une valeur par défaut par un des arguments est spécifiée dans la définition, il n'est plus nécessaire de fournir cet argument lors de l'appel.

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

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

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

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

Valeurs par défaut des arguments

Si plusieurs arguments ont des valeurs par défaut : Dans un appel, les arguments non nommés doivent toujours **précéder** 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 pas précéder un argument non nommé)

La commande `return`

La commande `return` peut figurer à un ou plusieurs endroits dans la définition d'une fonction. Dès que l'interpréteur la rencontre, il **abandonne la fonction** et continue l'exécution du programme à l'endroit de l'appel.

L'expression derrière le `return` est **retourné** et devient la valeur de l'expression d'appel de fonction.

```
def heaviside(r):                # la définition d'une fonction
    if r >= 0:
        return 1.0
    else:
        return 0.0

print("Theta(1) =", heaviside(1)) # un premier appel
print("Theta(-1) =", heaviside(-1)) # un deuxième appel
```

Si une fonction n'est pas terminée par un `return`, ou pour un `return` sans paramètre, la fonction retourne l'objet abstrait `None` ("aucun").

Récommandations pour créer du code plus lisible :

- Adopter des **conventions cohérentes** et les suivre partout.
- Ne pas économiser sur les **commentaires**.
- **Une instruction par ligne**. Eviter les point-virgules.
- **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
- **un seul espace** après `,` `;` `:`
- **un seul espace** à chaque coté des opérateurs d'affectation et de comparaison
- **Désignations parlantes** pour les identifiants importants.
Préférer des **minuscules** et éventuellement des **chiffres** et **tirets bas** `_` pour les variables et fonctions.
- Préférer des **majuscules** pour les classes en programmation orientée objet.

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. Pour une affectation à l'intérieur d'une définition de fonction, la portée de l'identifiant est limitée à **cette même définition de fonction**. Donc le code

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

produira une erreur. Par contre, si une variable est définie hors d'une définition de fonction, on peut tout de même l'utiliser à son intérieur :

```
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 correspondra à la variable globale x
    x = 1      # <- ne crée pas une nouvelle variable locale
f()
print(x)      # "1"
```

Récurtivité

Une fonction peut **s'appeler elle-même** :

```
def factorielle(n): # calcule n! recursivement
    if n < 0:       # factorielle pas définie
        print("Erreur: factorielle pas définie.")
        return
    elif n == 0:    # 0! = 1
        return 1
    else:
        return n * factorielle(n - 1) # n! = n (n-1)!
```

On parle de la **récurtivité**.

- La récurtivité permet parfois des codes courts et élégants.
- Mais un algorithme récursif est souvent **moins rapide** qu'un algorithme équivalent itératif (qui se sert des boucles).
- Le nombre d'appels récursives imbriqués est limité à 1000 par défaut.
- Faire attention d'inclure une condition de terminaison appropriée !

Fonctions comme arguments

On peut passer des **fonctions comme arguments** aux autres fonctions :

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

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

print(iterer(logistique, 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 "racine n-ième"
    def f(x):
        return x**(1/n)
    return f

g = racine(5) # définit la fonction "racine 5-ième"
print(g(32))  # affiche "2.0"
print(racine(3)(27)) # affiche "3.0"
```

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

Fonctions `lambda`, fonctions anonymes

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

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

on écrit

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

Plus généralement,

```
lambda ARGUMENTS : EXPRESSION
```

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

Fonctions `lambda` : Exemples

Fonctions d'ordre supérieur :

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

print(racine(4)(81))    # affiche "3.0"
```

Listes de fonctions :

```
# une fonction, sa dérivée et sa dérivée seconde
fonctions = [lambda x: 3*x**2 - 2*x,
             lambda x: 6*x - 2,
             lambda x: 6]
```

Les modules

La commande `import` sert à **importer** du code d'un autre fichier ou d'une bibliothèque.

Exemple : On enregistre dans un fichier `puissances.py` les définitions de fonctions

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

On peut ensuite les utiliser dans un autre projet

```
# fichier nouveauprojet.py (dans le même répertoire)
import puissances
print(puissances.carre(42))
```

sans recopier tout. Ou, si on veut seulement importer la fonction `carre()` :

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

La bibliothèque standard

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

- la manipulation des chaînes de caractères
- la manipulation des tableaux de données (**NumPy**, voir plus tard)
- les fonctions mathématiques (**math** et **cmath**, voir ci-dessous, ainsi que **NumPy**)
- les nombres rationnelles
- l'accès aux fichiers et leur manipulation
- les interfaces aux bases de données
- la programmation fonctionnelle
- la compression et la sauvegarde des données
- les services cryptographiques
- les interactions avec le système d'exploitation
- les services de réseau
- les services internet et les pages web
- multimédia

La bibliothèque standard : les modules `math` et `cmath`

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

```
import math    # pour importer toutes les fonctionnalités
               # du module 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 : les modules `math` et `cmath`

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 4 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` au lieu de `math`.

Quelques autres 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)
```

Quelques autres 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) # un entier aléatoire entre 10 et 19
```


Les fonctions : Erreurs fréquentes

- Deux-points oubliés après la commande `def`
- Il faut qu'une fonction soit définie **avant qu'on l'appelle**.

```
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()
```

- Une **définition de fonction** sans qu'elle soit **appelée** n'est pas un programme complet !

NumPy et graphisme

Python

- La bibliothèque `numpy`
- Manipulation des tableaux
- Calcul matriciel
- Importer et exporter des données
- Graphisme avec `matplotlib`

Les tableaux

On rappelle les caractéristiques d'une **liste** Python :

- contient plusieurs éléments qui ne sont **pas forcément du même type**
- taille **variable**, peut changer (p.ex. avec l'opérateur +=)
- **1-dimensionnel** = 1 seul indice (sauf si les éléments sont eux-mêmes des listes)

La bibliothèque **NumPy** se base sur un objet similaire, le **tableau** (anglais : "array")

Caractéristiques d'un tableau NumPy :

- contient plusieurs éléments qui sont **forcement du même type**, toujours 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 :

```
import numpy as np # pour importer toute la bibliothèque

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.]

id3x3 = 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 tableau peut se créer

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

```
import numpy as np
tab1 = np.arange(0, 1.2, 0.2) # [0. 0.2 0.4 0.6 0.8 1.0]
tab2 = np.linspace(0, 1, 6)   # [0. 0.2 0.4 0.6 0.8 1.0]
```



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

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 se correspondent pas, une opération arithmétique impliquant deux tableaux produira une erreur.

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 tableau 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 le **couper** comme une liste 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 $(0, 1, 2, \dots, 11)$ et le réarranger dans une matrice 3×4 :

```
a = np.reshape(np.array(range(12)), (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)
- en sélectionnant un élément sur `k` (par défaut : 1)

Exemples :

```
print(a[1, ::2])  # [4 6] (2ème ligne, colonnes paires)
print(a[1, 1::2]) # [5 7] (2ème ligne, colonnes impaires)
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)
```

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 l'**opérateur @** (et non pas avec * qui est la multiplication élément par élément !)

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(v @ w) # produit scalaire v . w, résultat: 1.0

print(M @ v) # matrice agit sur vecteur, M . v
              # résultat: [ 10. -1.  0.]

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

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

Pour rappel : définition des matrices de Pauli

$$\sigma^1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \sigma^2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \sigma^3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

et de la moyenne quantique d'un opérateur \hat{A} pour un système dans l'état

$$|\psi\rangle = \begin{pmatrix} \psi_1 \\ \psi_2 \end{pmatrix} \text{ (supposé normalisé, alors } \|\psi\|^2 = \psi_1^* \psi_1 + \psi_2^* \psi_2 = 1)$$

$$\langle \hat{A} \rangle = \langle \psi | \hat{A} | \psi \rangle = (\psi_1^* \ \psi_2^*) \hat{A} \begin{pmatrix} \psi_1 \\ \psi_2 \end{pmatrix}$$

On va calculer les moyennes de σ^1 , σ^2 et σ^3 dans un état fourni par l'utilisateur.

Multiplication matricielle avec les tableaux

```
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 norme(psi): # la norme d'un vecteur complexe
    psic = np.conjugate(psi)
    return np.sqrt(psic @ psi)

psi1 = complex(input("Entrer psi1: ")) # composantes de psi
psi2 = complex(input("Entrer psi2: "))
psi = np.array([psi1, psi2], dtype=complex)
psi /= norme(psi) # normaliser le vecteur psi
psic = np.conjugate(psi) # le vecteur conjugué complexe
vm = [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 tableau, alors

- `numpy.transpose(A)` représente la transposée de `A`
(raccourci : `A.T`)
- `numpy.trace(A)` calcule la trace $\sum_i A_{ii}$
- `numpy.conjugate(A)` calcule le tableau conjugué complexe
- `numpy.amax(A)` calcule l'élément maximal
- `numpy.sum(A)` calcule la somme des éléments
- ...

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

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]) # un tableau
print(np.arccos(x)) # "[ 3.14159265  1.57079633  0.]"
```

À préférer par rapport au code équivalent (mais moins vite et moins propre)

```
import math

x = [-1, 0, 1] # une liste
acosx = [math.acos(t) for t in x] # lent sur des grandes
                                # listes !

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 "superficielle"** avec `numpy.copy()` :

```
import numpy as np

a = sigma3          # permet d'accéder à sigma3
                    # avec la nouvelle référence a
b = np.copy(a)      # crée un nouveau tableau
                    # 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 doivent être séparés par un ou plusieurs espaces blancs
- lignes blanches et commentaires `#` sont 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
0.000000000000000000e+00
1.000000000000000000e+00
2.000000000000000000e+00
3.000000000000000000e+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 :

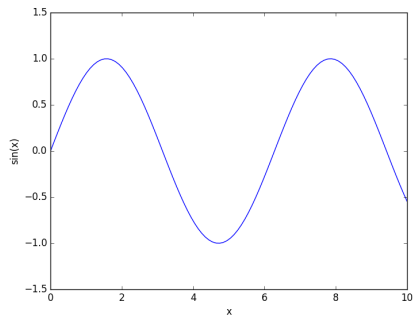
```
import numpy as np
import matplotlib.pyplot as plt

xpts = np.linspace(0., 10., 100) # 100 points entre 0 et 10
ypts = np.sin(xpts)             # Les sinus de ces points

plt.plot(xpts, ypts)             # tracer ypts sur xpts
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 :



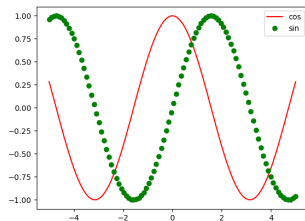
La fonction `matplotlib.pyplot.plot()`

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 / le style, p.ex.
 - '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
- d'autres arguments optionnels existent, p.ex. pour faire afficher une légende

Exemple :

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



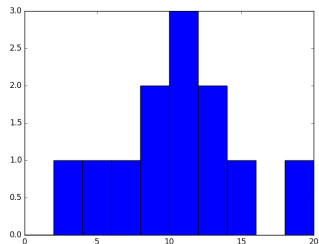
La fonction `matplotlib.pyplot.hist()`

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 ,

$$X = \left(\begin{array}{cccc} 0 & 0.1 & 0.2 & \dots & 10 \\ 0 & 0.1 & 0.2 & \dots & 10 \\ & & \dots & & \end{array} \right) \left. \vphantom{\begin{array}{c} \\ \\ \end{array}} \right\} 10 \text{ lignes}, \quad Y = \underbrace{\left(\begin{array}{cccc} 3 & 3 & \dots & 3 \\ 3.1 & 3.1 & \dots & 3.1 \\ \vdots & & & \vdots \\ 3.9 & 3.9 & \dots & 3.9 \end{array} \right)}_{101 \text{ colonnes}}$$

La fonction `matplotlib.pyplot.imshow()`

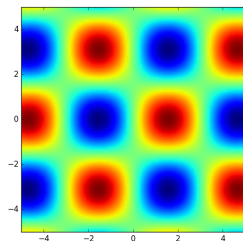
Tracer des cartes thermiques (heat map) : `matplotlib.pyplot.imshow(z, extent)`

- `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)`

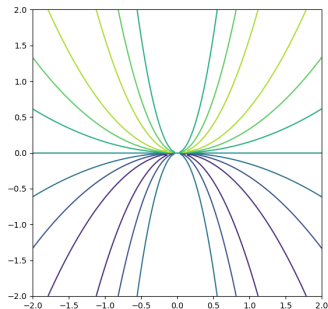
- `z` = tableau 2D avec les valeurs de fonction
- arguments facultatifs : `x`, `y` = tableaux avec les `x` et `y` correspondants

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()
```



Recherche des zéros

Algorithmes

- Méthode de la bisection
- Méthode de relaxation
- Méthode de Newton
- Généralisations de la méthode de Newton

Problème : Etant donné une fonction réelle dont on sait qu'elle a un zéro sur l'intervalle $I = [a, b]$, on cherche une valeur approximative de ce zéro.

Plus formellement :

Soit $I = [a, b]$ un intervalle et $f : I \rightarrow \mathbb{R}$ continue (ou même dérivable si nécessaire) 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 $f(x_0) = 0$. Si de plus f est monotone alors x_0 est unique.

Problème : déterminer (un des) x_0 numériquement avec une précision minimale donnée.

Théorème des valeurs intermédiaires :

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

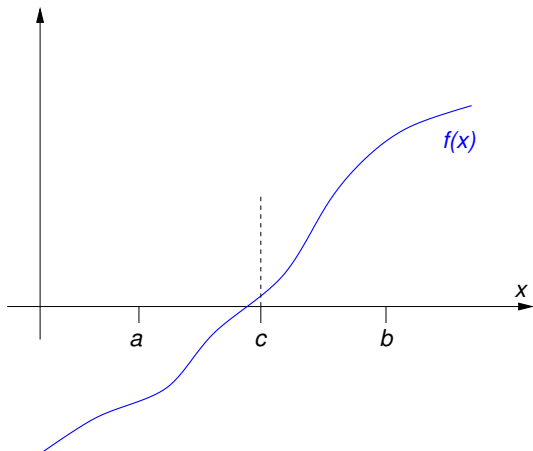
Méthode de la bisection

Soit $\epsilon > 0$ la précision souhaitée.

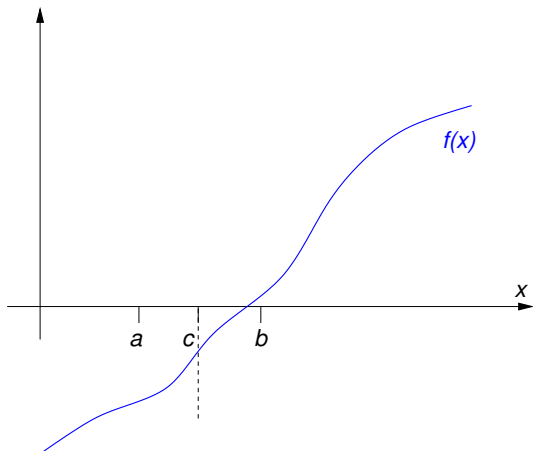
Algorithme :

- 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 $f(a)f(c) \leq 0$: il y a un zéro dans I_1 , alors répéter avec $I = I_1$.
- 5 Sinon, répéter avec $I = I_2$.

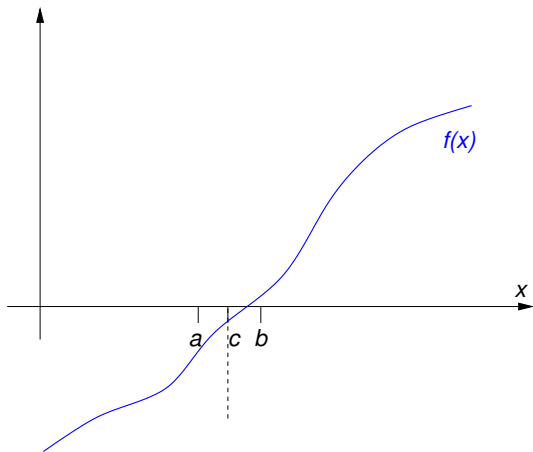
Méthode de la bisection



Méthode de la bisection



Méthode de la bisection



Implémentation en Python

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

```
def f(x):          # la fonction dont on cherche un zéro
    return x**5 - x - 1

a, b = 1.0, 2.0    # f(a) = -1 et f(b) = 29
                  # => il y a un zéro dans [a, b]
c = (a + b) / 2    # point du milieu
epsilon = 1.0E-5   # tolérance

while b - a >= 2 * epsilon: # on est assez proche ? sinon:
    if f(a) * f(c) <= 0:    # si zéro dans la moitié gauche:
        b = c              # pt de droite <- pt du milieu
    else:                  # sinon:
        a = c              # pt de gauche <- pt du milieu
        c = (a + b) / 2    # recalculer point du milieu

print("Le zéro est à x =", c)
```

Idée :

- Pour résoudre l'équation $f(x) = 0$, trouver une fonction $\phi(x)$ appropriée telle que

$$f(x) = 0 \Leftrightarrow \phi(x) = x$$

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

- On cherche alors un **point fixe** x^* de la fonction ϕ .
- Essayer de trouver un point fixe par l'application répétée de la fonction ϕ sur un point de départ x_1 (qui est aussi au choix et doit être bien choisi pour que la méthode fonctionne)

$$x_2 = \phi(x_1)$$

$$x_3 = \phi(x_2) = \phi(\phi(x_1))$$

$$x_4 = \phi(x_3) = \phi(\phi(\phi(x_1)))$$

...

$$\lim_{n \rightarrow \infty} x_n \stackrel{?}{=} x^*$$

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

$$x_2 = \phi(x_1) = 1.63212$$

$$x_3 = \phi(x_2) = 1.80448$$

...

$$x_9 = \phi(x_8) = 1.84141$$

$$x_{10} = \phi(x_9) = 1.84141$$

$$x_{11} = \phi(x_{10}) = 1.84141$$

...

Conclusion : Pour $x^* \approx 1.84141$ on a

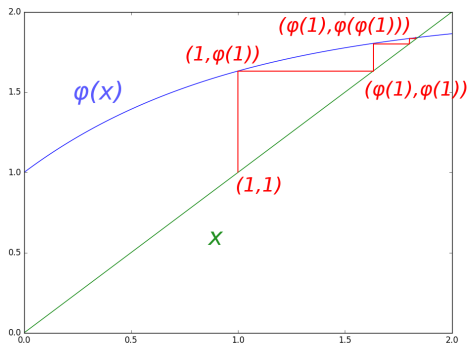
$$\phi(x^*) = x^*$$

$$\Leftrightarrow 2 - e^{-x^*} = x^*$$

$$\Leftrightarrow 2e^{x^*} - x^*e^{x^*} - 1 = 0$$

$$\Leftrightarrow f(x^*) = 0.$$

Méthode de relaxation : Illustration



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

Théorème du point fixe :

Soit $I = [a, b]$ un intervalle, $0 \leq k < 1$, et $\phi : I \rightarrow I$ 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, une suite (x_n) dans I définie par un x_1 et par $x_{n+1} = \phi(x_n)$ **convergera vers x^*** .

Démonstration :

Soit $x_1 \in I$ quelconque et $x_{n>1}$ défini par récurrence : $x_{n+1} = \phi(x_n)$. On a

$|x_{n+1} - x_n| \leq k^{n-1}|x_2 - x_1|$, donc

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

Comme $k < 1$, l'expression dans la dernière ligne tend vers zéro quand $n \rightarrow \infty$, alors les (x_n) forment une suite de Cauchy qui converge vers un x^* . On a $\phi(x^*) = x^*$ car ϕ est continue.

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** t_1 ainsi définie, et d'itérer :

- Poser

$$t_1(x) = f(x_1) + f'(x_1)(x - x_1)$$

(= développement limité de f en x_1 , alors $t_1(x) = f(x) + \mathcal{O}(|x - x_1|^2)$)

- Le zéro de $t_1(x)$ est à $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$.
- Itérer cette procédure tant que la distance entre les deux valeurs consécutives x_n et x_{n+1} est $> \epsilon$. Quand $|x_{n+1} - x_n| < \epsilon$, terminer et renvoyer x_{n+1} .

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

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

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

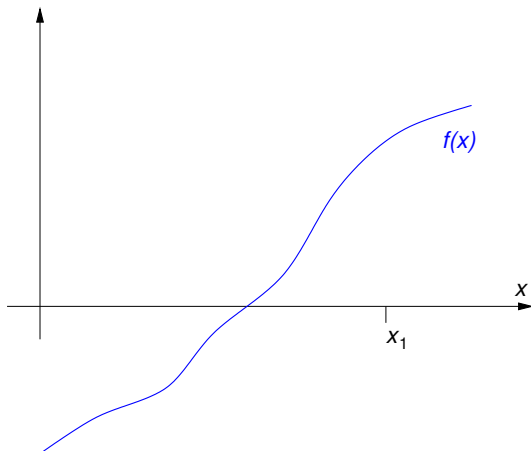
$$\phi : x \mapsto x - \frac{f(x)}{f'(x)}$$

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 $\left| \frac{f(x)f''(x)}{f'(x)^2} \right| \leq k$
("⇐" vient du théorème des accroissements finis, "⇒" de la définition de la dérivée)
- Pour que l'algorithme converge : Il est **suffisant** mais pas **nécessaire** que ϕ soit une contraction.

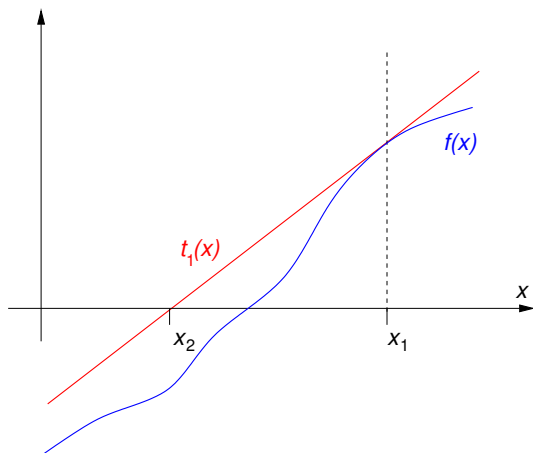
Etablir un critère suffisant et nécessaire peut être très difficile voire impossible en pratique, voir exercices sur le cas complexe.
- Evidemment il faut bien choisir le point de départ (un extremum de f , par exemple, serait un très mauvais choix — pourquoi ?)

Méthode de Newton



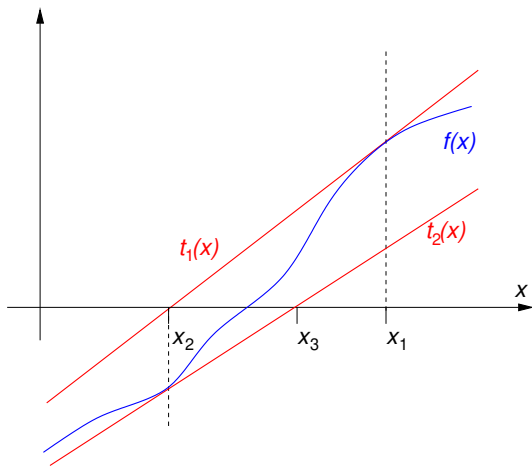
$$t_n(x) = f(x_n) + f'(x_n)(x - x_n)$$

Méthode de Newton



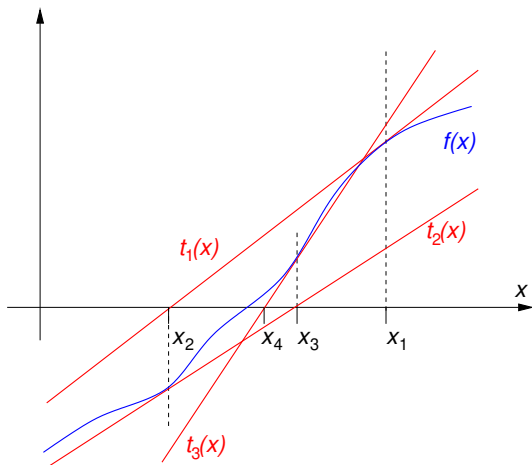
$$t_n(x) = f(x_n) + f'(x_n)(x - x_n)$$

Méthode de Newton



$$t_n(x) = f(x_n) + f'(x_n)(x - x_n)$$

Méthode de Newton



$$t_n(x) = f(x_n) + f'(x_n)(x - x_n)$$

Algorithme :

- ❶ Partir avec x au choix (mais choisi de manière réfléchie : aussi proche du zéro que possible, pas un point critique...)
- ❷ Remplacer $x_{\text{ancien}} \leftarrow x$, $x \leftarrow x - \frac{f(x)}{f'(x)}$.
- ❸ Si $|x - x_{\text{ancien}}| < \epsilon$, on est suffisamment proche du zéro : terminer et renvoyer x .
Sinon, répéter.

Pour tester la convergence, d'autres critères sont envisageables (par exemple, est-ce que $|f(x)| < \epsilon$?) en fonction du problème sous étude.

Méthode de Newton

Exemple :

On cherche un zéro de la fonction $f : x \mapsto x^5 - x - 1$ dont la dérivée est $f' : x \mapsto 5x^4 - 1$. La précision souhaitée est $\epsilon = 10^{-5}$ et le point de départ sera $x_0 = 1$.

```
def f(x):  
    return x**5 - x - 1  
  
def df(x):  
    return 5 * x**4 - 1  
  
epsilon = 1.E-5  
x = 1.  
x_ancien = x + 2 * epsilon # valeur initiale pas importante  
  
while abs(x - x_ancien) > epsilon:  
    x_ancien = x  
    x = x - f(x) / df(x)  
  
print("Le zéro est à x =", x)
```

Il est souvent une bonne idée de limiter le nombre d'itérations pour éviter des boucles infinies (au cas où la méthode ne converge pas avec le point de départ choisi).

Méthode de la sécante

Pour appliquer la méthode de Newton, il faut connaître la dérivée de la fonction f , de préférence analytiquement.

Sinon : approximer la dérivée par le taux d'accroissement

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

Au point $x = x_n$ avec $h = x_n - x_{n-1}$:

$$f'(x_n) = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} + \mathcal{O}(|x_n - x_{n-1}|)$$

Insérer $f'(x_n)$ dans la formule de récurrence de la méthode de Newton

$x_{n+1} = x_n - f(x_n)/f'(x_n)$, en supprimant le terme $\mathcal{O}(|x_n - x_{n-1}|)$:

$$x_{n+1} = x_n - f(x_n) \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

Méthode de la sécante.

Algorithme :

- Commencer par deux points au choix x_0 et x_1 ; poser $n = 1$.
- Calculer x_{n+1} avec la formule de récurrence ci-dessus.
- Itérer avec $n \leftarrow n + 1$; quand $|x_{n+1} - x_n|$ est suffisamment petit, terminer et renvoyer x_{n+1} .

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	Bissection	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 significatif pour cet exemple (implémentation pas optimisée, fonction f pas très compliquée). Voici le temps en secondes requis sur un ordinateur portable générique pour atteindre une précision fixe :

Precision	Bissection	Relaxation	Newton
1.00e+00	5.51e-07	3.17e-07	4.48e-07
1.00e-02	2.47e-06	5.24e-07	1.40e-06
1.00e-04	4.71e-06	7.45e-07	1.64e-06
1.00e-06	6.57e-06	9.51e-07	1.87e-06

Généralisations de la méthode de Newton

- La méthode peut être appliquée sans modifications pour des fonctions complexes. Les domaines de convergence vers les zéros forment des structures géométriques intéressantes : les “fractales de Newton” → 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 nécessite l'évaluation de la dérivée seconde.

- 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.

La méthode de Newton n -dimensionnelle

Étant donnée une fonction $\vec{f}: \mathbb{R}^n \rightarrow \mathbb{R}^n$, un zéro de \vec{f} peut être trouvé par la généralisation de la méthode de Newton : on itère

$$\vec{x}_{n+1} = \vec{x}_n - J^{-1}(\vec{x}_n) \vec{f}(\vec{x}_n)$$

où J est la matrice jacobienne des dérivées de \vec{f} ,

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \dots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}$$

et J^{-1} est son inverse. Voir le chapitre sur l'ajustement plus tard pour une application.

Algèbre linéaire numérique

Dans ce chapitre

Généralités

- Systèmes d'équations linéaires
- Diagonalisation des matrices

Algorithmes

- La méthode de Gauss
- La décomposition LU
- La décomposition QR et l'algorithme QR

Résoudre un système d'équations linéaires

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

$$a_{11} x_1 + a_{12} x_2 + \dots + a_{1n} x_n = b_1$$

$$a_{21} x_1 + a_{22} x_2 + \dots + a_{2n} x_n = b_2$$

\dots

$$a_{n1} x_1 + a_{n2} x_2 + \dots + a_{nn} x_n = b_n$$

L'algorithme de Gauss

L'**algorithme de Gauss** est une méthode pour résoudre ces systèmes d'équations linéaires.
En notation matricielle :

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

Regardons 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, \dots, x_n) du système est inchangée 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

Par une suite 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 $a'_{ij} = 0$ si $i > j$.

L'algorithme de Gauss

Pour transformer M en matrice triangulaire supérieure :

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

En pratique, on choisit souvent comme pivot le plus grand coefficient de chaque 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, \dots, 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
- ...
- $x_i = (b'_i - \sum_{k>i} a'_{ik} x_k) / a'_{ii}$, avec les x_k pour $k > i$ déjà connus
- ...
- $x_1 = (b'_1 - \sum_{k>1} a'_{1k} x_k) / a'_{11}$, avec les x_k pour $k > 1$ 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 (dans ce cas la solution pour x_i n'est pas unique).

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

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

- Au début :

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

- 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 on échange d'abord la deuxième ligne avec 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} :

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

L'algorithme de Gauss

Une procédure auxiliaire :

```
import numpy as np

# Transforme une matrice n*(n+1) en forme triang. supérieure
def triangulariser(M):
    n = M.shape[0]          # le nombre de lignes
    for i in range(n):      # boucle sur les premières n colonnes
        for k in range(i, n): # chercher pivot sous la diagonale
            if M[k, i] != 0:  # pivot trouvé dans ligne k ?
                M[[i, k], :] = M[[k, i], :] # échanger lignes i et k
                pivot = M[i, i]             # mémoriser pivot
                break                       # quitter boucle sur k
            else:               # tous les éléments sous la diagonale étaient 0 ?
                continue        # alors rien à faire pour cette colonne
        for k in range(i+1, n): # éliminer tout sous la diag.:
            facteur = -M[k, i]/pivot # ajouter (facteur)
            M[k, :] += facteur * M[i, :] # *(ligne du pivot)
                                         # à la ligne k.
```

L'algorithme de Gauss

```
def gauss(A, b): # trouver la solution x de  $Ax = b$ 
    n = A.shape[0] # le nombre de lignes
    M = np.empty((n, n+1)) # la matrice M
    M[:, :n] = np.copy(A) # copier A dans les 1ères n colonnes
    M[:, n] = np.copy(b) # copier b dans la dernière colonne
    triangulariser(M) # après, M est triangulaire supérieure

    x = np.empty(n) # on mettra la solution ici
    for i in range(n-1, -1, -1): # parcourir lignes en arrière
        sigma = 0. # sigma = la somme des (x_k déjà connus)*
        for k in range(i+1, n): # *(M_ik correspondants)
            sigma += M[i, k] * x[k]
        if M[i, i] == 0: # Matrice singulière?
            if M[i, n] - sigma == 0: # faut résoudre  $0 \cdot x[i] = 0$  ?
                print("Attention, solution pas unique!")
                x[i] = 42
            else: # faut résoudre  $0 \cdot x[i] = (\text{non nul})$  ?
                print("Erreur, pas de solution")
                return
        else: # sinon on peut diviser par M[i, i]
            x[i] = (M[i, n] - sigma) / M[i, i]
    return x
```

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** $n \times n$
($PM = M$ à une permutation de lignes près)

La matrice U résulte 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

Les opérations élémentaires sur une matrice carrée A peuvent être représentées par des **multiplications matricielles**.

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 :

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

Un produit de matrices du type $P_{(i,j)}$

= une **matrice de permutation** générale P

= 0 partout, sauf pour un seul élément = 1 dans chaque ligne et dans chaque colonne

La décomposition LU

Les opérations élémentaires sur une matrice carrée A peuvent être représentées par des **multiplications matricielles**.

Élimination de la j -ème colonne au-dessous du pivot : pour $i = j + 1, j + 2, \dots, n$, ajouter r_i -fois la ligne j du pivot à la ligne i (où $r_i = -a_{ij}/a_{jj}$)

$$A \rightarrow F_{(j;r_{j+1}, r_{j+2}, \dots, r_n)} A$$

avec

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

On note que l'inverse de la matrice $F_{(j;r_{j+1}, \dots, r_n)}$ est $F_{(j;r_{j+1}, \dots, r_n)}^{-1} = F_{(j;-r_{j+1}, \dots, -r_n)}$.

Calcul de la décomposition LU

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

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

- 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 ($P_{(i,j)} = \mathbb{1}$ si pas de changement nécessaire)
- Les $F_{(j;r_{j+1},\dots,r_n)}$ correspondent à l'élimination de la j -ème colonne

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

$$F_{(n-1;*)} P_{(*,n-1)} F_{(n-2;*)} \cdots P_{(*,2)} F_{(1;*)} P_{(*,2)} P_{(*,3)} \cdots P_{(*,n-1)} \\ \times P_{(*,n-1)} \cdots P_{(*,3)} P_{(*,2)} P_{(*,1)} A = U$$

On peut montrer : La première ligne de cette expression,

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

est triangulaire inférieure.

Calcul de la décomposition LU

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

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

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. On définit

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

et on multiplie par L à la gauche aux deux cotés :

$$L \hat{L} P A = L U$$

alors

$$P A = L U$$

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

- U = le résultat de l'algorithme de Gauss appliqué à A
- P = le produit de tous les $P_{(i,j)}$ (échanges de lignes effectués dans la procédure).
 - Démarrer l'algorithme de Gauss avec $P = \mathbb{1}$.
 - Quand on échange les lignes i et j , multiplier P à gauche par $P_{(i,j)}$.
- $L = \hat{L}^{-1} = P P_{(*,1)} F_{(1;*)}^{-1} P_{(*,2)} F_{(2;*)}^{-1} \cdots P_{(*,n-1)} F_{(n-1;*)}^{-1}$
avec les $F_{(i;*)}$ correspondant aux éliminations de colonnes.
 - Démarrer avec $L = \mathbb{1}$.
 - Quand on échange les lignes i et j , multiplier L à droite par $P_{(i,j)}$.
 - Lors de l'élimination de la colonne j , multiplier L à droite par le $F_{(j;*)}^{-1}$ correspondant.
Facile à calculer car $F_{(j;r_{j+1}, r_{j+2}, \dots, r_{n-1}, r_n)}^{-1} = F_{(j; -r_{j+1}, -r_{j+2}, \dots, -r_{n-1}, -r_n)}$.
 - Enfin multiplier L 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 : exemple

- 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

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

La décomposition LU : exemple

- Deuxième colonne : Maintenant

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

$U_{22} = 0$: 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 : exemple

Maintenant

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

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

Rien à faire pour la troisième ligne : $L \leftarrow L$.

Ajouter (-1) fois (deuxième ligne) à (quatrième ligne) :

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

La décomposition LU : exemple

- 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 : exemple

Enfin :

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

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

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

La décomposition LU

Deux fonctions auxiliaires :

```
import numpy as np

# P_ij = l'identité mais avec les lignes i et j échangées
def matriceP(n, i, j):
    P = np.identity(n)
    P[i, i] = P[j, j] = 0
    P[i, j] = P[j, i] = 1
    return P

# F = l'identité avec les éléments r_i au-dessous de F_jj
def matriceF(n, j, r):
    F = np.identity(n)
    F[j+1:, j] = r
    return F
```

La décomposition LU

```
def decomp_LU(A):
    n = A.shape[0]          # le nombre de lignes (et colonnes)
    U = np.copy(A)          # au début, U est A et P et L sont 1
    P, L = np.identity(n), np.identity(n)
    for i in range(n-1):    # boucle sur les colonnes
        for k in range(i, n): # trouver un pivot...
            if U[k, i] != 0:  # ...cet élément peut l'être ?
                U[[i, k], :] = U[[k, i], :] # échanger lignes...
                P = matriceP(n, k, i) @ P    # ...mettre à jour P
                L = L @ matriceP(n, k, i)    # ...et L...
                pivot = U[i, i]              # ...garder valeur du pivot...
                break                       # ...fin de la boucle sur k
        else:                          # toute la colonne est zéro ?
            continue                  # alors rien à faire pour celle-ci
    r = np.empty(n-i-1) # vecteur pour construire  $F^{-1}$ 
    for k in range(i+1, n): # éliminer sous la diagonale:
        facteur = -U[k, i]/pivot      # ajouter  $f * (\text{ligne } i)$ 
        U[k, :] += facteur * U[i, :]  # ...à la ligne k...
        r[k-i-1] = -facteur           # ...et mémoriser f
    L = L @ matriceF(n, i, r) # mettre à jour L
    return P, P @ L, U
```

La décomposition LU : applications

Une fois calculée la décomposition LU d'une matrice carrée A , on peut facilement résoudre **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 $L\vec{y} = \vec{b}'$:

$$y_1 = b'_1 / L_{11}$$

$$y_2 = (b'_2 - L_{21}y_1) / L_{22}$$

$$\vdots$$

$$y_n = (b'_n - \sum_{k < n} L_{nk}y_k) / L_{nn}$$

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

$$x_n = y_n / U_{nn}$$

$$x_{n-1} = (y_{n-1} - U_{n-1,n}x_n) / U_{n-1,n-1}$$

$$\vdots$$

$$x_1 = (y_1 - \sum_{k > 1} U_{1k}x_k) / U_{11}$$

La décomposition LU : applications

```
def resoudre(P, L, U, b): # trouve x dans  $LUx = Pb$ 
    n = U.shape[0] # les matrices sont n fois n
    bprime = P @ b # le vecteur  $b' = Pb$ 
    y = np.empty(n) # le vecteur y
    for i in range(n): # initialiser y
        sigma = 0
        for k in range(i): # k entre 0 et i-1
            sigma += L[i, k] * y[k]
        y[i] = (bprime[i] - sigma) / L[i, i]
    x = np.empty(n) # le vecteur x
    for i in range(n-1, -1, -1): # i entre 0 and n-1,
        sigma = 0 # en arrière
        for k in range(i+1, n): # k entre i+1 et n-1
            sigma += U[i, k] * x[k]
        x[i] = (y[i] - sigma) / 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
    P, L, U = decomp_LU(A)
    n = A.shape[0]
    Ainv = np.empty((n, n)) # l'inverse de A, vide au début
    iden = np.identity(n)   # l'identité (i-ème ligne = e_i)
    for i in range(n):
        Ainv[:, i] = resoudre(P, L, U, iden[i])
    return Ainv
```

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

- Le temps de calcul pour faire tourner la méthode de Gauss ou sa variante la décomposition LU est proportionnel à n^3 pour une matrice $n \times n$.

Si la taille de la matrice augmente par un facteur 10 \Rightarrow le programme prend $1000\times$ plus de temps ! Pour $n = 100$ cela prend une fraction d'une seconde, pour $n = 10\,000$ c'est déjà infaisable en pratique.

- 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 paramétrer les permutations par 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 telle que $S^T A S \equiv D$ est une matrice diagonale.

(Rappel : S orthogonale $\stackrel{\text{déf.}}{\Leftrightarrow} S^T S = \mathbb{1}$; D diagonale $\stackrel{\text{déf.}}{\Leftrightarrow} D$ 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 $A \stackrel{\text{déf.}}{\Leftrightarrow} \exists \vec{v} \in \mathbb{R}^n$ 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

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

avec $A_n = Q_n R_n$ la décomposition QR de A_n . On peut montrer que cette suite converge, sous certaines 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 .

Algorithme :

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

À vous de l'implémenter (\rightarrow exercices) !

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** :

- On construit Q par **orthonormalisation** des colonnes de A avec des transformations représentées 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 $\vec{v}_i \cdot \vec{v}_i = 1$.
- On définit la **projection** de \vec{w} sur \vec{v} :

$$\text{pr}_{\vec{v}} \vec{w} \equiv \frac{\vec{v} \cdot \vec{w}}{\|\vec{v}\|^2} \vec{v}$$

- 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 nul)
 - Soustraire de chaque colonne derrière \vec{v}_i sa projection sur \vec{v}_i ,
 $\vec{v}_j \leftarrow \vec{v}_j - \text{pr}_{\vec{v}_i} \vec{v}_j \quad \forall j > i$
 - Maintenant \vec{v}_i est orthogonal à tous les \vec{v}_j avec $j > i$ (et à toutes leurs 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

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

Le produit scalaire entre la 1ère et la 2ème colonne est $\frac{2}{3} \times 0 + \frac{2}{3} \times 4 + (-\frac{1}{3}) \times 2 = 2$. Il faut donc soustraire $2 \times (1\text{ère colonne})$ de la 2ème colonne.

Le produit scalaire entre la 1ère et la 3ème colonne est $\frac{2}{3} \times 1 + \frac{2}{3} \times (-1) + (-\frac{1}{3}) \times 3 = -1$, on soustrait alors $(-1) \times (1\text{ère colonne})$ 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

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

Le produit scalaire entre la 2ème et la 3ème colonne est 1. Il faut donc soustraire la 2ème de la 3ème colonne.

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

Maintenant les premières deux colonnes sont normalisées, orthogonales l'une à l'autre et orthogonales à la troisième.

Il ne reste qu'à normaliser cette dernière :

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

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):
    vnorme = norme(v)
    if vnorme < epsilon: # proj. sur vect. nul = vect. nul
        return np.zeros(v.shape[0])
    resultat = np.copy(v)
    resultat *= v @ w
    resultat /= vnorme**2
    return resultat
```

La décomposition QR

Orthogonalisation de Gram-Schmidt :

```
def gram_schmidt(A, epsilon=1.0E-10):  
    n = A.shape[0]  
    resultat = np.copy(A)    # tableau ou sera mis le résultat  
    for i in range(n):       # Gram-Schmidt modifiée:  
        v = resultat[:, i]   # pour tout vecteur de colonne v:  
        vnorme = norme(v)  
        if vnorme >= epsilon: # normaliser (sauf si nul)  
            v /= vnorme  
        for j in range(i+1, n): # de toute colonne derrière 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 = Q.T @ A  
    return Q, R
```


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 des vraies applications en physique numérique, il est préférable de se servir de ces méthodes optimisées au lieu de nos implémentations du cours « faites 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:  $P^{-1} A = L U$ 
Q, R = la.qr(A)            # décomposition QR
```

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

Ajustement

Généralités

- La méthode des moindres carrés

Algorithmes

- Régression linéaire
- Régression nonlinéaire : Méthodes de Gauss-Newton et de Levenberg-Marquardt

Description du problème :

Soient

- $(t_1, y_1) \dots (t_n, y_n)$ des **données**, p. ex. des données expérimentales,
- $f(t; \beta_1, \dots, \beta_p)$ une fonction (un "modèle") qui dépend de certains **paramètres** β_1, \dots, β_p .

On cherche les valeurs des paramètres telles que la fonction f correspondante décrit le mieux les données :

$$f(t_i; \vec{\beta}) \approx y_i \quad \forall i.$$

Ajustement

Exemple : La trajectoire d'un objet en **chute libre** est donnée par une fonction quadratique f du temps t ,

$$f(t; \beta_1, \beta_2, \beta_3) = \beta_1 + \beta_2 t + \beta_3 t^2 \stackrel{!}{=} y(t)$$

Ici

- $\beta_1 = y_0$ est la hauteur initiale à $t = 0$,
- $\beta_2 = v_0$ est la vitesse initiale,
- $\beta_3 = -g/2$ avec g l'accélération gravitationnelle.

Lors d'une expérience, on mesure

t[s]	y [m]
0	1
0.31	3
0.59	4
1.02	4
1.32	3
1.74	0

Comment en obtient-on les valeurs numériques de y_0 , v_0 et g ?

Méthode des moindres carrés

Méthode standard : Méthode des moindres carrés.

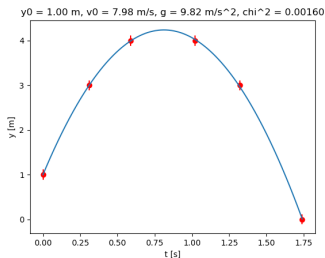
On définit le **résidu** r_i du point de données (t_i, y_i) par

$$r_i(\vec{\beta}) = f(t_i; \vec{\beta}) - y_i$$

et on cherche les valeurs des paramètres $\beta_1 \dots \beta_p$ telles que χ^2 , défini par

$$\chi^2(\vec{\beta}) \equiv \sum_{i=1}^n r_i^2(\vec{\beta})$$

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



Généralisation pour incertitudes variables

Si les données sont de la forme $(t_i, y_i \pm \sigma_i)$ avec des σ_i tous différents, alors il faut minimiser

$$\chi^2(\vec{\beta}) = \sum_{i=1}^n \left(\frac{r_i(\vec{\beta})}{\sigma_i} \right)^2, \quad r_i(\vec{\beta}) = f(t_i; \vec{\beta}) - y_i.$$

Méthode des moindres carrés pondérés.

Ainsi, les points de données avec grandes incertitudes σ_i contribuent à l'ajustement avec un poids moins important.

Généralisation pour des erreurs de mesure sur t

On a supposé que l'incertitude sur les t_i est négligeable ; sinon, il faut encore adapter la méthode.

Régression linéaire

Problème numérique : Comment minimiser χ^2 ?

Dans l'exemple de la chute libre, la fonction f dépend des paramètres $\beta_1, \beta_2, \beta_3$ **linéairement**,

$$f(t; \vec{\beta}) = \beta_1 + \beta_2 t + \beta_3 t^2.$$

Problème de **régression linéaire** : on cherche $\beta_1, \beta_2, \beta_3$ tels que

$$y_i \approx f(t_i; \vec{\beta}) \Leftrightarrow A\vec{\beta} \approx \vec{y}$$

où

$$A = \begin{pmatrix} 1 & x_1(t_1) & x_2(t_1) \\ 1 & x_1(t_2) & x_2(t_2) \\ 1 & x_1(t_3) & x_2(t_3) \\ \vdots & \vdots & \vdots \\ 1 & x_1(t_6) & x_2(t_6) \end{pmatrix} \quad \text{avec } x_1(t) = t, \quad x_2(t) = t^2.$$

Le système linéaire $A\vec{\beta} = \vec{y}$ est surdéterminé (6 équations pour seulement 3 inconnues ; $\chi^2 > 0$ génériquement). Le **meilleur ajustement** est donné par la **solution d'un système linéaire** de seulement 3 équations :

$$\boxed{A^T A \vec{\beta} = A^T \vec{y}}.$$

Preuve :

On souhaite minimiser $\vec{r}^2 = (A\vec{\beta} - \vec{y})^2$ par rapport à $\vec{\beta}$, alors on cherche le $\vec{\beta}$ où le gradient s'annule :

$$\frac{\partial}{\partial \beta_i} (A\vec{\beta} - \vec{y})^2 = 0.$$

Explicitement :

$$\begin{aligned}\frac{\partial}{\partial \beta_i} (A\vec{\beta} - \vec{y})^2 &= \frac{\partial}{\partial \beta_i} \sum_{ajk} (A_{aj}\beta_j - y_a)(A_{ak}\beta_k - y_a) \\ &= \sum_{ak} A_{ai}A_{ak}\beta_k + \sum_{aj} A_{aj}\beta_j A_{ai} - \sum_a y_a A_{ai} - \sum_a A_{ai}y_a \\ &= 2(A^T A\vec{\beta})_i - 2(A^T \vec{y})_i\end{aligned}$$

ce qui s'annule si $\vec{\beta}$ vérifie

$$A^T A\vec{\beta} = A^T \vec{y}.$$

Régression linéaire

```
import numpy as np
import numpy.linalg as la

# Les données:
t = np.array([0., .31, .59, 1.02, 1.32, 1.74])
y = np.array([1., 3., 4., 4., 3., 0.])

def x1(t):          # les variables prédicteur
    return t
def x2(t):
    return t**2

A = np.ones((6, 3)) # la matrice de coefficients
A[:, 1] = x1(t)      # (2ème colonne)
A[:, 2] = x2(t)      # (3ème colonne)

# Résoudre le système linéaire pour trouver les paramètres:
beta = la.solve(A.T @ A, A.T @ y)
y0, v0, g = beta[0], beta[1], -2 * beta[2]
```

Régression linéaire

Calculer χ^2 et tracer le résultat :

```
def f(t):  
    return y0 + v0 * t - g/2 * t**2  
  
r = f(t) - y          # les résidus  
chi2 = np.sum(r**2)  
print("chi^2 =", chi2)  
  
import matplotlib.pyplot as plt  
tpoints = np.linspace(0, 1.74, 100)  
plt.plot(t, y, 'ro') # tracer les données  
plt.plot(tpoints, f(tpoints)) # tracer la courbe théorique  
plt.xlabel("t [s]")  
plt.ylabel("y [m]")  
plt.show()
```

Régression linéaire : plusieurs variables indépendantes

Similaire pour plusieurs variables indépendantes

Exemple : on propose le modèle empirique

$$(\text{Note d'examen}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3$$

x_1 = pourcentage de TD manqués

x_2 = pourcentage de CM manqués

x_3 = heures de révision

Note	x_1	x_2	x_3
8	10	25	2
19.5	0	0	6
6	30	100	3
12.5	10	37.5	4
14.5	10	25	4
10	80	62.5	5
0	100	100	2
2.5	20	50	4
6.5	60	50	0

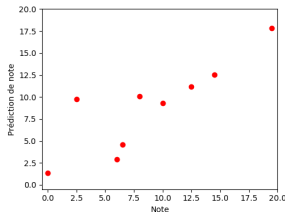
Régression linéaire : plusieurs variables indépendantes

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3$$

$$A = \begin{pmatrix} 1 & \begin{pmatrix} \\ \\ \\ \end{pmatrix} & \begin{pmatrix} \\ \\ \\ \end{pmatrix} & \begin{pmatrix} \\ \\ \\ \end{pmatrix} \\ 1 & \begin{pmatrix} \\ \\ \\ \end{pmatrix} & \begin{pmatrix} \\ \\ \\ \end{pmatrix} & \begin{pmatrix} \\ \\ \\ \end{pmatrix} \\ \vdots & \begin{pmatrix} \\ \\ \\ \end{pmatrix} & \begin{pmatrix} \\ \\ \\ \end{pmatrix} & \begin{pmatrix} \\ \\ \\ \end{pmatrix} \\ \vdots & \begin{pmatrix} \\ \\ \\ \end{pmatrix} & \begin{pmatrix} \\ \\ \\ \end{pmatrix} & \begin{pmatrix} \\ \\ \\ \end{pmatrix} \\ 1 & \begin{pmatrix} \\ \\ \\ \end{pmatrix} & \begin{pmatrix} \\ \\ \\ \end{pmatrix} & \begin{pmatrix} \\ \\ \\ \end{pmatrix} \end{pmatrix}$$

Matrice 9×4 (9 points de données, 4 paramètres à ajuster).

La solution de $A^T A \vec{\beta} = A^T \vec{y}$ donne $\beta_0 = 10.4$, $\beta_1 = -0.05$, $\beta_2 = -0.11$, $\beta_3 = 1.24$.



Régression linéaire

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

data = np.loadtxt("Regression_Notes")
datapoints = data.shape[0]

A = np.empty((datapoints, 4)) # matrice de coefficients
A[:, 0] = np.ones(datapoints) # première colonne = 1
A[:, 1:] = data[:, 1:]        # autres=variables prédicteur
y = data[:, 0]                # variables réponse

beta = np.linalg.solve(A.T @ A, A.T @ y) # ajustement

print("beta =", beta) # afficher résultat

def pred_y(x):            # prédiction pour un prédicteur donné
    return beta[0] + beta[1:] @ x

predictions = [pred_y(A[i, 1:]) for i in range(datapoints)]
plt.plot(y, predictions, 'ro')
plt.show()
```

Méthodes plus avancés :

Au lieu de directement résoudre les **équations normales**

$$A^T A \vec{\beta} = A^T \vec{y}$$

il peut être préférable de calculer la **décomposition en valeurs singulières** de la matrice A ,

$$A = U D V^T, \quad U = \text{orthogonale}, \quad D = \text{diagonale}, \quad V = \text{orthogonale}$$

et de calculer $\vec{\beta}$ avec U , V , D et \vec{y} .

Raison : stabilité numérique, problématique si $A^T A$ est (proche d'être) singulière.

Ici on n'a pas traité des algorithmes pour la décomposition en valeurs singulières, alors on ne détaillera pas la procédure.

Régression non linéaire

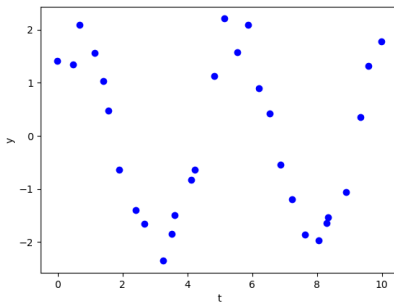
Dans les exemples avant, la fonction d'ajustement $f(t; \vec{\beta})$ dépendait des paramètres $\beta_1 \dots \beta_p$ **linéairement**.

Si la dépendance est plus compliquée, la minimisation de χ^2 devient plus difficile.

Exemple : Ajuster une fonction sinusoïdale,

$$f(t; \beta_1, \beta_2, \beta_3) = \beta_1 \sin(\beta_2 t + \beta_3)$$

où β_1 est l'amplitude, β_2 la fréquence, β_3 la phase à $t = 0$.



Régression non linéaire : L'algorithme de Gauss-Newton

On cherche un **minimum** de $\chi^2(\vec{\beta})$.

- 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** (potentiellement un **minimum**) 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 $g(\vec{\beta})$ une fonction de p variables $\vec{\beta}$. Pour trouver un point critique, itérer

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

où la **matrice hesséenne** $H(\vec{\beta})$ est

$$H = \begin{pmatrix} \frac{\partial^2 g}{\partial \beta_1^2} & \frac{\partial^2 g}{\partial \beta_1 \partial \beta_2} & \frac{\partial^2 g}{\partial \beta_1 \partial \beta_3} & \cdots & \frac{\partial^2 g}{\partial \beta_1 \partial \beta_p} \\ \frac{\partial^2 g}{\partial \beta_2 \partial \beta_1} & \frac{\partial^2 g}{\partial \beta_2^2} & \frac{\partial^2 g}{\partial \beta_2 \partial \beta_3} & \cdots & \frac{\partial^2 g}{\partial \beta_2 \partial \beta_p} \\ \vdots & & & & \vdots \\ \frac{\partial^2 g}{\partial \beta_p \partial \beta_1} & \frac{\partial^2 g}{\partial \beta_p \partial \beta_2} & \frac{\partial^2 g}{\partial \beta_p \partial \beta_3} & \cdots & \frac{\partial^2 g}{\partial \beta_p^2} \end{pmatrix}$$

Régression non linéaire : L'algorithme de Gauss-Newton

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

$$\chi^2(\beta_1 \dots \beta_p) = r_1(\vec{\beta})^2 + r_2(\vec{\beta})^2 + \dots + r_n(\vec{\beta})^2 = \vec{r} \cdot \vec{r}$$

avec les n résidus

$$r_i(\vec{\beta}) = f(t_i; \vec{\beta}) - y_i$$

Alors

$$\vec{\nabla} \chi^2 = \begin{pmatrix} 2 r_1 \frac{\partial r_1}{\partial \beta_1} + 2 r_2 \frac{\partial r_2}{\partial \beta_1} + \dots + 2 r_n \frac{\partial r_n}{\partial \beta_1} \\ 2 r_1 \frac{\partial r_1}{\partial \beta_2} + 2 r_2 \frac{\partial r_2}{\partial \beta_2} + \dots + 2 r_n \frac{\partial r_n}{\partial \beta_2} \\ \vdots \\ 2 r_1 \frac{\partial r_1}{\partial \beta_p} + 2 r_2 \frac{\partial r_2}{\partial \beta_p} + \dots + 2 r_n \frac{\partial r_n}{\partial \beta_p} \end{pmatrix} = 2 \vec{r} J$$

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

$$J_{ij} = \frac{\partial r_i}{\partial \beta_j}.$$

De plus,

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

(en supposant que les termes $\vec{r} \cdot \frac{\partial^2 \vec{r}}{\partial \beta_i \partial \beta_j}$ sont négligeables).

Régression non linéaire : L'algorithme de Gauss-Newton

Résumé : On a trouvé

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

et

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

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

$$r_i(\vec{\beta}) = f(t_i; \vec{\beta}) - y_i$$

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

$$J = \frac{\partial \vec{r}}{\partial \vec{\beta}} = \frac{\partial f}{\partial \vec{\beta}}.$$

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

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

Méthode de Gauss-Newton.

En pratique :

- Calculer analytiquement les dérivées $\frac{\partial f}{\partial \beta_j}$.
- Commencer avec un ensemble de paramètres $\vec{\beta}$ au choix
(mais aussi proche que possible de l'optimum pour améliorer la convergence)
- Mettre à jour les $r_i = f(t_i; \vec{\beta}) - y_i$ et les $J_{ij} = \frac{\partial f}{\partial \beta_j}(t_i; \vec{\beta})$.
- Remplacer $\vec{\beta} \leftarrow \vec{\beta} - (J^T J)^{-1} \vec{r} J$.

Équivalent, à préférer en pratique (car plus stable) : calculer le nouveau $\vec{\beta}$ avec la **solution d'un système linéaire**, trouvée p.ex. par la méthode de Gauss :

$$\vec{\beta}_{\text{nouveau}} = \vec{\beta}_{\text{ancien}} + \vec{\delta}, \quad \vec{\delta} = (\text{solution de } J^T J \vec{\delta} = -\vec{r} J).$$

- Itérer ces dernières deux étapes jusqu'à la convergence.
S'arrêter lorsque $\|\vec{\delta}\| < \epsilon$.

Régression non linéaire : L'algorithme de Gauss-Newton

Avec les fonctions $\frac{\partial f}{\partial \beta_j}$ données dans une liste `gradf` :

```
import numpy as np
import gauss      # pour la fonction gauss() du cours

def gauss_newton(t, y, f, gradf, beta0, epsilon=1.E-4):
    beta = np.copy(beta0)          # les paramètres à ajuster
    delta = np.ones(len(beta))    # diff. entre deux itérations

    while np.sqrt(np.sum(delta**2)) > epsilon:
        r = f(t, beta) - y        # les résidus
        J = np.array([df(t, beta) for df in gradf]).T # matr. J
        delta = gauss.gauss(J.T @ J, - r @ J) # sol. du système
        beta += delta

    chi2 = np.sum(r**2)            # chi^2 après minimisation
    return beta, chi2
```

Algorithme de Gauss-Newton : Exemple d'application

Par exemple, pour la fonction sinusoïdale ci-dessus :

$$f(t; \vec{\beta}) = \beta_1 \sin(\beta_2 t + \beta_3)$$

et donc

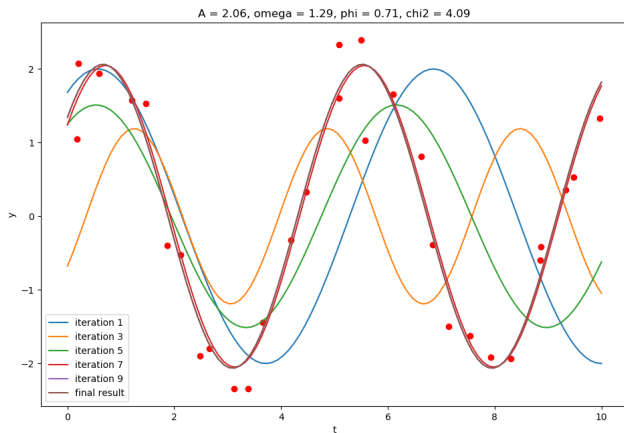
$$\frac{\partial f}{\partial \beta_1}(t; \vec{\beta}) = \sin(\beta_2 t + \beta_3), \quad \frac{\partial f}{\partial \beta_2}(t; \vec{\beta}) = \beta_1 \cos(\beta_2 t + \beta_3) t,$$
$$\frac{\partial f}{\partial \beta_3}(t; \vec{\beta}) = \beta_1 \cos(\beta_2 t + \beta_3) .$$

```
# la fonction modèle
def f(t, beta):
    return beta[0] * np.sin(beta[1] * t + beta[2])

# ses dérivées partielles par rapport aux paramètres
df = [lambda t, beta: np.sin(beta[1]*t + beta[2]),
      lambda t, beta: beta[0]*np.cos(beta[1]*t + beta[2])*t,
      lambda t, beta: beta[0]*np.cos(beta[1]*t + beta[2])]

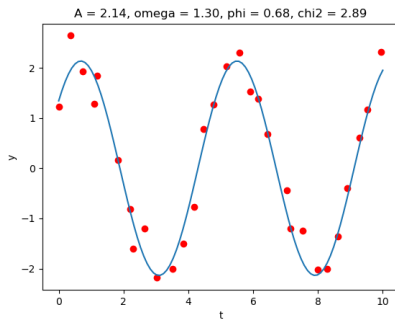
data = np.loadtxt('noisysin.txt') # les données
beta, chi2 = gauss_newton(data[:, 0], data[:, 1], f, df,
                          np.array([2., 1., 0.]))
```

Algorithme de Gauss-Newton : Exemple d'application



Algorithme de Gauss-Newton : Exemple d'application

$$f(t; A, \omega, \phi) = A \sin(\omega t + \phi)$$



Régression non linéaire : Algorithme de Levenberg-Marquardt

Faiblesse de la méthode de Gauss-Newton : si on commence avec des valeurs de départ pour $\vec{\beta}$ **trop loin du minimum**, alors l'itération **ne le trouvera pas**.

Ce problème est amélioré avec une modification de l'algorithme menant à la **méthode de Levenberg-Marquardt**.

Gauss-Newton :

$$\vec{\beta}_{\text{nouveau}} = \vec{\beta}_{\text{ancien}} + \vec{\delta}, \quad J^T J \vec{\delta} = -\vec{r} J$$

Levenberg-Marquardt :

$$\vec{\beta}_{\text{nouveau}} = \vec{\beta}_{\text{ancien}} + \vec{\delta}, \quad \left(J^T J + \lambda \mathbb{1} \right) \vec{\delta} = -\vec{r} J$$

où $\lambda \geq 0$ est un paramètre dit **d'amortissement**, à adapter à chaque itération.

- Si $\lambda \rightarrow 0$, la méthode s'approche à celle de Gauss-Newton.
- Pour des grands λ , on s'approche à la **méthode du gradient** : la variation $\vec{\delta}$ suit la direction de la plus forte pente $-\vec{r} J \propto -\vec{\nabla} \chi^2$.

Algorithme :

- Choisir une assez petite valeur initiale de λ (disons 10^{-4}).
- Faire tourner l'algorithme modifié de Gauss-Newton en remplaçant $J^T J \rightarrow J^T J + \lambda \mathbb{1}$.
- A chaque itération, calculer χ^2 .
 - Si χ^2 a grandi par rapport à l'itération précédente, retourner à l'ancien $\vec{\beta}$ et refaire avec $\lambda \leftarrow 10 \lambda$.
 - Si χ^2 a diminué, garder le nouveau $\vec{\beta}$ et continuer avec $\lambda \leftarrow \lambda/10$.
- S'arrêter dès que χ^2 ne diminue quasiment plus entre deux itérations (p.ex. diminue par moins que $\approx 10^{-2}$).

Propriétés de la méthode de Levenberg-Marquardt :

- Par rapport à Gauss-Newton, convergence **légèrement moins rapide** mais **plus stable**.
- Beaucoup de variations et d'optimisations existent, par exemple :
 - d'autres prescriptions pour adapter le paramètre d'amortissement λ
 - utiliser la combinaison $J^T J + \lambda \text{diag}(J^T J)$ au lieu de $J^T J + \lambda \mathbb{1}$, où $\text{diag}(J^T J)$ a les mêmes valeurs que $J^T J$ sur la diagonale et est 0 ailleurs

Equations différentielles ordinaires

Algorithmes

- Méthode d'Euler
- Méthode de Runge-Kutta classique

Généralités

- Application aux équations de mouvement des systèmes en mécanique classique

Equations différentielles ordinaires

Une **équation différentielle ordinaire du premier ordre** est une équation

- dont l'inconnue est une fonction $x(t)$ d'un **seul** paramètre t ("ordinaire")
- qui implique la **dérivée** de cette fonction inconnue, $\dot{x}(t) \equiv \frac{dx}{dt}$ ("différentielle")
- mais qui n'implique pas les dérivées d'ordre supérieur ("premier ordre").

On s'intéressera ici aux **problèmes de Cauchy** (ou "problèmes aux valeurs initiales") où on donne une EDO et une **condition initiale** $x(0) = x_0$.

Théorème de Cauchy-Lipschitz : On donne l'EDO

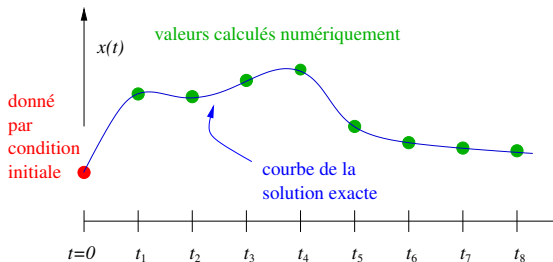
$$\dot{x}(t) = f(x(t), t)$$

avec f une fonction suffisamment régulière (par exemple, dérivable par rapport à son premier argument). Alors, pour tout $x_0 \in \mathbb{R}$, il existe un voisinage U de $t = 0$ et une fonction unique x sur U qui vérifie l'EDO ainsi que la condition initiale $x(0) = x_0$.

Equations différentielles ordinaires

Notre objectif sera de trouver une **approximation numérique** de la fonction inconnue $x(t)$ = un tableau de valeurs de fonction approximatives $x(t_1), x(t_2), \dots, x(t_n)$.

Ces valeurs seront successivement calculées à partir de la première valeur x_0 à $t = 0$, qui est donnée par la condition initiale.



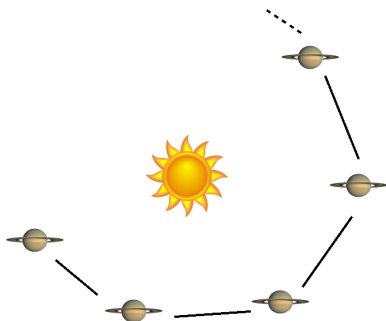
Equations différentielles ordinaires

Motivation physique importante : Les équations de mouvement

$$\vec{F} = m\vec{a}$$

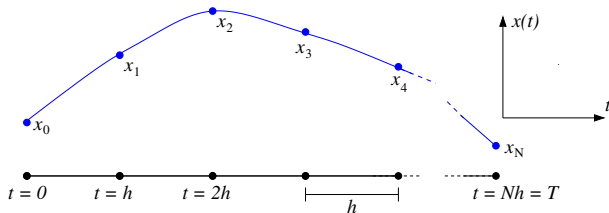
en **mécanique** peuvent s'écrire comme un système d'EDO du premier ordre. Les fonctions inconnues sont les **positions** et les **vitesse**s (ou plus généralement les coordonnées généralisées et leurs moments conjugués).

Avec les positions et vitesses donnés à $t = 0$, on pourra les trouver pour tout t numériquement.



Méthode d'Euler

Pour l'EDO $\dot{x}(t) = f(x(t), t)$ avec la condition initiale $x(0) = x_0$, on cherche la solution sur l'intervalle $[0, T]$. Notre objectif sera alors de créer un tableau de N valeurs de fonction $x_1 = x(h), x_2 = x(2h), x_3 = x(3h), \dots, x_N = x(Nh)$ avec $Nh = T$.



Méthode d'Euler : Développement limité pour h suffisamment petit,

$$x(h) = x(0) + \dot{x}(0)h + \frac{1}{2}\ddot{x}(0)h^2 + \frac{1}{3!}\ddot{\ddot{x}}(0)h^3 + \dots$$

On néglige les termes $\mathcal{O}(h^2)$ et supérieurs ; on substitue l'EDO $\dot{x}(0) = f(x(0), 0)$ et la condition initiale $x(0) = x_0$:

$$x_1 = x_0 + h f(x_0, 0) .$$

De même : Une fois x_n connu, on obtient x_{n+1} par développement limité et substitution,

$$x_{n+1} = x_n + \dot{x}_n h = x_n + h f(x_n, nh) .$$

Méthode d'Euler

On obtient ainsi successivement les $x_1, x_2, x_3, \dots, x_N = x(T)$:

$$x_{n+1} = x_n + h f(x_n, nh)$$

```
# Résoudre l'EDO dx/dt = f(x(t), t) avec la méthode d'Euler
#
# Arguments:
# f = fonction à deux arguments = membre de droite de l'EDO
# x0 = x(0) condition initiale
# h = pas d'incrément en t
# N = nombre de points à calculer
#
# Renvoie une liste de N+1 valeurs [x(0), ..., x(Nh)]
def euler(f, x0, h, N):
    x = [x0] # une liste qui au début contient seulement x0
    xn = x0
    for n in range(N):
        xn += h * f(xn, h*n)
        x += [xn] # ajouter xn à la liste des x
    return x
```

Méthode d'Euler : Un simple exemple

Pour p.ex. $f(x(t), t) = x(t) + t$ et $x_0 = 0$:

$$\dot{x}(t) = x(t) + t, \quad x(0) = 0.$$

Solution analytique :

$$x(t) = e^t - t - 1.$$

Solution numérique sur $[0, 1]$ avec incrément $h = 10^{-2}$:

```
from euler import euler # la fct. euler du fichier euler.py

def f(x, t):
    return x + t

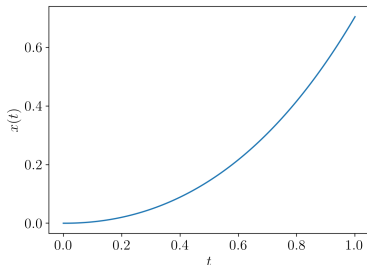
solution = euler(f, 0.0, 1.E-2, 100)
```

Affichage :

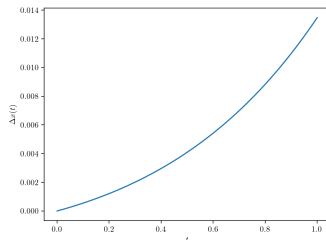
```
import matplotlib.pyplot as plt

plt.plot(np.linspace(0, 1, 101), solution)
plt.show()
```

Méthode d'Euler : Un simple exemple



Solution numérique $x(t)$



$\Delta x(t) = |x_{\text{numérique}}(t) - x_{\text{analytique}}(t)|$

- L'erreur numérique Δx à $t = T = 1$ est ≈ 0.014 et alors du même ordre que $h = 0.01$. Explication : à chaque pas on néglige des termes $\mathcal{O}(h^2)$ dans le développement limité ; il y a $N = 1/h$ pas pour arriver à $t = 1$.
- La méthode d'Euler est une **méthode du premier ordre** : l'erreur numérique globale est de l'ordre h .
- Pour améliorer la précision numérique par un facteur 2, il faudrait calculer $\sim 2\times$ plus de points.

Exemple physique

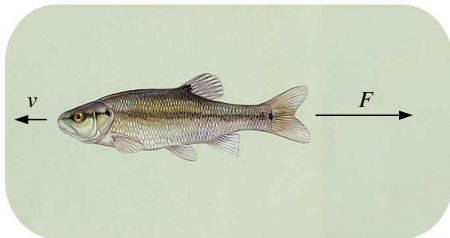
On regarde un objet qui se déplace dans un milieu fluide et qui est ralenti par une force de traînée dépendant de la vitesse, $F(v)$,

$$F(v) = f_1 v + f_2 v^2, \quad f_i = \text{ctes.}$$

L'équation de mouvement est alors $F = ma = m\dot{v} = f_1 v + f_2 v^2$, ou

$$\dot{v} = -\alpha v - \beta v^2$$

$\alpha = -\frac{f_1}{m}$, $\beta = -\frac{f_2}{m}$ positives (traînée opposée au mouvement).



Exemple physique

$$\dot{v} = -\alpha v - \beta v^2$$

- Petites vitesses, écoulement laminaire : $\dot{v} = -\alpha v$ (loi de Stokes) avec solution $v(t) = v_0 e^{-\alpha t}$.
- Grandes vitesses, écoulement turbulent : traînée $\propto v^2$, $\dot{v} = -\beta v^2$ donc $v(t) = \frac{v_0}{1 + \beta v_0 t}$.
- Cas général : exemple d'une **équation différentielle de Bernoulli**,

$$v(t) = \frac{\alpha v_0}{e^{\alpha t}(\alpha + \beta v_0) - \beta v_0}.$$

Calculons une solution numérique pour comparer avec la solution exacte.

Exemple physique

```
# Constantes physiques:
alpha = 1.
beta = .5
v0 = 2.

# Constantes numériques:
T = 5.          # intervalle de temps
N = 1000        # nombre de pas à calculer
h = T / N       # pas d'incrément

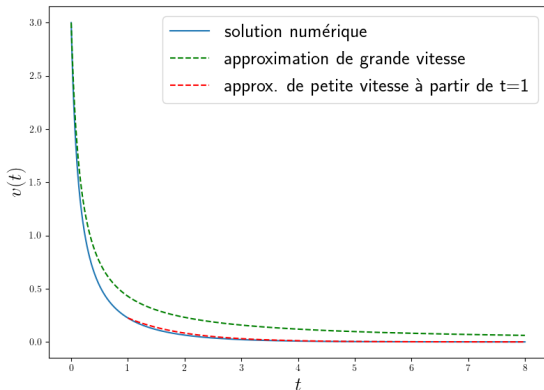
def f(v, t):
    return -alpha * v - beta * v**2

from euler import euler
solution = euler(f, v0, h, N)

import numpy as np
import matplotlib.pyplot as plt
plt.plot(np.linspace(0, T, N+1), solution)
plt.show()
```

Exemple physique

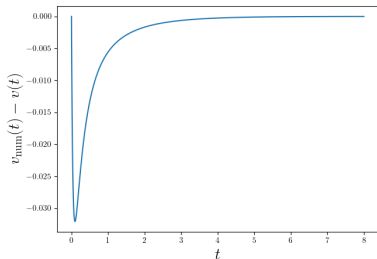
Solution numérique pour $\alpha = 1$, $\beta = 2$, $v_0 = 3$ calculée avec $N = 1000$ points :



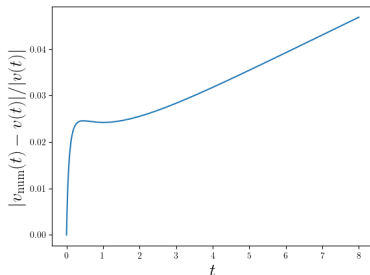
Exemple physique

Différence entre la solution numérique pour $\alpha = 1$, $\beta = 2$, $v_0 = 3$ avec $N = 1000$ points et la solution exacte $v(t) = \frac{\alpha v_0}{e^{\alpha t}(\alpha + \beta v_0) - \beta v_0}$

Erreur absolue



Erreur relative



Un système de n équations différentielles du premier ordre avec n fonctions inconnues :

$$\dot{x}(t) = f_x(x(t), y(t), z(t), \dots, t)$$

$$\dot{y}(t) = f_y(x(t), y(t), z(t), \dots, t)$$

$$\dot{z}(t) = f_z(x(t), y(t), z(t), \dots, t)$$

$$\vdots$$

Il faut n conditions initiales $x(0) = x_0, y(0) = y_0, z(0) = z_0, \dots$ pour un problème de Cauchy bien posé.

Pour le résoudre, on se sert de la même méthode qu'avant, en regroupant les n fonctions inconnues x, y, z, \dots et les n fonctions des membres de droite f_x, f_y, f_z, \dots dans des fonctions vectorielles \vec{u} et \vec{f} :

$$\dot{\vec{u}}(t) = \vec{f}(\vec{u}(t), t)$$

Après développement limité et substitution :

$$\vec{u}(t+h) = \vec{u}(t) + h\vec{f}(\vec{u}(t), t) + \mathcal{O}(h^2)$$

Méthode d'Euler n -dimensionnelle.

EDO du second ordre

Application importante : équations du **second ordre** qui peuvent toujours se transformer en **deux équations du premier ordre**. Par exemple :

$$\ddot{x} = \frac{1}{m} F(x, \dot{x}, t)$$

Introduire une fonction inconnue auxiliaire $v(t)$ par

$$v = \dot{x}$$

On obtient un système de **deux EDO du premier ordre** :

$$\dot{x} = v$$

$$\dot{v} = \frac{1}{m} F(x, v, t)$$

Plus généralement : en d dimensions,

$$\ddot{\vec{x}} = \frac{1}{m} \vec{F}(\vec{x}, \dot{\vec{x}}, t)$$

équivalent aux $2d$ équations du **premier ordre**

$$\dot{\vec{x}} = \vec{v}$$

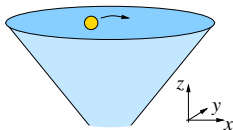
$$\dot{\vec{v}} = \frac{1}{m} \vec{F}(\vec{x}, \vec{v}, t)$$

Exemple

Une bille de masse m se déplace sans friction sur l'intérieur d'un étonnoir conique, dont la surface est décrite, en coordonnées cartésiennes, par

$$x^2 + y^2 = \tan^2(\alpha) z^2$$

avec $2\alpha =$ l'angle d'ouverture du cône. La gravité agit en direction négative des z . On souhaite résoudre les équations de mouvement.



On utilise des coordonnées cylindriques (r, ϕ, z) . Avec la vitesse radiale $v_r = \dot{r}$ et le moment cinétique $\ell = mr^2\dot{\phi}$ (conservé ici) on obtient

$$\begin{aligned}\dot{r} &= v_r \\ \dot{v}_r &= \frac{\ell^2 \sin^2 \alpha}{m^2} \frac{1}{r^3} - g \sin \alpha \cos \alpha \\ \dot{\phi} &= \frac{\ell}{mr^2}\end{aligned}$$

Exemple

Importer la bibliothèque numpy, initialiser les constantes :

```
import numpy as np

# Constantes physiques:
g = 9.81      # accélération gravitationnelle en m/s^2
alpha = np.pi/4 # 1/2 * angle d'ouverture
m = 1.E-3     # masse de la bille en kg
# Constantes numériques:
h = 1.E-4     # pas d'incrément
N = 50000     # nombre de pas à calculer
# Conditions initiales:
r0 = .1       # rayon
v0 = .2       # vitesse radiale
phi0 = 0.0    # angle
omega0 = 4.   # vitesse angulaire
# Constantes dépendantes:
L = m * r0**2 * omega0 # moment cinétique
sa, ca = np.sin(alpha), np.cos(alpha)
```

Exemple

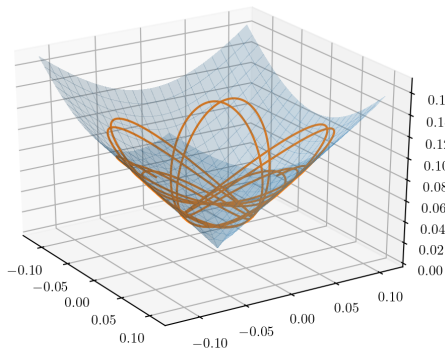
Résoudre les équations de mouvement par la méthode d'Euler :

```
def euler(f, u0, h, N):
    u = np.empty([N+1, 3]) # variables dynamiques
    u[0] = u0              # u[i] = ligne i du tableau u
    for n in range(N):
        u[n+1] = u[n] + h * f(u[n])
    return u

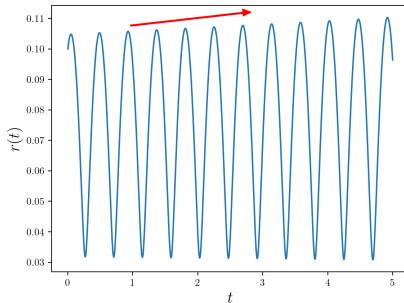
# La fonction vectorielle des membres de droite des e.d.m.
# u = un vecteur à 3 composantes (r, v et phi au temps t)
def f(u):
    r, v, phi = u
    fr = v
    fv = L**2 * sa**2 / (m**2 * r**3) - g * sa * ca
    fphi = L / (m * r**2)
    return np.array([fr, fv, fphi])

u0 = np.array([r0, v0, phi0])
solution = euler(f, u0, h, N)
```

Exemple



Trajectoire pendant 5 s



$r(t)$

Visiblement l'amplitude de la solution numérique pour r est croissante (voir **flèche**) malgré la conservation d'énergie : **artéfact numérique**. Pour l'éviter on peut soit **réduire h** et **augmenter N** soit **utiliser une méthode numérique plus puissante**.

Méthode de Runge-Kutta classique

Problème : Résoudre le problème de Cauchy $\dot{x} = f(x(t), t)$, $x(0) = x_0$.

Euler : Étant donné $x(t)$, pour obtenir $x(t+h)$: **développement limité** autour de $x(t)$, ne retenir que le **premier terme**,

$$x(t+h) = x(t) + h \dot{x}(t) + \mathcal{O}(h^2) = x(t) + h f(x(t), t) + \mathcal{O}(h^2).$$

Runge-Kutta : Étant donné $x(t)$, pour obtenir $x(t+h)$: combinaisons linéaires de **plusieurs développements limités** autour des valeurs intermédiaires $x(t+\tau_i)$ avec $0 \leq \tau_i \leq h$. Pour p_i, ξ_i, τ_i bien choisis, l'erreur local devient $\mathcal{O}(h^N)$ avec $N > 1$:

$$x(t+h) = x(t) + h \sum_{i=1}^r p_i k_i + \mathcal{O}(h^N), \quad k_i = f(x + \xi_i, t + \tau_i), \quad 0 < p_i < 1.$$

Variante la plus importante : **Méthode de R-K du 4ème ordre**, "RK4", "RK classique"

$$x(t+h) = x(t) + h \left(\frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 \right) + \mathcal{O}(h^5)$$

$$k_1 = f(x(t), t), \quad k_2 = f\left(x(t) + \frac{h}{2}k_1, t + \frac{h}{2}\right),$$

$$k_3 = f\left(x(t) + \frac{h}{2}k_2, t + \frac{h}{2}\right), \quad k_4 = f(x(t) + h k_3, t + h).$$

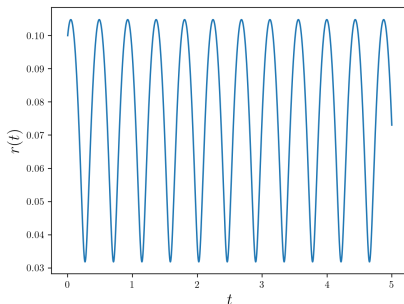
Méthode de Runge-Kutta classique

- La méthode RK4 est **beaucoup plus précise** que celle d'Euler.
- Elle nécessite **plus d'évaluations de la fonction f** (4 par pas — une seule par pas pour Euler), mais en pratique cela est plus que compensée par le fait qu'il faut **beaucoup moins de pas** pour atteindre la même précision.
- Elle est presque aussi facile à implémenter que la méthode d'Euler (4 lignes de plus).
- Elle aussi peut être appliquée aux systèmes de plusieurs EDO.

Exemple : Pour l'exemple de la bille dans l'étonnoir, remplacer la fonction euler par une fonction rk4 :

```
def rk4(f, u0, h, N):  
    u = np.empty([N+1, 3]) # variables dynamiques  
    u[0] = u0               # u[i] = ligne i du tableau u  
    for n in range(N):  
        k1 = f(u[n])  
        k2 = f(u[n] + h/2 * k1)  
        k3 = f(u[n] + h/2 * k2)  
        k4 = f(u[n] + h * k3)  
        u[n+1] = u[n] + h * (k1/6 + k2/3 + k3/3 + k4/6)  
    return u
```

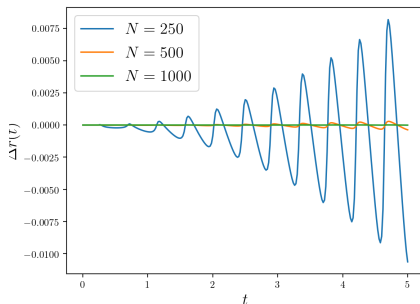
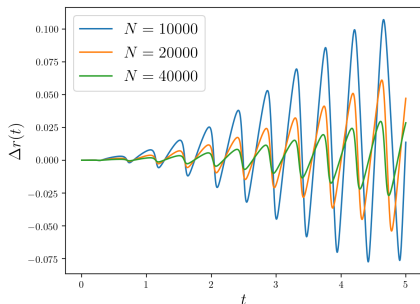
Méthode de Runge-Kutta classique



Plus de croissance artificielle visible de l'amplitude grace à une **meilleure précision de l'algorithme**.

Précision numérique, Euler vs. RK4

- Avec la méthode d'**Euler** on néglige des termes $\mathcal{O}(h^2)$ à chaque pas. Après $N = \mathcal{O}(1/h)$ pas, l'erreur accumulée est alors $\mathcal{O}(h)$.
- Redoubler le nombre de pas \Rightarrow l'erreur est réduite par la moitié
- Avec la méthode **RK4** les termes négligés sont $\mathcal{O}(h^5)$, l'erreur accumulée est $\mathcal{O}(h^4)$ ("methode du 4ème ordre")
- Redoubler le nombre de pas \Rightarrow l'erreur est réduite par un facteur $2^4 = 16$.



Erreur en fonction de N : Euler

RK4

Résumé : EDO, problèmes de Cauchy

Pour numériquement résoudre un problème un problème aux valeurs initiales en mécanique :

- Choisir un système de coordonnées (en prenant en compte la symétrie du système, les contraintes éventuelles. . .)
- Trouver les équations de mouvement, par exemple avec $\vec{F} = m\vec{a}$.
- Les transformer en système d'équations du premier ordre.
- Résoudre les e.d.m. avec la ~~méthode d'Euler~~ **méthode RK4** (plus puissante).
- Résultat : un tableau de coordonnées et vitesses/quantités de mouvement à différents t .