Chapitre 1

Les maths avec Python

Objectifs

- **Les incontournables :**
  - Savoir construire une liste (ou un ensemble) et savoir la (ou le) manipuler.
  - Savoir bâtir une procédure simple.
  - Savoir utiliser les boucles conditionnelles notamment pour définir une fonction définie par plusieurs types d'images selon la valeur de la variable.
  - Savoir manipuler des entiers avec Python et en particulier savoir diviser ou sommer.
  - Savoir étudier et tracer des graphes de fonctions numériques ou des courbes paramétrées et tracer une famille de graphes sur le même dessin.
  - Savoir manipuler ou faire des opérations sur les polynômes avec Python.
  - Savoir déterminer numériquement des intégrales, des solutions de $f(x) = 0$ ou d'équations différentielles avec des fonctions Python prédéfinies.
  - Savoir afficher une liste de termes d'une suite réelle avec une boucle.
  - Savoir afficher et manipuler des matrices et donc connaître les fonctions Python qui fournissent les opérations principales liées aux matrices ou la résolution de systèmes linéaires.
  - Savoir déterminer avec Python les valeurs propres et les vecteurs propres d'une matrice carrée donnée.
  - Savoir manipuler des vecteurs, calculer leur produit scalaire ou vectoriel.
  - Savoir tracer une ligne de niveau $f(x, y) = \lambda$ ou une surface.
  - Savoir simuler une suite de tirages et les fonctions Python fournissant les lois de probabilités principales.
- **Et plus si affinités...**
  - Savoir créer les procédures permettant les opérations élémentaires des matrices.
  - Savoir résoudre $f(x) = 0$ par des méthodes algorithmiques traduites en Python (Dichotomie, Newton) sans utiliser des fonctions prédéfinies.
  - Savoir déterminer numériquement une intégrale par la formule des Rectangles traduite en Python et calculer la marge d'erreur sans utiliser des fonctions prédéfinies.
  - Savoir résoudre une équation différentielle linéaire d'ordre 1 ou d'ordre 2 par la méthode algorithmique d'Euler traduite en Python sans utiliser des fonctions prédéfinies.
  - Savoir résoudre un système différentiel linéaire d'ordre 1 en utilisant Python.
  - Savoir calculer la valeur propre de plus grand module.
  - Savoir décomposer une matrice sous la forme QR à l'aide de Python.
  - Savoir programmer l'algorithme d'orthonormalisation de Gram-Schmidt.

Résumé de cours

L'objectif du chapitre est double. C'est d'abord de développer les quelques algorithmes écrits dans le programme officiel de 2TSI(et de rappeler les principaux vus en 1TSI). Et bien entendu de les faire « tourner » avec Python, ce qui permet d'illustrer le cours de Maths au fur et à mesure de son avancée. Puis c'est de préparer l'étudiant à la prise en compte de l'informatique dans les épreuves de Mathématiques. Citons entre-autre l'Oral de Mathématiques II de Centrale-SupElec qui propose l'utilisation d'un ordinateur équipé de Python (distribution Pyzo) et de Scilab.

Les fonctions de bases, les modules et les attributs

On commence par télécharger une version de Python (Python 3.4, Python 2.7 ou autre) généralement au travers d'un environnement de développement.
Parmi les plus connus citons Spyder, winpython, Pyzo et IDLE.

1. Les fonctions de base

Quand on télécharge le Python « sec », on peut utiliser un certain nombre de fonctions rentrées d'office. On peut citer entre autre celles qui enrichissent une procédure : *input* ou *return* ou encore *print*. Dans ce chapitre, vous pourrez faire un listing des plus utilisées.

2. Les modules et les sous-modules

En plus de l'environnement et du langage Python lui-même, il existe des modules (on dit aussi bibliothèques) contenant un certain nombre de fonctions prédéfinies.

Dans le rapport de jury de l'Oral de Math II de Centrale-Supelec, il est écrit :
« concernant Python, **le jury attend que les candidats soient familiarisés avec l'utilisation des bibliothèques numpy, scipy et de la bibliothèque de visualisation associée matplotlib.** Cependant, *aucune connaissance de fonctions particulières n'est exigée* ; les candidats auront à leur disposition, pendant l'épreuve, des documents listant un certain nombre de fonctions qui peuvent être utilisées pour résoudre les exercices proposés. » La consigne est quasiment la même pour l'épreuve Orale de Mathématiques et Algorithmique des Arts et Metiers ParisTech. En tout cas découvrir toutes ces fonctions et d'autres qui peuvent être utiles le jour de l'Oral n'est pas très efficace !

Comment charger un module ou une fonction d'un module ?
Soit un module générique que nous appelerons *package*. Il contient toutes les expressions et fonctions python définis dans le fichier *package.py*.
Pour importer *package* ou des fonctions ou des classes incluses dans *package*, on applique une des règles suivantes :
import package : accès à la fonction $fonction$ de *package* en tapant $package.fonction$.
import package as pa : accès à la fonction $fonction$ de *package* en tapant $pa.fonction$.
On dit que *pa* est un alias de *package*.
$from\ package\ import\ *$: accès à la fonction $fonction$ de *package* en tapant $fonction$ (sans préfixage).
$from\ package\ import\ fonction$: accès à la seule fonction $fonction$ de *package* en tapant $fonction$.
Parfois la fonction qui nous intéresse est dans un sous-module $sous-package$ d'un module principal *package* et on tape :

import package.sous − package as sp ou *from package.sous − package import fonction*

Comment s'informer sur le contenu d'un module ou sur une fonction d'un module ?
dir( ) liste les modules ou fonctions déjà chargés à un moment donné.
Après avoir charché le module *package* avec *import package* :
dir(*package*) liste les fonctions et constantes du module *package*.
help(*package*) renvoie des informations sur les fonctions du module *package*.
help(*package.fonction*) renvoie des informations sur *fonction* du module *package*.

Donnons maintenant les modules incontournables.

- Le module **math** ; il contient les principales fonctions et constantes mathématiques usuelles.
- Le module **cmath** ; il permet d'utiliser certaines commandes relatives aux nombres complexes. Il possède beaucoup de fonctions en commun avec *math*
  On tape *import cmath*. Voir la **méthode 1.2**.
- Le module **numpy** ; il permet de faire du calcul scientifique et de manipuler des tableaux.
  On l'importe en totalité avec l'instruction : *import numpy as np* Son alias usuel est *np*.
  Notons ici une fonction de *numpy* bien pratique. C'est *copy* qui permet de copier une liste (ou un tableau) donnée en argument, de travailler sur cette copie sans modifier la liste initiale.
  Voir la **méthode 1.12** où on utilise *copy*.

  Le sous-module de *numpy* que l'on utilise principalement est *linalg*, consacré à tout ce qui est calcul matriciel.
  On l'importe avec : *import numpy.linalg as alg* et donc son alias usuel est *alg*.
  Voir la **méthode 1.11**, la **méthode 1.13** et la **méthode 1.14**
  Notons aussi *polynomial*, utile comme son nom l'indique, pour le calcul polynomial.
  On l'importe avec : *from numpy.polynomial import Polynomial*. Voir la **méthode 1.9**
- Le module **scipy** ; il permet lui aussi de faire du calcul scientifique.
  Pour le charger, on tape : *import scipy as sp*
  De plus, on utilise principalement deux de ses sous-modules.
  On les importe avec : *import scipy.optimize as resol* et *import scipy.integrate as integr*.
- Noter le choix explicite des alias. Voir : **méthode 1.7**, **méthode 1.10**, **méthode 1.19**.
  On peut rajouter le sous-module *scipy.special* qui contient notamment la fonction *binom* (**méthode 1.1** et **méthode 1.24**) et le sous-module *scipy.stats* utile en probabilités.
- Le module **matplotlib** ; il permet de faire du graphisme (tracé de courbes en particulier).
  On importe son sous-module important avec l'instruction : *import matplotlib.pyplot as plt*.
  L'alias est ici *plt*. Voir la **méthode 1.4**, la **méthode 1.5**, la **méthode 1.22**
- Le module **random** ; il permet en fait de générer des nombres aléatoires.
  On rappelle ses fonctions principales dans la **méthode 1.24**.
  On l'importe en totalité avec : *import random as rd*
- Le module **mpl_toolkits.mplot3d** ; il permet des tracés en dimension 3.
  La fonction la plus utile est *Axes*3*D*.
  On l'importe avec : *from mpl_toolkits.mplot*3*d import Axes*3*D* (voir la **méthode 1.22**).

Passons à deux modules qui ne sont pas au programme officiel mais qui sont intéressants si vous savez les manipuler. De temps en temps, dans les méthodes, essentiellement en remarque, on vous propose des exemples d'utilisation de *sympy*.

- Le module **sympy** ; il permet de faire du calcul formel, c'est-à-dire de manipuler des symboles et des expressions littérales, sans utiliser des valeurs numériques. Voir la **méthode 1.6**, la **méthode 1.7**, la **méthode 1.10**, la **méthode 1.13**, la **méthode 1.14**, la **méthode 1.19** et la **méthode 1.21**.
- Le module **time** ; ce module gère tout ce qui concerne le temps d'exécution. Donnons les commandes pour connaître le temps mis pour l'exécution d'une procédure donnée. Pour cela on utilise la fonction $perf\_counter$ du module $time$.
  $>>>\ from\ time\ import\ perf\_counter\ as\ pc$
  $>>>\ t = pc(\,);\ expression;\ print(pc(\,) - t)$
  La syntaxe « expression » est en général une valeur de la fonction dont on veut analyser la rapidité d'exécution.
  Allez voir le premier exemple de la **méthode 1.8**.

3. Les attributs

Il faut noter aussi l'existence de fonctions utilisées comme attributs. On les fait fonctionner ainsi : on affecte une variable, notons là X par exemple et on tape $X.fonction$ ce qui fait opérer $fonction$ sur X. Donnons les principaux domaines où on va les rencontrer.

- La variable affectée a est un complexe.
  Exemples : $a.real$ et $a.imag$. Voir la **méthode 1.2**.
- La variable affectée p est un polynôme.
  Exemples : $p.coef$, $p.degree$, $p.roots$, $p.deriv$. Voir la **méthode 1.9**.
- La variable affectée A est une matrice.
  Exemples : $A.shape$, $A.reshape$, $A.dot$, $A.T$. Voir la **méthode 1.11**.
- La variable affectée va est une variable aléatoire réelle.
  Exemples : $va.pmf$, $va.mean$, $va.sdt$. Voir la **méthode 1.24**.
- La variable affectée L est une liste.
  Exemples : $L.insert$, $L.append$, $L.extend$, $L.reverse$ etc.

Il y en a encore d'autres que vous allez rencontrer au fil du chapitre ou de vos futurs programmes.

■ Les commandes de base

Variables et affectation avec le symbole $=$

$>>>\ a = 3\,;\ b = 5\,;\ c, d, e\ =\ 2, 4, 6$
$>>>\ a + b$

La fonction $print$ On affiche des résultats avec $print$ en séparant par des virgules.

$>>>\ a = 3\,;\ print(2 * a, a * a, a * *10)$
6 9 59049

$>>>\ for\ i\ in\ range(2)\ :$
$\quad\quad print('trop\,facile', i\,, 'non\,?')$
donne :
$trop\ facile\ 0\ non\,?$
$trop\ facile\ 1\ non\,?$

La fonction $return$: elle agit comme $print$, c'est-à-dire affiche le résultat. Mais attention, si l'on fait appel à $print$ trois fois dans une procédure, il y aura trois affichages.
Pour $return$***, c'est le premier exécuté qui s'affiche***.
C'est important de le souligner dans des boucles conditionnelles.

La fonction *assert* : si l'on tape l'instruction *assert p*, où *p* est une proposition, l'exécution envoie un message d'erreur si *p* est fausse. Voir l'**exercice ??**.

Booléens : faux est $False$ et vrai est $True$. *A and B* signifie que l'on a *A* et *B* à la fois. *A or B* signifie que l'on a *A* ou *B*. Enfin, *not A* signifie que l'on a la négation de *A*.
Noter que $return(a == b)$ renvoie un booléen.

Fonctions de test ou pour faire une boucle : il y en a trois, if puis $while$ et enfin for. De nombreux exemples d'utilisation de ces fonctions sont proposésde la **méthode 1.1** à la **méthode 1.25**.

Opérations concernant les entiers, flottants et complexes.

Pour commencer, trois types de nombres existent principalement, les entiers tout d'abord, puis les nombres à virgule flottante appelés aussi flottants et les nombres complexes (voir la **méthode 1.2** pour ces derniers). La notation $1e-7$ par exemple désigne 10^{-7}. Il faut s'en rappeler pour des arrêts de boucles.

- ***Opérations arithmétiques somme, différence, produit et divise*** : $+ \ - \ * \ /$
- ***Exponentiation*** a^b. On écrit $a \ ** \ b$
- ***Comparaison.***
  a est inférieur (respectivement supérieur) ou égal à b s'écrit $a <= b$ (respectivement $a >= b$).
  a est égal à b s'écrit $a == b$. (Remarquer que l'égalité est == et non = qui est l'affectation.)
  a n'est pas égal à b s'écrit $a != b$.
- ***Minimum ou maximum*** (s'applique à autant d'argument que voulu) : min, max.
- ***Sommation***. La commande $k += n$ signifie que l'on rajoute n à k.
- ***Produit***. La commande $k *= n$ signifie que l'on multiplie n à k.
- ***Conversion***. La commande $float(n)$ convertit l'entier n en flottant et la commande $int(x)$ donne l'entier le plus proche de x.

Fonctions réelles prédéfinies les plus utiles.

Les fonctions prédéfinies peuvent venir parfois du module principal ou de *numpy* avec l'alias *np* ou de *math* avec l'alias *mt* ou de *scipy* avec l'alias *sp* ou enfin de *sympy*. On peut se rappeler que si par exemple, on tape $from\ numpy\ import\ *$ alors plus besoin d'alias.

- ***Nombres remarquables à connaître***. Pour la constante de Neper e, c'est $mt.e$ ou $np.e$, pour π c'est $mt.pi$ ou $np.pi$ et pour $+\infty$, c'est $np.inf$.
- ***Valeur absolue ou module*** de x. On écrit $abs(x)$.
- ***Fonction racine carrée*** de x. On écrit $np.sqrt(x)$.
- ***Partie entière et arrondi*** de x. La commande $mt.floor(x)$ donne la partie entière de x et $round(x,p)$ donne un arrondi de x avec p décimales.
- ***Fonctions trigonométriques ou hyperboliques***. On écrit $mt.sin(x)$, $mt.cos(x)$, $mt.tan(x)$ ou $np.sin(x)$, $np.cos(x)$, $np.tan(x)$. On peut rajouter les fonctions (explicites) $np.arctan(x)$, $np.arccos(x)$, $np.arcsin(x)$, $np.exp(x)$, $np.log(x)$ (qui donne $\ln x$). Ces fonctions peuvent aussi se mettre avec l'alias *mt* ou *sp*.

■ Les listes et les ensembles

Les listes. Elles sont ordonnées et commencent par le symbole [et finissent par le symbole].

- ***Syntaxe d'une liste.*** Par exemple, on tape [3, 1, 2]
- ***Liste vide***. Pour la créer, on écrit [ ]
- ***Longueur d'une liste*** L. On tape $len(L)$
- ***Premier élément de la liste*** L. On écrit $L[0]$
- ***Dernier élément de la liste*** L. On écrit $L[-1]$ ou $L[len(L)-1]$
- ***Ajout de l'élément*** x ***en position*** i dans L. On écrit $L.insert(i,\, x)$
- ***Ajout de l'élément*** x ***en fin de la liste*** L. On écrit $L.append(x)$
- ***Rajout à une liste*** L ***d'une plage formée par*** n ***éléments égaux à*** a***.***
  On tape $L \;=\; L \;+\; n * [a]$
- ***Détermination de l'index de la première occurence de*** x ***dans*** L. On écrit $L.index(x)$
- ***Renvoie le nombre d'occurences de*** x ***dans*** L. On écrit $L.count(x)$
- ***Modifie la liste*** L ***en inversant l'ordre des éléments***. On écrit $L.reverse()$
- ***Permet de récupérer les éléments de*** L ***à l'envers.*** On tape $L[:: -1]$
- ***Suppression de la première occurrence de l'élément*** x ***dans*** L. On écrit $L.remove(x)$
- ***Suppression de l'élément d'indice*** i ***dans une liste*** L. On écrit $del(L[i])$
- ***Appartenance de l'élément*** x ***à une liste*** L. On écrit $x\ in\ L$
- ***Création d'une copie*** LL ***d'une liste*** L. On écrit $LL = L[:]$
- ***Tri d'une liste*** L. On écrit $L.sort(\,)$
- ***Création d'une liste ayant le même caractère de taille donnée***.
  Par exemple $r = [0] * 25$ crée une liste r remplie de 0 et de taille 25.
- ***Somme des éléments d'une liste*** L. On tape $sum(L)$
- ***Maximum ou minimum d'une liste*** L. On tape $max(L)$ ou $min(L)$.
- ***Concaténation d'une liste*** $L1$ ***et d'une liste*** $L2$. On tape $L1 + L2$
- ***Sous-liste tirée de*** L ***de*** i ***à*** $j-1$. On écrit $L[i : j]$
- ***Sous-liste tirée de*** L ***du début à*** $j-1$. On écrit $L[: j]$ ou $L[0 : j]$
- ***Sous-liste tirée de*** L ***de*** i ***à la fin***. On écrit $L[i :]$ ou $L[i : len(L)]$
- ***Sous-liste tirée de*** L ***de*** i ***à*** $j-1$ ***en ne gardant qu'un élément sur*** k.
  On tape $L[i : j : k]$
- ***Sous-liste tirée de*** L ***du début à*** $j-1$ ***en ne gardant qu'un élément sur*** k***.***
  On tape $L[: j : k]$ ou $L[0 : j : k]$
- ***Sous-liste tirée de*** L ***de*** i ***à la fin en ne gardant qu'un élément sur*** k***.***
  On tape $L[i :: k]$ ou $L[i : len(L) : k]$

Les ensembles. Pas de répétition et pas d'ordre dans les éléments.
Les ensembles commencent par le symbole { et finissent par le symbole }.

- ***Syntaxe d'un ensemble***. On écrit par exemple {3, 1, 2}
- ***Conversion en ensemble d'une liste*** L***.*** On écrit $set(L)$
- ***Ensemble vide***. C'est { }
- ***Cardinal de l'ensemble*** Ens***.*** On écrit $len(Ens)$
- ***Ensemble donné en compréhension.*** La structure générale est $\{\, expr\ for\ i\ in\ iterable \,\}$
- ***Ajout de*** x ***dans l'ensemble*** Ens. On tape $Ens.add(x)$
- ***Suppression de*** x ***dans l'ensemble*** Ens. On écrit $Ens.discard(x)$
- ***Intersection de*** $Ens1$ ***et de*** $Ens2$. On écrit $Ens1$ & $Ens2$
- ***Réunion de*** $Ens1$ ***et de*** $Ens2$. On écrit $Ens1.union(Ens2)$

Méthodes

Autour des entiers

Méthode 1.1.— Comment manipuler des entiers avec Python

- $range(n)$ donne les entiers de 0 à $n-1$.
- $range(a,b)$ donne les entiers compris entre a (compris) et b (non compris).
- $range(a,b,p)$ donne les entiers compris entre a et b avec un pas de p.
- $range(n,i,-1)$ pour $i<n$, donne les entiers de $i+1$ à n dans l'ordre décroissant. (Voir le deuxième exemple de la **méthode 1.14**).
- $a\,//\,b$ donne le quotient dans la division euclidienne de a par b.
  Ainsi $a\,//\,n$ donne $\lfloor \frac{a}{n} \rfloor$.
- $a\,\%\,b$ donne le reste dans la division euclidienne de a par b.
  Ainsi $a\,\%\,b \;==\; 0$ signifie que a est un multiple de b.
- Pour écrire $S=\sum_{k=1}^{n} a_k$, on peut **utiliser une boucle**.
  Par exemple, pour $n=5$ et $a_k=k^2$, on tape :
  $>>>\; S \;=\; 1**2$
  $>>>\; for\ k\ in\ range(2,6)\ :$
  $\qquad S \;+=\; k**2$
  $>>>\; S$
  Ou alors on utilise la **fonction prédéfinie** *sum*. Dans ce cas, on tape :
  $>>>\; S \;=\; sum(k**2\ for\ k\ in\ range(1,6))\,;\, S$
  Rappelons au passage que si les a_k sont directement donnés sous forme d'une liste L, la commande $sum(L)$ donne S.
- $n!$ s'écrit $factorial(n)$ et préalablement on tape $from\ math\ import\ factorial$
- Pour écrire $P=\prod_{k=1}^{n} a_k$, on peut utiliser encore une boucle comparable à celle de la somme mais on remplace $S+=$ par $P*=$
- On a parfois besoin des coefficients binomiaux $\binom{n}{k}$ (qui sont des entiers) dans un exercice d'arithmétique ou en probabilité (voir la **méthode 1.25**). On peut les rentrer en usant de la fonction $factorial$ bien entendu mais on peut utiliser le sous-module *special* de *scipy*. On tape :
  $>>>\; from\ scipy.special\ import\ binom\,;\, binom(n,k)$
  On peut aussi faire une procédure récursive ou itérative.

Remarque : Ne pas confondre $a//b$ qui est le quotient de la division de a par b et est donc toujours entier et a/b qui est la division et n'est pas toujours un entier.

Exemples : 1. *Écrire une fonction nommée multiple qui prend un entier un argument n entier, qui affiche les 20 premiers multiples de cet entier en plaçant un astérisque à côté de ceux qui sont multiples de 3.*

```
>>> def multiple(n) :
        for i in range(20) :
            if i * n % 3 == 0 :
                print(i * n, " * ")
            else :
                print(i * n)
```

2. *Écrire une fonction f telle que $f(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} ij$ avec une double boucle.*

```
>>> def f(n) :
        s = 0
        for i in range(1, n + 1) :
            for j in range(i, n + 1) :
                s += i * j
        return(s)
```

3. *Écrire une fonction binaire qui calcule la liste des chiffres de l'écriture binaire d'un entier.*

Soit $n = a_p 2^p + ... + a_1 2^1 + a_0 2^0$ et alors sa décomposition binaire est la liste $[a_0, ..., a_p]$.
Tous les entiers a_i sont dans $\{0, 1\}$. On remarque que a_0 est le reste dans la division par 2 de n et que l'écriture du quotient q est $q = a_p 2^{p-1} + ... + a_2 2^1 + a_1 2^0$. Il suffit finalement de créer une liste initialement vide [], d'y ajouter le reste dans la division par 2 de n et de remplacer n par son quotient dans la division par 2, d'ajouter le reste dans sa division par 2 en queue de liste, de remplacer à nouveau n par son quotient dans la division par 2. On tape :

```
>>> def binaire(n) :
        l = []
        while n > 0 :
            l.append(n%2)
            n = n//2
        return(l)
```

■ Autour des réels, des complexes et des fonctions

❑ Méthode 1.2.— Comment travailler dans $\mathbb{C}$ avec Python

Il faut d'abord savoir que i se note $1j$ en Python. Ainsi $2 + 3i$ s'écrit $2 + 3j$.

- Pour calculer la partie réelle (imaginaire) de z, on tape $z.real$ ($z.imag$).
- Pour calculer le module du complexe z, on tape $abs(z)$.
- On peut **utiliser aussi le module** *cmath*. On tape *import cmath*
  Puis $cmath.phase(z)$ (respectivement $cmath.polar(z)$) donne l'argument (respectivement l'écriture exponentielle) de z.
  Réciproquement, la commande $cmath.rect(\rho, \theta)$ passe de la forme exponentielle $\rho e^{i\theta}$ à la forme algébrique.

Exemple : *Appliquer toutes les fonctions de la* **méthode 1.2** *à* $z = -3 + 4i$.
On tape :
$>>>\ z = -3 + 4j$
$>>>\ z.real, z.imag$
$(-3.0\,,\,4.0)$
$>>>\ import\ cmath$
$>>>\ cmath.phase(z)$
2.214297435588181
$>>>\ cmath.polar(z)$
$(5.0\,,\,2.214297435588181)$
$>>>\ cmath.rect(5.0\,,\,2.214297435588181)$
$(-2.999999999999999 + 4.000000000000001j)$

❑ **Méthode 1.3.— Comment rentrer une fonction d'une ou plusieurs variables réelles à valeurs dans $\mathbb{R}$ ou $\mathbb{R}^2$ avec Python**

- Si la fonction est ***explicitement du type*** $x \mapsto f(x)$ ***(respectivement du type*** $(x,y) \mapsto f(x,y)$,***) où l'expression, notée*** $expression$ ***de la quantité*** $f(x)$ ***(respectivement*** $f(x,y)$***) est unique***, on tape
  $>>>\ def\ f(x)$: (respectivement $def\ f(x,y)$:)
  $return\ expression$
- Si la fonction est ***du type*** $x \mapsto f(x)$, ***où*** $f(x)$ ***a plusieurs expressions selon le domaine des*** x, par exemple : $f : x \mapsto \begin{cases} f_1(x) & \text{pour} & x < a \\ f_2(x) & \text{pour} & a \leqslant x < b \\ f_3(x) & \text{pour} & b \leqslant x \end{cases}$:

```
>>> def f(x) :
        if x < a :
            return(f1(x))
        elif a <= x and x < b :
            return(f2(x))
        else :
            return(f3(x))
```

Exemple : *On considère la fonction* g *définie sur* $[0,2[$ *et* f *définie sur* $[0,+\infty[$ *par :*

$$g : x \mapsto \begin{cases} x & \text{pour} & 0 \leqslant x < 1 \\ 1 & \text{pour} & 1 \leqslant x < 2 \end{cases} \text{ et } f : x \mapsto \begin{cases} g(x) & \text{pour} & 0 \leqslant x < 2 \\ \sqrt{x}\, f(x-2) & \text{pour} & x \geqslant 2 \end{cases}.$$

Définir les deux fonctions g *et* f. *Construire la liste des valeurs* $f(k)$ *pour* $k \in [\![1,6]\!]$.
D'après Concours Arts et Metiers ParisTech-ESTP-POLYTECH, filière PSI

On tape :

```
>>> def g(x) :
        if 0 <= x and x < 1 :
            return x
        elif 1 <= x and x < 2 :
            return 1
        else :
            print( " valeur de l'antécédent erroné " )
>>> import numpy as np
>>> def f(x) :
        if 0 <= x and x < 2 :
            return g(x)
        elif x >= 2 :
            return np.sqrt(x) * f(x - 2)
        else :
            print( " valeur de l'antécédent erroné " )
>>> [f(k) for k in range(1,7)]      """ On remarque que f(x) = 0 pour x ∈ 2ℕ """
[1 , 0.0 , 1.732050807 , 0.0 , 3.87298334 , 0.0 ]
```

❑ **Méthode 1.4.— Comment tracer une fonction à valeurs dans $\mathbb{R}$ avec Python**

On rentre préalablement *numpy* (d'alias *np*) et *matplotlib.pyplot* d'alias *plt*.

- Si l'on a besoin de $+\infty$, $(-\infty)$, on tape $np.inf$ $(-np.inf)$.
- Pour ***tracer une ligne de segments brisés***, par exemple on veut relier les points $A0(x0, y0)$ à $An(xn, yn)$, on tape :

  ```
  >>> plt.axis('equal') ; plt.plot([x0, ..., xn], [y0, ..., yn])
  >>> plt.grid( ) ; plt.show( )
  ```

- Pour ***tracer le graphe de f pour $x \in [a, b]$ définie explicitement***, on définit d'abord une liste d'abscisses X avec *np.arange* en rentrant les deux extrémités a et b et le pas h ou avec *np.linspace* en rantrant a, b et le nombre de points p. Puis, on rentre la liste Y des images des éléments de X. Et on utilise *plot* et *show* du sous module *matplotlib.pyplot*. On tape

  ```
  >>> X = np.arange(a, b, h) ; Y = [ f(x) for x in X ]
  >>> plt.plot(X, Y) ; plt.show( )
  ```

  Remarque : on peut taper $Y = f(X)$ à la place de $Y = [\, f(x)\ for\ x\ in\ X\,]$.

- Pour tracer une courbe paramétrée, on définit d'abord la liste des valeurs données au paramétre puis on construit la liste des abscisses et des ordonnées correspondantes. On effectue alors le tracé de $t \mapsto (x(t), y(t))$ pour $t \in [a, b]$ avec un pas h :

  ```
  >>> T = np.arange(a, b, h) ; X = x(T) ; Y = y(T) ; plt.plot(X, Y) ; plt.show( )
  ```

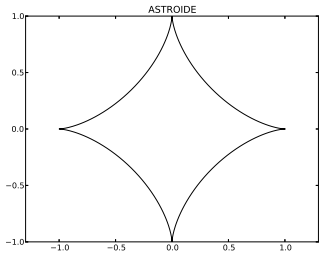
  Remarque : on peut taper aussi $T = np.linspace(a, b, p)$ (où le pas h devient le nombre p de points).

Méthodes

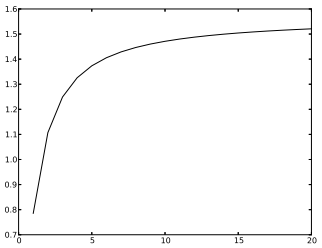
Remarque : *plt.plot* possède des options de présentation *color* et *linestyle*.
Par exemple $plt.plot(X, Y, color =' r', linestyle =' --')$ donne un tracé rouge et avec des lignes discontinues. $'g'$, $'b'$ et $'0'$ sont les codes respectivement pour la couleur verte, bleu et noire puis $'-'$ et $' :'$ sont les codes pour une ligne continue et en pointillée.
Pour tracer les axes Ox et Oy en noir (pour les distinguer de la (les) courbe(s)) : $plt.axhline(color =$ '0') ; $plt.axvline(color =$ '0'). On peut aussi incorporer des légendes ou du texte.
Ainsi : $plt.title("Courbe")$ fait afficher « Courbe » au dessus du graphe.
Puis : $plt.xlabel("Axe\ des\ abscisses")$ fait afficher « Axe des abscisses » sous l'axe des abscisses.
Enfin, soulignons l'utilité de *plt.axis* (voir un des exemples de la **méthode 1.19**).

Remarque : Si la liste des abscisses est constituée des entiers entre p et n, pour tracer $y = f(x)$, on peut utiliser la syntaxe $X = list(range(p, n+1))$ et $Y = [f(j)\ for\ j\ in\ X]$.

Exemple : *Tracer l'astroïde paramétrée par* $t \mapsto (\cos^3(t), \sin^3(t))$.
```
>>> import matplotlib.pyplot as plt ; import numpy as np
>>> T = np.linspace(0, 2*np.pi, 400)        """ On trace pour t ∈ [0, 2π] """
>>> def x(t) : return(np.cos(t)**3)
>>> def y(t) : return(np.sin(t)**3)
>>> plt.plot(x(T), y(T), color = '0') ; plt.axis('equal') ; plt.show( )
```



Exemple : *Écrire les lignes Python qui permettent de tracer la ligne de segments brisés passant par les points* $(k, f(k))$, *où* k *varie de* 1 *à* 20. *Appliquer et afficher avec* $f : x \mapsto \arctan x$.
```
>>> import matplotlib.pyplot as plt ; import numpy as np
>>> def f(x) : return(np.arctan(x))
>>> X = [k for k in range(1,21)] ; Y = [f(k) for k in range(1,21)]
>>> plt.plot(X, Y, color = '0') ; plt.show( )
```



❑ **Méthode 1.5.— Comment tracer plusieurs graphes ensemble avec Python**

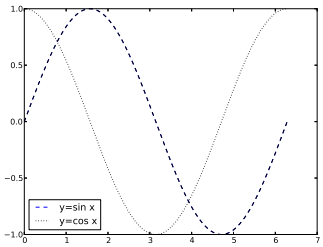
On rentre préalablement *numpy* en alias *np* et *matplotlib.pyplot* en alias *plt*.

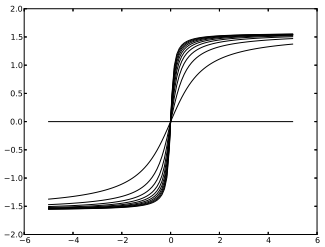
- Pour ***tracer deux graphes sur le même dessin***, $y = f(x)$ en rouge et $y = g(x)$ en vert pour $x \in [a, b]$, avec p points, on tape :
  $>>>\ X = np.linspace(a, b, p)\,;\ plt.plot(X, f(X), color =' r')$
  $>>>\ plt.plot(X, g(X), color =' g')\,;\ plt.axis('equal')\,;\ plt.show()$
  On peut rajouter $plt.legend(('y = f(x)', 'y = g(x)'), 'best')$.
- Pour ***tracer une famille de graphes sur le même dessin***, on peut utiliser une boucle *for*. Pour tracer ensemble les graphes des fonctions f_n pour $n \in [\![n1, n2]\!]$ et pour $x \in [a, b]$ avec p points, on définit une fonction des deux variables $f(x, n)$ et :
  $>>>\ X = np.linspace(a, b, p)$
  $>>>\ for\ n\ in\ range(n1, n2 + 1)\ :$
  $\qquad Y\ =\ f(n, X)\,;\ plt.plot(X, Y)\,;\ plt.show()$
- Pour ***tracer deux graphes (ou plus) côte à côte***, par exemple $y = f(x)$, $y = g(x)$ et $y = h(x)$ pour $x \in [a, b]$, avec p points, le graphe Γ_1 de f étant en rouge, celui Γ_2 de g en vert et celui Γ_3 de h en bleu par défaut, on tape :
  $>>>\ X = np.linspace(a, b, p)\,;\ plt.subplot(221)\,;\ plt.plot(X, f(X), color =' r')\,;$
  $plt.subplot(222)\,;\ plt.plot(X, g(X), color =' g')\,; plt.subplot(223)\,;\ plt.plot(X, h(X))$
  On obtient Γ_1 et Γ_2 côte à côte et Γ_3 en dessous. La commande $plt.subplot(n_1n_2n_3)$ donne le nombre n_1 de lignes, le nombre n_2 de colonnes et n_3 est le numéro de la figure.

Exemples : 1. *Tracer* sin *(en pointillé) et* cos *(en ligne discontinue) ensemble.*
$>>>\ import\ matplotlib.pyplot\ as\ plt\,;\ import\ numpy\ as\ np\,;\ X\ =\ np.linspace(0, 2{*}np.pi, 500)\,;$
$>>>\ plt.plot(X, np.sin(X), linestyle =' --')\,;\ plt.plot(X, np.cos(X), linestyle =':')$
$>>>\ plt.legend(("y = sin\,x"\,, "y = cos\,x)\,, 'best')\,;\ plt.show()$

2. *Superposer les graphes des* $f_n : t \mapsto arctan(nt)$ *entre* -5 *et* 5 *pour* $n \in [\![0, 10]\!]$.
$>>>\ import\ matplotlib.pyplot\ as\ plt\,;\ import\ numpy\ as\ np\,;\ X\ =\ np.linspace(-5, 5, 100)$
$>>>\ for\ n\ in\ range(11)$
$\qquad Y\ =\ np.arctan(n * X)\,;\ plt.plot(X, Y)\,;\ plt.show()$

On a le dessin de la question à gauche et celui de la question 2 à droite :





❑ **Méthode 1.6.— À lire en seconde lecture**

Comment étudier formellement des fonctions avec Python

Le module essentiel est ici *sympy* que l'on charge avec $from\ sympy\ import\ *$

Attention, sous *sympy*, on ne définit pas la fonction f de façon classique mais on travaille sur l'expression $f(x)$, où x a été défini avec *symbols*. Voir plus loin.

- On peut ***calculer une limite*** avec la fonction *limit*. Ainsi pour $\lim\limits_{x\to a} f(x)$:
  $>>>\ x\ =\ symbols('x')\,;\ limit(f(x),x,a)$
- On peut ***calculer une dérivée*** avec la fonction $diff$. Ainsi pour $f'(x)$ et $f''(x)$:
  $>>>\ x\ =\ symbols('x')\,;\ diff(f(x),x)\,,\ diff(f(x),x,x)$
- On peut ***calculer le développement limité*** $DL_n(a)$ avec la fonction *series* :
  $>>>\ x\ =\ symbols('x')\,;\ series(f(x),x,a,n+1)$

Exemple : *Pour* $f : x \mapsto \frac{1}{x+1} + \ln\frac{x+1}{x+2}$, *trouver* $f'(x)$, $\lim\limits_{x\to+\infty} f(x)$ *et son* $DL_4(0)$ *avec sympy.*

```
>>> from sympy import *; import numpy as np; x = symbols('x')
>>> series(1/(x+1)+log((x+1)/(x+2)),x,0,5)
1 - log(2) - x/2 + 5*x**2/8 - 17*x**3/24 + 49*x**4/64 + O(x**5)
>>> limit(1/(x+1)+log((x+1)/(x+2)),x,np.inf)
0
>>> p = diff(1/(x+1)+log((x+1)/(x+2)),x); simplify(p)
1/(-x**3 - 4*x**2 - 5*x - 2)
>>> factor(1/p)      """ On notera l'utilisation de simplify puis factor """
-(x+1)**2*(x+2)
```

❑ **Méthode 1.7.— Comment résoudre $f(x) = 0$**

- On peut utiliser $fsolve$ du sous-module *scipy.optimize*. Il faut préciser la valeur initiale de l'algorithme employé par $fsolve$. Le résultat peut dépendre de cette valeur. Il est conseillé de prendre plusieurs valeurs de x_0. La syntaxe est :
  $>>>\ import\ scipy.optimize\ as\ resol\,;\ resol.fsolve(f,x0)$
- On peut utiliser $resol.root(f,x0)$ de *scipy.optimize*.
- On peut utiliser ***l'algorithme de Dichotomie*** en le construisant (voir le deuxième exemple pour la présentation de l'algorithme et la conception de la fonction Python correspondante) ou en utilisant la fonction prédéfinie *bisect* de *scipy.optimize*. Ainsi, $resol.bisect(f,a,b)$ détermine une racine de f dans $[a,b]$.
- On peut utiliser ***l'algorithme de Newton*** en le construisant (voir le troisième exemple pour la présentation de l'algorithme et la conception de la fonction Python correspondante) ou en en utilisant la fonction prédéfinie *newton* de *scipy.optimize*. Ainsi, $resol.newton(f,x0)$ détermine une racine en partant de x_0.

Exemple : *Déterminer les réels x tels que $x^2 = 2$ de façon numérique avec la fonction fsolve du module scipy.optimize avec $x_0 = -3$ puis $x_0 = 1$ puis en utilisant la fonction newton avec $x_0 = -3$ puis avec $x_0 = 0$.*

```
>>> import scipy.optimize as resol ; def f(x) : return(x**2 - 2)
>>> resol.fsolve(f, -3) ; resol.fsolve(f, 1) ; resol.newton(f, -3) ; resol.newton(f, 0)
 array([-1.414213562]) , array([1.414213562]) , -1.414213562 , 1.414213562
```

Exemple : *Soit f une fonction continue, a et b deux réels avec $a < b$ tels que $f(a)f(b) \leqslant 0$. Le T.V.I affirme l'existence de $l \in [a, b]$ tel que $f(l) = 0$. Le* **principe de la dichotomie** *consiste à obtenir une suite $(I_n)_n$ de segments emboîtés, dont la longueur tend vers 0 et contenant chacun la limite l. En notant g_n et d_n les bornes de ces intervalles, on dispose de deux suites adjacentes, convergeant vers l. Pour obtenir cette suite de segments emboîtés, on procède comme suit :*

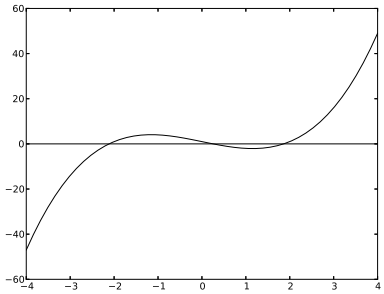
- *on pose initialement $g_0 = min(a, b)$ et $d_0 = max(a, b)$;*
- *en supposant construit le segment $I_n = [g_n, d_n]$, on pose $m = \frac{g_n + d_n}{2}$ (milieu du segment) et on calcule $f(m)$: si $f(m)$ est du signe opposé à $f(g_n)$ alors il existe une racine entre g_n et m; on pose alors $I_{n+1} = [g_n, m]$, c'est-à-dire $g_{n+1} = g_n$ et $d_{n+1} = m$. Sinon, il y a une racine entre m et d_n, donc on pose $g_{n+1} = m$ et $d_{n+1} = d_n$.*

Écrire une fonction dichotomie$(f, a, b, epsilon)$ qui renvoie une valeur approchée à epsilon près de l. On suppose que a et b vérifient les conditions plus haut.
Appliquer à $x^3 - 4x + 1 = 0$ pour trouver ses trois racines à 10^{-8} près.

```
>>> def dichotomie(f, a, b, epsilon) :
        g, d = min(a, b), max(a, b)
        while d - g > 2 * epsilon :
            m = (g + d)/2
            if f(m) == 0 :
                return m
            elif f(g) * f(m) < 0 :
                d = m
            else :
                g = m
        return (g + d)/2
```

On sort de la boucle *while* dès que $d - g <= 2 * epsilon$. Le milieu de $[g, d]$ est bien à une distance de l (qui appartient à ce segment) d'au plus *epsilon*. Puis on tape :

```
>>> import numpy as np ; import matplotlib.pyplot as plt
>>> def f(x) : return(x**3 - 4*x + 1)
>>> X = np.linspace(-4, 4, 500) ; Y = f(X) ; plt.plot(X, Y) ; plt.axhline( ) ; plt.show( )
```



Ainsi, chacun des intervalles $[-3,-1]$, $[-1,1]$ et $[1,3]$ contient une racine et une seule.
>>> $dichotomie(f,-3,-1,1e-8)$, $dichotomie(f,-1,1,1e-8)$, $dichotomie(f,1,3,1e-8)$
$(-2.114907544106245, 0.2541016899049282, 1.8608058504760265)$

Exemple : *La* **méthode de Newton** *consiste en la construction d'une suite $(x_n)_n$. Le premier terme x_0 est choisi « proche » de l, solution de $f(x)=0$ (après une étude rapide de f). La valeur x_{n+1} est l'intersection de la tangente à la courbe représentative de f au point d'abscisse x_n avec l'axe des x. Écrire une fonction $newton(f,df,a,epsilon)$ qui met en oeuvre cette méthode (df est la fonction dérivée de f) et la précision ϵ sera considérée atteinte lorsque $|f(x_n)| \leqslant \epsilon$.*
Appliquer à $x^3-4x+1=0$ pour trouver ses trois racines à 10^{-8} près.

La tangente en M d'abscisse x_n a pour équation : $y=f(x_n)+f'(x_n)(x-x_n)$.
Rapidement, on a : $x_{n+1}=x_n-\dfrac{f(x_n)}{f'(x_n)}$. Il reste à passer en Python.

```
>>> def newton(f,df,a,epsilon) :
        x = a ; n = 0 ; val = f(x)
        while abs(val) > epsilon :
            n += 1 ; x = x - val / df(x) ; val = f(x)
        print("nombre d'itérations nécessaires :" , n) ; return(x)
>>> def f(x) : return(x**3 - 4*x + 1)
>>> def df(x) : return(3*x**2 - 4)
>>> newton(f,df,-2,1e-8)        " Nombre d'itérations nécessaires : 4 "
-2.114907541476756
```

Remarque : On peut utiliser *solve* du module *sympy*. Reprenons le premier exemple :

```
>>> from sympy import * ; x = symbols('x') ; solve(x**2 - 2, x)
[-sqrt(2), sqrt(2)]
```

❑ **Méthode 1.8.— Comment étudier la suite $u_{n+1}=f(u_n)$, $u_0=a$ avec Python**

- Pour ***calculer un certain nombre de termes consécutifs***, on procède de **manière itérative** ou de **manière récursive**. Les deux pistes peuvent avoir des coûts différents. Donnons ci-dessous un exemple générique. Nous comparerons la vitesse de convergence dans les deux cas.
  Nous donnerons (pour ceux qui en veulent encore plus) une méthode intéressante d'accélération de la rapidité d'exécution avec la **méthode 1.9**.
- Pour ***tracer la ligne polygonale joignant les points*** (k,u_k) pour $0 \leqslant k \leqslant n$ en posant $u_k=f(u_{k-1})$ et $u_0=a$, on tape :

```
>>> import matplotlib.pyplot as plt
>>> def tracer(f,a,n) :
        X = [i for i in range(n+1)] ; u = a ; Y = [u]
        for i in range(n) :
            u = f(u) ; Y.append(u)
        plt.plot(X,Y) ; plt.show( )
```

Exemple : *Écrire deux procédures Python qui calcule les termes u_{38} et u_{39} de la suite définie par : $u_{n+1} = \cos(u_n)^2 - u_n$ avec $u_0 = \pi$ de façon itérative et l'autre de façon récursive, comparer leur rapidité . On utilisera perf_counter de time et on rentrera numpy avec l'alias np.*

```
>>> def terme(f,a,n) :        """ Pour la première procédure itérative """
        u = a
        for i in range(n) :
            u = f(u)
        return u
>>> from time import perf_counter as pc; def f(x) : return(np.cos(x)**2-x)
>>> for k in range(38,40) :
        print(terme(f,np.pi,k)); t = pc(); terme(f,np.pi,k); print(pc()-t)
1.6872557743882985  0.000324692591675572
-1.6737541772198121  0.00032749160575740674
>>> def u(f,a,n) :     """ Pour la deuxième procédure récursive """
        if n == 0 :
            return a
        else :
        return f(u(f,a,n-1))
>>> from time import perf_counter as pc
>>> for k in range(38,40) :
        print(u(f,np.pi,k)); t = pc(); u(f,np.pi,k); print(pc()-t)
1.6872557743882985  0.00026217993748
-1.6737541772198121  0.00025378271539
```

Exemple : *Écrire deux procédures qui calculent les termes de $u_{n+2} + au_{n+1} + bu_n = 0$ avec $u_0 = c$ et $u_1 = d$ de façon itérative puis récursive et comparer la rapidité pour $a = b = -1$, $c = 0$, $d = 1$ et $n = 38$ puis $n = 39$.*

```
>>> from time import perf_counter as pc
>>> def recurs(a,b,c,d,n) : """ De façon récursive """
        if n == 0 :
            return(c)
        elif n == 1 :
            return(d)
        else :
            return(-a*recurs(a,b,c,d,n-2)-b*recurs(a,b,c,d,n-1))
>>> def iter(a,b,c,d,n) : """ De façon itérative """
        u = c; v = d
        for i in range(n) :
            w = -b*u - a*v; u = v; v = w
        return(u)
>>>  for n in range(38,40) : """ On fait tourner le programme itératif """
        print(iter(-1,1,0,1,n)); t = pc(); iter(-1,-1,0,1,n); print(pc()-t)
39088169  0.00016777669992507072     63245986  0.00012850981266865347
>>> for n in range(38,40) : """ Avec le programme récursif, c'est bien plus long ! """
        print(recurs(-1,-1,0,1,n)); t = pc(); recurs(-1,-1,0,1,n); print(pc()-t)
39088169  48.205838361428505     63245986  76.26249289403205
```

La procédure récursive est donc parfois beaucoup plus lente. Cela vient du fait que les calculs s'organisent en pile LIFO (last in first out), ainsi le calcul de $recurs(a,b,c,d,n)$ engendre celui de

$recurs(a,b,c,d,n-1)$ et de $recurs(a,b,c,d,n-2)$, celui de $recurs(a,b,c,d,n-1)$ engendre celui de $recurs(a,b,c,d,n-2)$ et de $recurs(a,b,c,d,n-3)$, ainsi de suite jusqu'à $recurs(a,b,c,d,1)$ et $recurs(a,b,c,d,0)$. Ce dernier calcul étant effectif, il reste à « dépiler » les calculs pour revenir à $recurs(a,b,c,d,n)$.

Quand on fait une récursivité sur son écran et que l'on en demande un peu trop, on bloque l'exécution, ce qui est très énervant, surtout le jour de l'Oral de Centrale par exemple. La méthode qui suit est un moyen efficace de palier à ce problème. On utilisera ici la notion de dictionnaire qui ne fait pas parti des acquis obligatoires du programme officiel. Donc, ce sera à vous de décider si cela peut vous être utile et à condition bien entendu de le maîtriser.

❑ **Méthode 1.9.— Comment accélérer la rapidité d'exécution dans une procédure récursive (À lire en seconde lecture)**
Quand on calcule par exemple une fonction définie par récursivité nécessitant de nombreux appels redondants, il est judicieux de mémoriser les résultats obtenus en cours de calcul afin d'éviter une grande perte de temps à refaire des calculs déjà effectués. L'idée est d'enregistrer les appels effectués dans un dictionnaire. Ainsi, chaque fois qu'un appel est lancé, la fonction va d'abord chercher la réponse dans le dictionnaire. Si elle ne s'y trouve pas, alors elle effectue le calcul et met à jour le dictionnaire avec ce résultat.
Un dictionnaire est un ensemble (donc non ordonné) dans laquelle chaque valeur est accessible par une clé (au lieu d'un indice pour une liste). Ainsi, si par exemple l'on a une fonction f de trois variables n_1, n_2, n_3, trois entiers, (n_1, n_2, n_3) est la clé de $f(n_1, n_2, n_3)$.
On nomme un dictionnaire par exemple dic que l'on initialise par :
>>> $dic = \{\}$
On rentre ensuite les valeurs $f(n_1, n_2, n_3)$ dans dic.
Ainsi la commande $dic([(4,7,9)])$ fournit $f(4,7,9)$.
Le mieux pour retenir les autres syntaxes utiles d'un dictionnaire (et savoir les utiliser), c'est de regarder l'exemple générique qui va suivre.

Exemple : *Calculer les combinaisons* $\binom{n}{p}$ *par une procédure récursive par deux procédures, l'une sans l'aide d'un dictionnaire (notée* $c1$*) et l'autre avec l'aide d'un dictionnaire (notée* $c2$*). Comparer les vitesses d'exécution de* $c1(30,6)$ *et de* $c2(30,6)$*.*

```
>>> def c1(n,p) :
        if p == 0 : return 1
        if n == 0 : return 0
        return c1(n-1,p-1)+c1(n-1,p)
>>> appels = {}
>>> def c2(n,p) :
        if (n,p) in appels.keys() :
            return appels[(n,p)]
        if p == 0 : return 1
        if n == 0 : return 0
        appels[(n,p)] = c2(n-1,p-1)+c2(n-1,p)
        return appels([(n,p)]
>>> from time import perf_counter as pc
>>> t = pc(); c1(30,6); print(pc()-t)
```

1.5414269173029 14
>>> $t = pc(\,);\ c2(30,6);\ print(pc(\,) - t)$
0.0003797424655651582

❑ **Méthode 1.10.— Comment faire des opérations sur les polynômes**

On peut rentrer un polynôme comme une fonction (voir la **méthode 1.3**).
On peut aussi utiliser la fonction *Polynomial* du sous-module *numpy.polynomial* et un certain nombre d'attributs associés. On commence donc par taper :
>>> *from numpy.polynomial import Polynomial*

- Pour ***créer un polynôme***, on liste ses coefficients par ordre de degré croissant. Ainsi pour $p = a0+a1X+a2X^2+a_3X^3$, on tape : $p = Polynomial([a0, a1, a2, a3])$. L'attribut $coef$ donne accès aux coefficients ordonnés par degré croissant. Ainsi, si l'on tape : $p.coef$, on obtient $array([a0., a1., a2., a3.])$
  On suppose dans la suite de la méthode que p ***a été défini ainsi.***
- Pour ***calculer la valeur*** $p(a)$, on tape simplement : $p(a)$ et pour obtenir $[p(x1), ..., p(xn)]$, on tape : $p([x1, ..., xn])$.
- $p.coef[i]$ fournit le ***coefficient devant*** X^i.
- $p.degree(\,)$ donne le ***degré*** et $p.roots(\,)$ les ***racines*** de p.
- Pour ***dériver le polynôme*** p, on tape : $p.deriv(\,)$ qui renvoie un nouveau polynôme. Cet attribut a un argument optionnel n qui indique le nombre de dérivations à effectuer. Si l'on tape par exemple $p.deriv(2).coef$, l'on obtient les coefficients du polynôme dérivée seconde de p.
- Pour ***intégrer le polynôme*** p, on tape : $p.integ(\,)$ qui renvoie un nouveau polynôme. Cet attribut a un premier argument optionnel donnant le nombre d'intégrations à effectuer et un second argument optionnel qui donne la constante d'intégration (ou la liste de constantes si l'on intègre plusieurs fois) à utiliser.
- Pour faire des ***opérations*** sur des polynômes, on utilise $+$, $-$ et $*$ pour additionner, soustraire et multiplier des polynômes. Par exemple, l'opération $(p**3).coef$ permet d'obtenir la liste des coefficients de p^3.
- Pour faire la ***division euclidienne*** du polynôme p par le polynôme q, la commande $p//q$ donne le polynôme quotient et $p\%q$ donne le reste.

Exemple : *Soit le polynôme* $P = X^3 + 2X - 3$, *calculer successivement* $P(0)$, $[P(1), P(2), P(3)]$, deg P, *les racines de* P, *puis les polynômes* P', P'' *et enfin le quotient et le reste de la division euclidienne de* P *par* $B = X^2 + 1$.
>>> *from numpy.polynomial import Polynomial*
>>> $P = Polynomial([-3, 2, 0, 1]);\ P(0);\ P([1, 2, 3]);\ , P.degree(\,)$
$-3.0\,,\ array([0.,\ 9.,\ 30.])\,,\ 3$
>>> $P.roots(\,);\ P.deriv(\,).coef\,,\ P.deriv(2).coef$
$array([-0.5 - 1.6583124j,\ -0.5 + 1.6583124j\,,\ 1.0 + 0.j\,])\,,\ array([2.,\ 0.,\ 3.])\,,\ array([0.,\ 6.])$
>>> $B = Polynomial([1,\ 0,\ 1];\ Q = P\,//\,B;\ R = P\,\%\,B;\ Q.coef;\ R.coef$
$array([0.,\ 1.])\,,\ array([-3.,\ 1.])$

❑ **Méthode 1.11.— Comment calculer une intégrale avec Python**

- On peut calculer des ***valeurs approchées d'intégrales*** en utilisant la fonction *quad* du sous module *scipy.integrate*. Cette fonction renvoie une valeur approchée de l'intégrale ainsi qu'un majorant de l'erreur commise.
  Ainsi pour approcher $\int_a^b f(t)\,dt$, on tape :
  $>>>$ *import scipy.integrate as integr ; integr.quad*(f, a, b)
- On peut utiliser la ***méthode des rectangles*** en tapant :

```
>>> def rectangles(f,a,b,n) :
        h = (b-a)/float(n) ; z = 0
        for i in range(n) :
            z = z + f(a+i*h)
        return h*z
```

- On peut utiliser la ***méthode des trapèzes***. Les seules différences entre *trapezes*(f, a, b, n) et *rectangles*(f, a, b, n) est d'une part l'initialisation de z qui devient : $>>>$ $z = 0.5 * (f(a) - f(b))$
  et d'autre part dans la boucle *range*(n) devient *range*$(1, n)$.

Exemple : *Donner des valeurs approchées de* $I = \int_0^1 e^{-t}\,dt$.

```
>>> import math as mt ; import scipy.integrate as integr
>>> def f(t) : return mt.exp(-t)
>>> rectangles(f,0,1,20) , trapezes(f,0,1,20)
 (0.64805525909553 , 0.632252245124816)
>>> integr.quad(f,0,1)
(0.6321205588285578 , 7.017947987503856e-15)
```

Exemple : *Calculer* $I_n = \int_0^{\frac{\pi}{2}} \sin^n x\,dx$ *pour* $n = 100$ *et estimer* $\lim_{n\to+\infty} I_n$.

```
>>> import scipy.integrate as integr ; import numpy as np
>>> def I(n) :
        def f(x) : return(np.sin(x)**n)
        return(integr.quad(f,0,np.pi/2))
>>> I(100)
 (0.12501848174018731 , 1.266817667907242e-13) """ À partir de I(1e9), 0 est renvoyé """
```

Remarque : Pour le ***calcul formel***, on utilise *integrate* de *sympy*.
Reprenons le premier exemple avec le calcul de I.

```
>>> import math as mt ; from sympy import *
>>> x = symbols('x') ; integrate(exp(-x),(x,0,1))
-exp(-1)+1
>>> 1-1/(mt.e)
 0.6321205588285577
```

■ Matrices et opérations sur les matrices

❑ **Méthode 1.12.— Comment écrire puis manipuler des matrices avec Python**

On travaille surtout avec *numpy* et son sous-module *numpy.linalg* en tapant :
$>>>$ *import numpy as np*; *import numpy.linalg as alg*.
Dans la suite, $A \in \mathcal{M}_{n,p}(\mathbb{R})$, dont les lignes sont les listes $L_1, ..., L_n$, toutes de même taille p et les colonnes sont les listes $C_1, ..., C_p$ toutes de même taille n.
Attention, en Math, les indices partent de 1 ***mais en Python de*** 0**.**

- Pour ***définir la matrice*** $A \in \mathcal{M}_{n,p}(\mathbb{R})$, sous *numpy*, on tape :
  $>>>$ $A = np.array([L_1, ..., L_n])$
  Sous *sympy*, on utilise $Matrix$ à la place de $np.array$. Voir les exemples.
- $A.shape$ donne la ***taille*** (n, p) de A et $A.reshape((r, s))$ ***redimensionne une matrice*** en r lignes et s colonnes sans modifier ses coefficients (il faut que $np = rs$).
  $A.size$ renvoie $n \times p$, $np.size(A, 0)$ (*resp.* $np.size(A, 1)$) renvoie n (*resp.* p).
- Le ***coefficient*** de A à l'intersection de la ligne L_i et colonne C_j est $A[i-1, j-1]$.
- $A[i, :]$ (ou aussi $A[i]$) est la ***ligne*** L_{i+1} (tableau à une dimension) de A.
- $A[:, j]$ est la ***colonne*** C_{j+1} (tableau à une dimension) de A.
- $A[i : j, k : l]$ donne la ***sous-matrice*** de la ligne L_{i+1} à la ligne L_j et de la colonne C_{k+1} à la colonne C_l.
- $np.zeros((n, p))$ donne la ***matrice nulle*** de $\mathcal{M}_{n,p}(\mathbb{R})$ et $np.eye(n)$ donne I_n.
- $np.ones((n, p))$ donne la matrice composée que de coefficients 1 de $\mathcal{M}_{n,p}(\mathbb{R})$
- $np.diag(L)$ donne la ***matrice carrée diagonale*** dont les coefficients de la diagonale principale sont les éléments de la liste L dans l'ordre.
- Si l'on veut réunir des matrices A et B (donc les ***concatener***),
  $np.concatenate((A, B), axis = 0)$ donne $\begin{pmatrix} A \\ B \end{pmatrix}$
  $np.concatenate((A, B), axis = 1)$ donne $\begin{pmatrix} A & | & B \end{pmatrix}$.
- Pour ***ajouter*** A et B, on tape $A + B$ et pour $3A$ par exemple, on tape $3 * A$.
- Pour ***effectuer le produit matriciel*** $A \times B$, on tape $np.dot(A, B)$
  (on peut aussi utiliser *dot* comme attribut et taper $A.dot(B)$.
  Pour calculer le produit de trois matrices A, B et C, on tape $A.dot(B).dot(C)$.
  Pour calculer A^n, on peut utiliser la fonction *matrix_power* en tapant $alg.matrix\_power(A, n)$.
- Pour calculer la ***transposée*** de A, on tape $np.transpose(A)$
  (on peut aussi utiliser l'attribut T en écrivant $A.T$).
  Remarque : une application de *np.transpose* est de convertir un tableau-ligne en tableau-colonne (ou le contraire). Cela peut être utile.
- Pour calculer directement le ***déterminant*** de A (si $n = p$), on tape $alg.det(A)$.
- Pour calculer directement le ***rang*** de A, on tape $alg.matrix\_rank(A)$.
- Pour calculer directement la ***trace*** de A, on tape $np.trace(A)$.
- Pour calculer ***l'inverse*** de A (si $n = p$), on tape $alg.inv(A)$.
- Pour appliquer f à tous les coefficients de $A = (a_{i,j})$, on tape $f(A)$.

Exemple : $A = \begin{pmatrix} 1 & -1 & 0 \\ -2 & 4 & 1 \\ 0 & 3 & -5 \end{pmatrix}$, *calculer* A^T, $\mathrm{Tr}(A)$, $\mathrm{Rg}(A)$, $\mathsf{Det}\,(A)$ *et* A^7 *avec numpy.linalg.*

On tape :

$>>>$ *import numpy as np* ; *import numpy.linalg as alg.*

$>>>$ $A = np.array([[1,-1,0],[-2,4,1],[0,3,-5]])$

$>>>$ $np.transpose(A)$

$array([[1,-2,0],[-1,4,3],[0,1,-5]])$

$>>>$ $np.trace(A)$, $alg.matrix\_rank(A)$, $alg.det(A)$

0 , 3 , -13

$>>>$ $alg.matrix\_power(A,7)$

$array([[6929,-14365,676],[-28730,47996,18421],[4056,55263,-116441]])$

Exemple : *Les matrices « serpents » sont des matrices que l'on remplit d'entiers naturels en « serpent » à partir de l'entier* 1. *Voici l'exemple de celle de* $\mathcal{M}_3(\mathbb{R})$ *:* $M(3) = \begin{pmatrix} 1 & 2 & 3 \\ 6 & 5 & 4 \\ 7 & 8 & 9 \end{pmatrix}$.

Plus généralement, on note $M(n)$ *celle de* $\mathcal{M}_n(\mathbb{R})$.

1. *Définir une fonction* $coeff\_serpent$ *qui à* (n,i,j) *associe le coefficient d'indice* (i,j) *de* $M(n)$.

2. *Définir une fonction Serpent qui renvoie la matrice serpent* $M(n)$. *La tester pour* $1 \leqslant n \leqslant 5$.

3. *À l'aide de Python, afficher le rang de* $M(n)$ *pour* $1 \leqslant n \leqslant 8$. *Conjecturer.*

4. *Créer la fonction permettant d'afficher la ligne brisée formée par les points* $(k, \mathrm{Tr}(M(k)))$. *L'exécuter pour* $n = 100$ *puis* $n = 1000$.

5. *Afficher les* 100 *premières valeurs de* $\dfrac{\mathrm{Tr}(M(n))}{n^3}$. *Commenter.*

(***D'après Centrale-Supelec***).

1. Si M est la matrice serpent d'ordre n, notons $M_{i',j'}$ son coefficient à l'intersection de la $i'^{\text{ème}}$ ligne et de la $j'^{\text{ème}}$ colonne, ces indices varient entre 1 et n. On remarque que si i' est impair alors pour tout j' entier de 1 à n, $M_{i',j'} = n(i'-1)+j'$ et si i' est pair alors pour tout j' entier de 1 à n, $M_{i',j'} = i'n+1-j'$. Maintenant, dans la procédure, les indices de la matrice serpent commencent à 0. Il faut donc décaler en posant $i = i'-1$ et $j = j'-1$. Donc si i est pair, (donc i' est impair) pour tout j entier de 0 à $n-1$, $coeff\_serpent(n,i,j)$ vaut $n(i'-1)+j' = n\,i+j+1$ et de même, si i est impair, (donc i' est pair) pour tout j entier de 0 à $n-1$, $coeff\_serpent(n,i,j)$ vaut $n\,i'+1-j' = (i+1)n-j$.

Maintenant, tapons courageusement :

```
>>> import numpy as np
>>> def coeff_serpent(n,i,j) :
        if i % 2 == 0 :
            return n * i + j + 1
        else :
            return (i + 1) * n - j
```

2. Ici on va introduire une matrice A carrée d'ordre n et remplie de 0 que l'on va remplacer peu à peu par des $coeff\_serpent(n,i,j)$. De plus, on suppose *numpy* déjà importé.

```
>>> def Serpent(n) :
        A = np.zeros((n,n))
        for i in range(n) :
            for j in range(n) :
                A[i,j] = coeff_serpent(n,i,j)
        return A
>>> Serpent(5)
array([[1., 2., 3., 4., 5.], [10., 9., 8., 7., 6.], [11., 12., 13., 14., 15.]
[20., 19., 18., 17., 16.], [21., 22., 23., 24., 25.]]
```

Pour gagner de la place, on a affiché que $M(5)$ (à vous d'afficher le reste !)

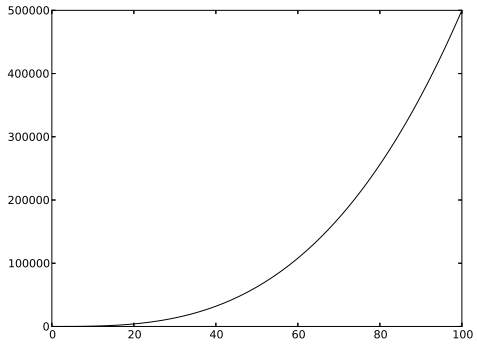
3. On tape (on affiche en ligne les résultats pour gagner de la place) :

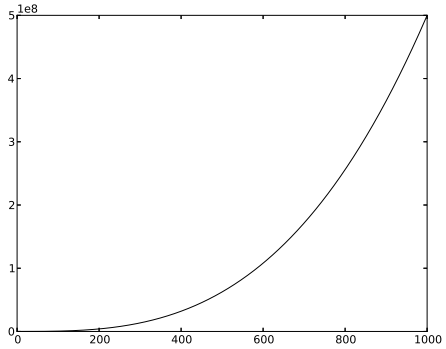
```
>>> import numpy.linalg as alg
>>> for n in range(2,9) :
        print(alg.matrix_rank(Serpent(n)))
2 2 2 2 2 2 2 2
```

Il semble que le rang de $M(n)$ soit toujours 2.

4. On applique la **méthode 1.5** ou plus exactement le deuxième exemple qui suit.
Les deux courbes apparaissent l'une à côté de l'autre par commodité.

```
>>> import matplotlib.pyplot as plt
>>> X = [k for k in range(1,101)] ; Y = [np.trace(Serpent(n)) for n in range(1,101)]
>>> plt.plot(X,Y,color = '0') ; plt.show()
```






```
>>> XX = [k for k in range(1,1001)] ; YY = [np.trace(Serpent(n)) for n in range(1,1001)]
>>> plt.plot(XX,YY,color = '0') ; plt.show()
```

5. On tape :

```
>>> [np.trace(Serpent(n))/n**3 for n in range(2,101)]
```

La liste obtenue est une alternance de 0.5 pour les valeurs n paires et des réels commençant à 0.555555558 et décroissants jusqu'à 0.500005105202 pour les valeurs n impaires.
En conclusion, on peut conjecturer que $\mathrm{Tr}(M(n)) \sim \frac{n^3}{2}$ pour n impair et $\mathrm{Tr}(M(n)) = \frac{n^3}{2}$ si n est pair.

❑ **Méthode 1.13.— Comment écrire en langage Python les opérations élémentaires classiques sur les lignes de matrices**

Il s'agit de créer les fonctions $L_i \leftarrow L_i + xL_j$, $L_i \leftarrow xL_i$ et $L_i \leftrightarrow L_j$. Aucune fonction prédéfinie n'existe. ***Attention, ici les lignes sont numérotées à partir de*** 0 ***et il vaut mieux que les coefficients de*** A ***soient des flottants.***

- Pour ***effectuer la transvection*** $L_i \leftarrow L_i + xL_j$ sur A, on tape :

```
>>> def transvecligne(A,i,j,x) :
        p = len(A[0])
        for m in range(p) :
            A[i,m] = A[i,m] + x*A[j,m]
        return A
```

  Une version alternative est :

```
>>> def transvecligne(A,i,j,x) :
        A[i] = A[i] + x*A[j]
        return A
```

- Pour ***effectuer la dilatation*** $L_i \leftarrow xL_i$ sur A, on tape :

```
>>> def dilatligne(A,i,x) :
        A[i] = x*A[i]
        return A
```

- Pour ***effectuer la permutation*** $L_i \leftrightarrow L_j$ sur A, on tape :

```
>>> def permutligne(A,i,j) :
        p = len(A[0])
        for m in range(p) :
            temp = A[i,m]; A[i,m] = A[j,m]; A[j,m] = temp
        return A
```

  Une version alternative est :

```
>>> def permutligne(A,i,j) :
        A[i], A[j] = np.copy(A[j]), np.copy(A[i])
        return A
```

Remarque : Vous avez remarqué que dans la version alternative de *permutligne*, on a utilisé la fonction *np.copy* du module *numpy* (d'alias *np*). En effet, si l'on n'utilise pas *np.copy*, on se retrouverait avec une matrice ayant deux lignes égales.

Pour ***effectuer les mêmes opérations sur les colonnes,*** on adapte sans souci. Attention, cependant, c'est $A[:, j]$ qui désigne la colonne d'indice j. Développons tout cela sous forme d'exemple.

Exemple : *Écrire les fonctions Python qui correspondent aux trois opérations sur les colonnes que l'on notera transveccolonne, dilatcolonne et permutcolonne. Appliquer alors à transformer la matrice :* $A = \begin{pmatrix} 1 & -1 & 3 \\ 0 & 1 & 2 \\ -1 & -2 & 0 \end{pmatrix}$ *en* I_3. *en utilisant un certain nombre des six fonctions Python, de la façon dont cela vous semble nécessaire.*

Pour ***effectuer la transvection*** $C_i \leftarrow C_i + kC_j$ sur A, on tape :

```
>>> def transveccolonne(A,i,j,k) :
        A[:,i] += k*A[:,j]
        return A
```

Pour ***effectuer la dilatation*** $C_i \leftarrow kC_i$ sur A, on tape :

```
>>> def dilatcolonne(A,i,x) :
        A[:,i] = x*A[:,i]
        return A
```

Pour ***effectuer la permutation*** $C_i \leftrightarrow C_j$ sur A, on tape :

```
>>> def permutcolonne(A,i,j) :
        A[:,i] , A[:,j] = np.copy(A[:,j]) , np.copy(A[:,i])
        return A
```

Application à la matrice proposée en supposant toutes les fonctions de la **méthode 1.12** tapées.

```
>>> import numpy as np
>>> A = np.array([[1.,-1.,3.],[0.,1.,2.],[-1.,-2.,0.]])
>>> transveccolonne(A,1,0,1) , transveccolonne(A,2,0,-3)    """ effectue C1 ← C1+C0 puis C2 ← C2 − 3C0 """
>>> transveccolonne(A,2,1,-2), dilatcolonne(A,2,1/9)    """ effectue C2 ← C2 − 2C1 puis C2 ← 1/9C2 """
>>> transveccolonne(A,1,2,3), transveccolonne(A,0,2,1)    """ effectue C1 ← C1+3C2 puis C0 ← C0 + C2 """
array([[1,0,0],[0,1,0],[0,0,1]])
```

❑ **Méthode 1.14.— Comment résoudre un système linéaire** $AX = B$

- On peut appliquer une ***méthode directe*** lorsque A est carrée et inversible en employant la fonction *solve* du sous-module *numpy.linalg*. On tape :

  ```
  >>> alg.solve(A,B)
  ```

  Nous obtenons X sous forme d'un tableau-ligne.
  Voir le premier exemple qui suit pour la mise en forme.
- On peut appliquer la ***méthode de Gauss*** et les fonctions construites à la **méthode 1.13**. Voir le second exemple qui suit pour le programme Python correspondant.

Mise en œuvre : exercice ??.

Exemple : *Utiliser numpy pour résoudre :* $\left\{ \begin{array}{rcl} 10^{-20}x + y & = & 1 \\ x + y & = & 2 \end{array} \right.$.

On tape :

```
>>> import numpy as np; import numpy.linalg as alg
>>> A = np.array([[10**(-20),1],[1,1]]); B = np.array([[1],[2]])
>>> alg.solve(A,B)
array([[1.],[1.]])
```

Il est clair que le résultat est « arrondi ».

Méthodes

Exemple : *Écrire une fonction systLin qui résout le système linéaire de matrice A (carrée et inversible) et de second membre B par la méthode du pivot de Gauss. Les tableaux A et B seront les arguments de systLin. La fonction ne devra modifier ni la matrice A, ni la matrice B (il faudra en faire des copies, sur lesquelles on fera les opérations). On pourra utiliser les fonctions Python correspondantes aux opérations élémentaires sur les lignes vues à la* **méthode 1.13**.

Tester la fonction systLin sur les deux systèmes $\left\{\begin{array}{rcl} 10^{-20}x+y & = & 1 \\ x+y & = & 2 \end{array}\right.$, $\left\{\begin{array}{rcl} 10^{-5}x+y & = & 1 \\ x+y & = & 2 \end{array}\right.$.

0n tape :

```
>>> def systLin(A,B) :
        n = len(A); A1 = np.copy(A); B1 = np.copy(B)
        for j in range(n-1) :
            for i in range(j+1,n) :
                c = A1[i,j]/A1[j,j]; transvecligne(A1,i,j,-c); transvecligne(B1,i,j,-c)
        for j in range(n-1,-1,-1) :
            for i in range(j-1,-1,-1) :
                c = A1[i,j]/A1[j,j]; transvecligne(A1,i,j,-c); transvecligne(B1,i,j,-c)
            dilatligne(B1,j,1./A1[j,j])
        return(B1)
>>> A = np.array([[1e-20,1.],[1.,1.]]); AA = np.array([[1e-5,1.],[1.,1.]])
>>> B = np.array([[1.],[2.]])
>>> systLin(A,B), systLin(AA,B) """ On retrouve les soucis d'arrondis """
array([[0.],[1.]]) , array([[1.00001000009070],[0.999989999899999]])
```

Remarque : On peut **utiliser la fonction** *solve_linear_system* **du module** *sympy* ***quand le système possède un paramètre***. Ainsi dans le cas d'un système à deux inconnues x et y, si m est le paramètre dont dépendent certains des coefficients du système : $\left\{\begin{array}{l} a\,x+b\,y=c \\ d\,x+e\,y=f \end{array}\right.$:

```
>>> from sympy import *
>>> m = symbols('m'); x = symbols('x'); y = symbols('y')
>>> syst = Matrix([[a,b,c],[d,e,f]]); solve_linear_system(syst,x,y)
```

On peut utiliser aussi la fonction *solve* de *sympy* en nommant les deux équations $eq1 = a*x+b*y-c$ et $eq2 = d*x+e*y-f$ puis en tapant :

```
>>> solve([eq1,eq2],[x,y])
```

Donnons un exemple de système de trois équations à trois inconnues avec un paramètre.

Utiliser sympy pour résoudre ($m \in \mathbb{R}$) *:* $\left\{\begin{array}{rcl} x+my+z & = & 1 \\ x+y+mz & = & 0 \\ 7x+(4+m)y+(3+2m)z & = & -1 \end{array}\right.$.

On tape :

```
>>> from sympy import *
>>> m = symbols('m'); x = symbols('x'); y = symbols('y'); z = symbols('z')
>>> syst = Matrix([[1,m,1,1],[1,1,m,0],[7,4+m,3+2*m,-1]])
>>> solve_linear_system(syst,x,y,z)
{z : (m-2)/(2*m*(m-1)), x : -(m+2)/(2*m) , y : (3*m-2)/(2*m*(m-1))}
```

On trouve les solutions exactes en fonction de m (mais on occulte le cas $m = 0$ ou $m = 1$).

$$x = \frac{-(m+2)}{2m},\ y = \frac{3m-2}{2m(m-1)},\ z = \frac{m-2}{2m(m-1)}.$$

❑ **Méthode 1.15.— Comment diagonaliser avec Python**

On ***utilise le sous-module*** *numpy.linalg* et on tape :

$>>>$ *import numpy.linalg as alg*

La fonction *eigvals* renvoie les valeurs propres de la matrice A sous la forme d'un tableau en ligne. On tape $alg.eigvals(A)$.
La fonction *eig* renvoie un tableau L formé par une liste $L[0]$ qui est la liste des valeurs propres et par un tableau $L[1]$ qui fournit la matrice de passage de la base canonique à la base de vecteurs propres. On tape $alg.eig(A)$.
Si l'on veut retrouver la matrice A, on tape

$>>>$ $L = alg.eig(A)$
$>>>$ $L[1].dot(np.diag(L[0])).dot(alg.inv(L[1]))$

(Voir la **méthode 1.11** pour le produit de trois matrices et l'inversion.)
Enfin, $numpy.poly(A)$ donne le polynôme caractéristique de A sous forme d'une liste des coefficients de χ_A dans l'ordre décroissant des degrés.

Exemple : *À l'aide de fonctions du sous module numpy.linalg, déterminer les valeurs propres et une matrice de vecteurs propres pour* $A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$.

$>>>$ *import numpy as np*
$>>>$ *import numpy.linalg as alg*
$>>>$ $A = np.array([[1,1,1,1],[1,1,1,1],[1,1,1,1],[1,1,1,1]])$
$>>>$ $alg.eig(A)$
$(array([0.,4.,0.,0.]),$
$array([[-8.66025404e-01, 5.00000000e-01, -2.77555756e-17, -2.77555756e-17],$
$[2.88675135e-01, 5.00000000e-01, -5.77350269e-1, -5.77350269e-1],$
$[2.88675135e-01, 5.00000000e-01, 7.88675135e-1, -2.11324865e-1]$

Remarque : On remarque que la matrice de passage n'est pas trop lisible dans l'exemple précédent. Cela provient du fait que *numpy.linalg* fournit des flottants. On peut ***utiliser le module*** *sympy* qui nous donnera un résultat proche de ce que l'on ferait à la main. Dans ce cas, une matrice ne se définit plus par *np.array* mais par $Matrix$ (voir l'exemple). Si A est une matrice ainsi définie, $A.eigenvals()$ donne la séquence des valeurs propres et $A.eigenvects()$ donne en plus des valeurs propres des vecteurs propres associés.
Astuce : on peut avoir un meilleur affichage des résultats en tapant :

$>>>$ $from\ sympy\ import\ init\_printing$
$>>>$ $init\_printing()$

Reprenons l'exemple de la matrice précédente et utilisons *sympy* pour la réduire.

On commence par rentrer le module *sympy* puis on rentre la matrice A avec la fonction $Matrix$:

$>>>$ $from\ sympy\ import\ *$
$>>>$ $A = Matrix([[1,1,1,1],[1,1,1,1],[1,1,1,1],[1,1,1,1]])$

$>>>\ A.eigenvects(\,)$
$[(0, 3, [Matrix([[-1], [1], [0], [0]]), Matrix([[-1], [0], [1], [0]]),$
$Matrix([[-1], [0], [0], [1]])], (4, 1, [Matrix([[1], [1], [1], [1]])])]$
$>>>\ init\_printing(\,)$

$$\left[\left(0, 3, \left[\begin{bmatrix} -1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \end{bmatrix}\right]\right), \left(4, 1, \left[\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}\right]\right)\right]$$

sympy est bien plus visible que *lin.alg*.
Et combiné avec *init_printing*, cela devient hyper-convivial !

❑ **Méthode 1.16.— Comment déterminer une valeur approchée (ou exacte) de la valeur propre de plus grand module à l'aide du quotient des traces de deux puissances itérées consécutives avec Python**

Cet algorithme étant proposé dans le programme officiel de 2TSI en Mathématiques, nous nous devons d'en parler. Ici $A \in \mathcal{M}_n(\mathbb{K})$ et diagonalisable dans $\mathcal{M}_n(\mathbb{K})$.

▸ ***Partie théorique***
Si $\mathsf{Sp}\,(A) = \{\lambda_1, ..., \lambda_n\}$, ces n valeurs propres n'étant pas nécessairement distinctes mais on fait l'hypothèse que l'une de ces valeurs propres, λ_n par exemple pour fixer les idées, est de module strictement plus grand que le module des autres : $\forall\, i \in [\![1, n-1]\!]$, $|\lambda_i| < |\lambda_n|$. Alors :

$$\boxed{\lim_{p \to +\infty} \frac{\mathrm{Tr}(A^{p+1})}{\mathrm{Tr}(A^p)} = \lambda_n.}$$

▸ ***Programme Python associé***
L'idée est donc de calculer informatiquement les éléments de la suite $(\mathrm{Tr}(A^p))_{p \in \mathbb{N}^\star}$ puis de faire le rapport de deux éléments consécutifs de cette suite pour p grand et on obtient une valeur approchée de la valeur propre de plus grand module de A.
Nommons $VPGM$ cette procédure d'argument la matrice A.
Si par exemple, l'on veut les n premiers termes de la suite, on tape :

$>>>\ import\ numpy\ as\ np$
$>>>\ import\ numpy.linalg\ as\ alg$
$>>>\ def\ VPGM(A, n)\ :$
$\quad for\ p\ in\ range(n)\ :$
$\quad\quad u = np.trace(alg.matrix\_power(A, p+1))$
$\quad\quad / np.trace(alg.matrix\_power(A, p))$
$\quad\quad print(u)$

ps : par manque de place, la seconde partie de l'affectation de u est sous la première partie mais on les met bien entendu au même niveau quand on tape réellement la procédure.

Exemple : *Sous les hypothèses de la* **méthode 1.16**, *justifier mathématiquement le résultat théorique.*

Comme A est diagonalisable, $\exists\, P \in \mathrm{GL}_n(\mathbb{K})$, $A = PDP^{-1}$, où $D = \mathrm{diag}(\lambda_1, ..., \lambda_n)$.
De façon classique, on montre rapidement que pour tout $k \in \mathbb{N}$, $A^k = PD^kP^{-1}$. Les matrices A^k et D^k sont semblables et elles ont la même trace, d'après une des propriétés de la trace :
$\mathrm{Tr}(A^k) = \mathrm{Tr}(D^k) = \sum_{i=1}^{n} \lambda_i^k$.

On en déduit alors : $\dfrac{\mathrm{Tr}(A^{p+1})}{\mathrm{Tr}(A^p)} = \dfrac{\sum_{i=1}^{n} \lambda_i^{p+1}}{\sum_{i=1}^{n} \lambda_i^p} = \lambda_n \dfrac{1 + \sum_{i=1}^{n-1} \left(\dfrac{\lambda_i}{\lambda_n}\right)^{p+1}}{1 + \sum_{i=1}^{n-1} \left(\dfrac{\lambda_i}{\lambda_n}\right)^{p}}$.

Par hypothèse, pour tout $i \in [\![1, n-1]\!]$, $|\lambda_i| < |\lambda_n|$, c'est-à-dire : $\left|\dfrac{\lambda_i}{\lambda_n}\right| < 1$.

Or $\lim\limits_{p\to+\infty} \left|\dfrac{\lambda_i}{\lambda_n}\right|^p = 0$ et on a bien : $\lim\limits_{p\to+\infty} \dfrac{\mathrm{Tr}(A^{p+1})}{\mathrm{Tr}(A^p)} = \lambda_n$.

Exemple : *Déterminer une valeur approchée de la valeur propre de plus grand module de la matrice* $A = \begin{pmatrix} 4 & -3 & -3 \\ 3 & -2 & -3 \\ 3 & -3 & -2 \end{pmatrix}$ *en utilisant la procédure Python* $VPGM$ *de la* **méthode 1.16** *et vérifier le résultat avec la fonction eigvals (voir la* **méthode 1.15**).
Faire de même avec $B = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$. *Que constate-t-on ?*

On tape :
$>>>$ *import numpy as np*
$>>>$ *import numpy.linalg as alg*
$>>>$ $A = np.array([[4, -3, -3], [3, -2, -3], [3, -3, -2]])$
$>>>$ $VPGM(A, 12)$
$0.0\,,\ inf\,,\ -1.0\,,\ -3.0\,,\ -1.6666666666667$
$-2.2\,,\ -1.9090909090909\,,\ -2.04761904762\,,\ -1.97674418605$
$-2.01176470588\,,\ -1.99415204678$ """ par commodité on a regroupé sur 3 lignes """
$>>>$ $alg.eigvals(A)$
$array([1., -2., 1.])$
$>>>$ $B = np.array([[1, 1, 1], [1, 1, 1], [1, 1, 1]])$
$>>>$ $VPGM(B, 4)$
1.0
3.0
3.0
3.0
On remarque déjà que A a une trace nulle. Maintenant, Python affiche inf quand il doit diviser par 0 et après poursuit tranquillement ses calculs. On a l'air de bien converger vers -2 qui est la valeur propre de plus grand module. L'intérêt de faire B aussi est de voir un exemple de convergence très rapide (la valeur propre de plus grand module est 3).

■ Géométrie euclidienne

❑ **Méthode 1.17.— Comment calculer une base orthonormée par l'algorithme de Gram-Schmidt avec Python**

Cette méthode est la mise sous Python de l'algorithme de GRAM-SCHMIDT.
On se place dans un espace euclidien E muni d'un produit scalaire $\langle\ ,\ \rangle$ et on considère un sous-espace vectoriel F de E (qui peut être E entier).
On suppose m et n deux entiers avec $m \geqslant n$.

▸ ***On commence par écrire la base de V en Python.***
• Si $V \subset \mathbb{R}^m$ et si $(u1, u2, ..., un)$ est une base de V, on charge *numpy as np* et on rentre par exemple le vecteur $u[i]$ de composantes $(a1, ..., an)$ en tapant : $u[i] = np.array(Li)$ où $Li = [a1, ..., an]$ est une liste de taille n.
On peut aussi rentrer une matrice A dont les colonnes sont $u_1, ..., u_n$. Alors $A[:, i-1]$ correspond à ui.
• Si $V \subset \mathbb{R}_m[X]$ et si $(P0, P1, ..., Pn)$ est une base de V, on charge *Polynomial* depuis *numpy.polynomial* (voir **méthode 1.10**) et on rentre par exemple le polynôme Pi en tapant $P[i] = Polynomial(Li)$ où Li est la liste des coefficients de Pi dans l'ordre croissant des degrés.

▸ ***On détecte le produit scalaire utilisé et on l'écrit en Python.***
• Si $V \subset \mathbb{R}^m$, pour calculer le produit scalaire (canonique) de deux vecteurs u et v, on tape $np.vdot(u, v)$. Voir le premier exemple.
• Si $V \subset \mathbb{R}_m[X]$, plusieurs produits scalaires existent (à partir par exemple de produits de valeurs de polynômes ou d'intégrales). On définit une fonction $phi(p, q)$ dont les arguments sont p et q deux polynômes. Voir le deuxième exemple.

▸ ***Pour « orthonormaliser », on utilise ensuite le cours.***
• Dans le cas de $V \subset \mathbb{R}^m$ et si $(u1, ...un)$ est la base de V que l'on doit normaliser, on crée une fonction *Gram_Schmidt* d'argument A matrice dont les colonnes sont $u_1, ..., un$. On commence par récupérer le nombre de lignes de A avec $m = np.size(A, 0)$ et le nombre de colonnes de A par $n = np.size(A, 1)$. Puis on définit une matrice W de taille (m, n) (remplie de 0 par exemple). Puis, on tape :

```
>>> W[:,0] = A[:,0]
>>> for k in range(1,n) :
        W[:,k] = A[:,k] - sum(np.vdot(A[:,k],W[:,i])/(np.vdot(W[:,i],W[:,i]))
        *W[:,i]  for i in range(0,k))
>>> for k in range(0,n) :
        W[:,k] = W[:,k]/np.sqrt(np.vdot(W[:,k],W[:,k]))
    return(W)
```

• Dans le cas où $V \subset \mathbb{R}_m[X]$, on remplace *np.vdot* par *phi* et A par des listes de polynômes. Voir le deuxième exemple.

Exemple : *On considère la famille* $\{\vec{u}_1(2,2,-3),\ \vec{u}_2(1,-3,-3),\ \vec{u}_3(2,0,1)\}$ *dans* $\mathbb{R}^3$.
Orthonormaliser cette famille avec la formule du cours et la procédure donnée dans la **méthode 1.17**.

```
>>> def Gram_Schmidt(A) :
        m = np.size(A,0) ; n = np.size(A,1) ; W = np.zeros((m,n))
        W[:,0] = A[:,0]
        for k in range(1,n) :
            W[:,k] = A[:,k] - sum(np.vdot(A[:,k],W[:,i])*W[:,i]/np.vdot(W[:,i],W[:,i])
            for i in range(0,k))
        for k in range(0,n) :
            W[:,k] = W[:,k]/np.sqrt(np.vdot(W[:,k],W[:,k]))
        return(W)
>>> W = Gram_Schmidt(np.array([[2,1,2],[2,-3,0],[-3,-3,1]]))
>>> W
array([[0.48507125 , 0.09834798 , 0.86892667] ,
[0.48507125 , −0.8570324 , −0.17378533] ,
[−0.72760688 , −0.50578961 , 0.46342756]])
>>> np.vdot(W[:,1],W[:,2])
−2.7755575615628914e − 17    """ cela fait 0 avec des erreurs d'arrondis """
>>> 2 * np.sqrt(17.)/17
0.48507125007266594        """ Cela est cohérent avec plus haut """
```

Exemple : *On considère* $\mathbb{R}_7[X]$ *muni du produit scalaire :*

$$phi :\ (P,Q) \mapsto \int_{-1}^{1} P(t)Q(t)\,dt.$$

1. *Créer en Python la base canonique* $(e(0),...,e(7))$ *de* $\mathbb{R}_7[X]$.

2. *Définir en Python le produit scalaire phi sous forme d'une fonction (notée phi).*
Calculer $\phi(e(i),e(i))$ *pour tout* $i \in [\![0,7]\!]$.

3. *En utilisant le procédé de Gram-Schmidt, orthonormaliser la base canonique pour le produit scalaire* ϕ. *Donner le premier, le troisième et le septième polynôme de cette nouvelle base.*

```
>>> import numpy as np ; from numpy.polynomial import Polynomial
>>> def e(n) : """ Commençons par une procédure valable dans R_n[X] """
        L = [ 0 for i in range(0,n+1) ] ; L[n] = 1
        return(Polynomial(L))
>>> e(7).coef
array([0., 0., 0., 0., 0., 0. 0., 1. ])
>>> def phi(p,q) :        """ Passons à la question 2 """
        r = p * q ; R = r.integ( ) ; return(R(1) − R(−1))
>>> for i in range(8) :
        print(phi(e(i),e(i)))
2.0000000000000000 0.666666666666666667 0.4000000000000000002 0.2857142857142857
0.222222222222222221 0.18181818181818182 0.15384615384615385 0.1333333333333333333
>>> f = [e(0)]      """ Passons à la question 3 par l'initialisation """
```

```
>>> for k in range(1,8) :     """ on construit une base orthogonale sous forme d'une liste """
        pol = e(k) - sum(phi(f[i],e(k)) * f[i] / phi(f[i],f[i])
        for i in range(k))
        f.append(pol)
>>> g = []     """ on initialise la base orthonormale """
>>> for k in range(8) :
        g.append(f[k]/np.sqrt(phi(f[k],f[k])))
>>> g[6].coef    """ Par exemple pour le septième vecteur """
```

$array([-0.7967218\,,\ 0.\,,\ 16.73115778\,,\ 0.\,,\ -50.19347334\,,\ 0.\,,\ 36.80854711])$

Remarque : on peut **orthonormaliser avec le calcul formel et** *sympy* **pour** $V \subset \mathbb{R}^m$.
Après avoir chargé *sympy*, on rentre $ui(a1,...,an)$ par : $u[i] = Matrix((a1,...,an)).reshape(n,1)$.
Puis on tape : $>>> GramSchmidt((u1,u2,...,un), orthog = True)$

Donnons un exemple. Reprenons la famille $\{\vec{u}_1(2,2,-3),\ \vec{u}_2(1,-3,-3),\ \vec{u}_3(2,0,1)\}$.
On tape alors :

```
>>> from sympy import *; import numpy as np
>>> u1 = Matrix((2,2,-3)).reshape(3,1); u2 = Matrix((1,-3,-3)).reshape(3,1)
>>> u3 = Matrix((2,0,1)).reshape(3,1)
>>> GramSchmidt([u1,u2,u3], orthog = True)
```

$[[2*sqrt(17)/17, 2*sqrt(17)/17, -3*sqrt(17)/17],\ [7*sqrt(5066)/5066, -61*sqrt(5066)/5066, -18*sqrt(5066)/2533],\ [15*sqrt(298)/298, -3*sqrt(298)/298, 4*sqrt(298)/149]]$

❑ **Méthode 1.18.— Comment calculer la projection orthogonale d'un vecteur de E en utilisant le langage Python.** Cette méthode est basée sur la définition.
Notons x le vecteur de E qui est projeté, V le sous-espace vectoriel de dimension finie sur lequel on projette et enfin $y = p_V(x)$ cette projection.

- ***Détermination de*** $p_V(x)$ ***en utilisant une base orthonormale de*** V
  Plaçons nous dans $\mathbb{R}^m$. Si l'on note $W[:,0]$, ..., $W[:,n-1]$ la base orthonormale de V, X (respectivement Y) la liste correspondant à x (respectivement y), on tape :
  $>>> Y = sum(np.vdot(W[:,i],X) * W[:,i]\ \ for\ i\ in\ range(0,n))$
- ***Détermination de*** $p_V(x)$ ***en résolvant un système linéaire traduisant l'orthogonalité de*** $x - p_V(x)$ ***aux vecteurs d'une base de l'espace vectoriel*** V ***de dimension finie***
  Toujours dans $E = \mathbb{R}^m$ (on s'y ramène si nécessaire), on note $(u1,...,up)$ une base de V avec $p < m$. On suppose que $y = p_V(x)$ a pour composantes $(y1,...,ym)$. On détermine une équation de V sous forme de $m-p$ égalités indépendantes puis on pose $z = x - y$. On exprime $z.ui = 0$ pour tout $i \in [\![1,p]\!]$. Cela fait p nouvelles égalités. On obtient un système de m équations à m inconnues.
  On le résout en utilisant la **méthode 1.14**.

Exemple : *On pose* $F = \mathsf{Vect}\,((1,1,1,1),\ (0,2,0,3)) \subset \mathbb{R}^4$.
Déterminer la projection orthogonale de $(1,-1,2,0)$ *sur* F *:*
1. *en utilisant une base orthonormale de* F *;*
2. *en résolvant un système linéaire traduisant l'orthogonalité.*

1. On tape (on appelle $(W[:,0], W[:,1])$ la base orthonormée de $\mathsf{Vect}\,(F)$ issue de Gram-Schmidt et X le vecteur que l'on projette :

$>>>\ import\ numpy\ as\ np\,;\ X\ =\ np.array([[1,-1,2,0]])$
$>>>\ W\ =\ Gram\_Schmidt(np.array([[1,0],[1,2],[1,0],[1,3]]))$
$>>>\ Y\ =\ sum(np.vdot(W[:,i],X)*W[:,i]\ \ for\ i\ in\ range(0,2))$
$>>>\ Y$

$array([1.333333333e+00,\ 5.55111512e-17,\ 1.333333333e+00,\ -6.666666667e-01])$

Le vecteur cherché est donc $(4/3, 0, 4/3, -2/3)$.

Remarque : Que cela donne t-il avec *sympy*, juste par curiosité ?

$>>>\ from\ sympy\ import\ *\,;\ \ import\ numpy\ as\ np$
$>>>\ u1\ =\ Matrix((1,1,1,1)).reshape(4,1)\,;\ u2\ =\ Matrix((0,2,0,3)).reshape(4,1)$
$>>>\ W\ =\ GramSchmidt([u1,u2]\,,\ orthog\ =\ True)$
$>>>\ W\qquad$ """ On peut rentrer u1 aussi sous la forme $Matrix([[1,1,1,1]])$ (idem u2) """

$[[1/2],[1/2],[1/2],[1/2]],[[-5*sqrt(3)/18],[sqrt(3)/6],[-5*sqrt(3)/18],[7*sqrt(3)/18]]$

$>>>\ X\ =\ np.array([[1,-1,2,0]])$ """ On va « mixer » *sympy* et *numpy* """
$>>>\ Y\ =\ np.vdot(np.array(W[0]),X)*np.array(W[0])$
$+\,np.vdot(np.array(W[1]),X)*np.array(W[1])$
$>>>\ W$
$array([[4/3],[0],[4/3],[-2/3]],\ dtype\ =\ object)$

2. On pose $x = (1,-1,2,0)$, $y = (y1,y2,y3,y4)$ et $z = x - y = (1-y1, -1-y2, 2-y3, -y4)$.
Nous allons construire mathématiquement le système à résoudre.

Comme $y \in F$, $\exists (t,s) \in \mathbb{R}^2$, $\left\{\begin{array}{lcl} y1 &=& t \\ y2 &=& t+2s \\ y3 &=& t \\ y4 &=& t+3s \end{array}\right.$ Cela donne : $\left\{\begin{array}{rcl} y1-y3 &=& 0 \\ -y1+3y2-2y4 &=& 0 \end{array}\right.$

Puis $z.u1 = z.u2 = 0 \Rightarrow \left\{\begin{array}{rcl} y1+y2+y3+y4 &=& 2 \\ -2y2-3y4 &=& 2 \end{array}\right.$

y est la projection orthogonale si $(y1,y2,y3,y4)$ est solution du système :

$$\left\{\begin{array}{rcl} y1-y3 &=& 0 \\ -y1+3y2-2y4 &=& 0 \\ y1+y2+y3+y4 &=& 2 \\ -2y2-3y4 &=& 2 \end{array}\right.$$

Il reste à passer en Python :

$>>>\ import\ numpy\ as\ np\,;\ import\ numpy.linalg\ as\ alg$
$>>>\ A\ =\ np.array([[1,0,-1,0],[-1,3,0,-2],[1,1,1,1],[0,-2,0,-3]])$
$>>>\ B\ =\ np.array([[0],[0],[2],[2]])$
$>>>\ alg.solve(A,B)$
$array([[1.333333333e+00],[-7.40148683e-17],[1.33333333e+00],[-6.666666667e-01]])$

On retrouve que $(4/3, 0, 4/3, -2/3)$ est la projection cherchée.

Remarque : on peut aussi obtenir le résultat précédent directement lisible avec *sympy* :

$>>>\ from\ sympy\ import\ *$
$>>>\ y1\ =\ symbols('y1')\,;\ y2\ =\ symbols('y2')\,;\ y3\ =\ symbols('y3')\,;\ y4\ =\ symbols('y4')$
$>>>\ syst\ =\ Matrix([[1,0,-1,0,0],[-1,3,0,-2,0],[1,1,1,1,2],[0,-2,0,-3,2]])$
$>>>\ solve\_linear\_system(syst,y1,y2,y3,y4)$

$\{\,y3\,:\,4/3\,,\,y2\,:\,0\,,\,y1\,:\,4/3\,,\,y4\,:\,-2/3\,\}$

❑ **Méthode 1.19.— Comment décomposer une matrice A inversible et carrée d'ordre n sous la forme QR avec Python**

Encore un algorithme « conseillé » dans le programme officiel de Mathématiques.
Notons $T_n^+(\mathbb{R})$ le sous ensemble de $\mathrm{GL}_n(\mathbb{R})$ constitué des matrices triangulaires supérieures, à coefficients diagonaux > 0. Si $A \in \mathrm{GL}_n(\mathbb{R})$, on démontre qu'il existe un unique couple $(Q, R) \in \mathrm{O}_n(\mathbb{R}) \times T_n^+(\mathbb{R})$ tel que $A = QR$.

- ***Preuve mathématique.*** La preuve de l'existence et de l'unicité peut se faire en exercice. On se place dans le cadre de $\mathbb{R}^n$ euclidien, rapporté à $\mathcal{B}$ sa base canonique. Si $\mathcal{B}' = \{\vec{u}_1, ..., \vec{u}_n\}$ est la famille des vecteurs colonnes de A, et si $\mathcal{B}'' = \{\vec{w}_1, ...\vec{w}_n\}$, la famille orthonormée obtenue à partir de $\mathcal{B}'$ par le procédé de Gram-Schmidt alors Q est la matrice de passage (orthogonale) de $\mathcal{B}$ à $\mathcal{B}''$ et R est la matrice de passage de $\mathcal{B}''$ à $\mathcal{B}'$.
- ***Mise en Python de l'algorithme correspondant*** On remarque que $Q^{-1} = Q^T$. Ainsi, pour déterminer la décomposition QR de A, on détermine Q par Gram-Schmidt (car Q est la matrice W de la procédure *Gram_Schmidt* de la **méthode 1.17**) et quant à R, on utilise :
  $A = QR \Rightarrow Q^TA = Q^TQR = R$. Il suffit donc de faire le produit de la transposée de Q par A. On tape :

  $>>>$ *def Decomp_QR(A) :*
  W = Gram_Schmidt(A) ; R = np.dot(np.transpose(W), A)
  L = [W, R] ; return(L)
- Il existe des fonctions prédéfinies Python pour les moins courageux : *qr* du sous-module *numpy.linalg* et *QRdecomposition( )* du module *sympy* (voir l'exemple suivant pour l'utilisation).

Exemple : *En utilisant d'une part les fonctions prédefinies et d'autre part la procédure Decomp_QR de la* **méthode 1.19**, *trouver la décomposition QR de* $A = \begin{pmatrix} 12 & -51 & 4 \\ 6 & 167 & -68 \\ -4 & 24 & -41 \end{pmatrix}$.

On tape :
$>>>$ *import numpy as np* """ On suppose *Gram_Schmidt* et *Decomp_QR* tapées """
$>>>$ *Decomp_QR(np.array([[12, −51, 4], [6, 167, −68], [−4, 24, −41]]))*
[*array*([[0.85714286 , −0.39428571 , −0.33142857] , [0.42857143 , 0.90285714 , 0.03428571] ,
[−0.28571429 , 0.17142857 , −0.94285714]]) , *array*([[1.4000*e* + 01 , 2.1000*e* + 01 , −1.4000*e* + 01],
[−6.66133815*e*−16 , 1.75000*e*+02 , −7.000*e*+01] , [0.0000*e*+00 , 1.42108547*e*−14 , 3.5000*e*+01]])]
$>>>$ *import numpy.linalg as alg ; from sympy import ∗*
$>>>$ *alg.qr(np.array([[12, −51, 4], [6, 167, −68], [−4, 24, −41]]))*
(*array*([[−0.85714286 , 0.39428571 , 0.33142857] , [−0.42857143 , −0.90285714 , −0.03428571] ,
[0.28571429 , −0.17142857 , 0.94285714]]) , *array*([[−14., −21. , 14.], [0., −175., 70.], [0., 0., −35.]]))
$>>>$ $(Q, R) =$ *Matrix([[12, −51, 4], [6, 167, −68], [−4, 24, −41]]).QRdecomposition( )*
$>>>$ *Q*
[6/7, −69/175, −58/175], [3/7, 158/175, 6/175], [−2/7, 6/35, −33/35]
On peut remarquer que *alg.qr* donne $-Q$ et $-R$, comme $A = QR = (-Q)(-R)$, le produit reste juste mais attention, alors $-R \in T_n^-(\mathbb{R})$. Enfin, Q et R sont exacts avec le module *sympy*.

■ Équations et systèmes différentiels

❑ Méthode 1.20.— Comment résoudre de façon numérique une EDL1

On commence par écrire l'équation différentielle sous la forme $y'(t) = f(y(t), t)$ et la condition initiale $y(t_0) = y_0$, ce qui permet de généraliser à une équation non linéaire, c'est-à-dire dans le cas où $f(y(t), t)$ n'est pas de la forme $a(t)y(t) + b(t)$.

- ***Première piste. On peut utiliser la fonction** odeint **du sous-module** scipy.integrate.* Cette fonction nécessite une liste de valeurs de t, commençant en t_0 et une condition initiale y_0. La fonction *odeint* renvoie des valeurs approchées (aux points contenus dans la liste des valeurs de t) de la solution y de l'équation différentielle qui vérifie $y(t_0) = y_0$. Plus précisément, si l'on désire les solutions pour $t \in [t_0, t_n]$ avec un pas h, on tape :

  >>> import scipy.integrate as integr ; def f(y,t) : return expression
  >>> T = np.arange(t0,tn,h) ; Y = integr.odeint(f,y0,T)

  Y renvoyé est un tableau dont chaque ligne correspond à la valeur d'une fonction à un instant donné. Ainsi $Y[0]$ renvoie y_0 et $Y[-1]$ renvoie $y(t_1)$. On tape alors :
  >>> plt.plot(T,Y) ; plt.show( )

  On a le tracé de la courbe intégrale.
- ***Deuxième piste. On peut construire une procédure Python basée sur l'algorithme d'Euler vu en première année.*** Voir le premier exemple ci-dessous où l'on rappelle la méthode et la fonction Python associée.

Exemple : *On cherche à déterminer une solution approchée de $y'(t) = f(y(t), t)$ valable sur un intervalle donné, c'est-à-dire $t \in [t_0, t_n]$. On suppose la condition initiale $y(t_0) = y_0$. Pour cela, on choisit une subdivision $(t_0, t_1, ..., t_n)$ de $[t_0, t_n]$ à pas constant $h = (t_n - t_0)/n$. Donc :*

$$\forall k \in [\![0, n-1]\!],\ t_{k+1} = t_k + h.$$

On définit alors la liste $(y_1, ..., y_n)$ telle que $\forall k \in [\![0, n-1]\!],\ y_{k+1} = hf(y_k, t_k) + y_k$.

1. *Expliquer l'algorithme dit* **méthode d'Euler**.

2. *Écrire une fonction Euler qui prend en argument f, $t0$, tn, n et $y0$. On créera dans une boucle la liste T des abscisses et celle Y des valeurs prises par la solution y aux points considérés. On renverra Y. Appliquer à $(t+1)y'(t) + y(t) = \cos t$ pour $t \in [0, 10]$ avec $y(0) = 1$ et $n = 10$.*

Écrire ensuite une fonction $Euler\_Affich$ de mêmes arguments, qui doit renvoyer l'affichage de la courbe reliant les points calculés. En notant Δt et Δy les amplitudes des valeurs manipulées, on créera une fenêtre graphique qui déborde de 10% de chaque côté du tracé.

3. *Appliquer à $(t+1)y'(t) + y(t) = \cos t$ pour $t \in [0, 10]$ avec $y(0) = 1$ et $n = 35$.*

1. La méthode d'Euler consiste à considérer que si h est petit alors $y(t_k + h)$ est proche de $y(t_k) + hy'(t_k)$, c'est-à-dire de $y(t_k) + hf(y(t_k), t_k)$. Ainsi, pour connaître $y(t_k + h) = y(t_{k+1})$, on doit partir de t_k et de $y(t_k)$ et on suppose qu'entre les points $M_k(t_k, y(t_k))$ et $M_{k+1}(t_{k+1}, y(t_{k+1}))$, la courbe représentative de l'unique solution y reste « proche » de sa tangente au point t_k.

2. On tape alors (*numpy* est importé en prévision de l'apparition de cos dans l'exemple) :
>>> import numpy as np

Méthodes

```
>>> def Euler(f,t0,tn,n,y0) :
        h = (tn-t0)/float(n); t = t0; y = y0
        T = [t0]; Y = [y0]
        for k in range(n) :
            y = y+h*f(y,t); t = t+h; T.append(t); Y.append(y)
        return Y
>>> def f(x,t) : return (np.cos(t)-x)/(t+1)
>>> Euler(f,0,10,10,1)
```

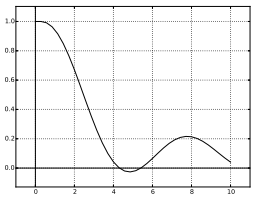
$array([1.\ ,1.\ ,0.77015115\,,\ 0.37471849\,,\ 0.03354074\,,$
$-0.10389613\,,\ -0.03930308\,,\ 0.10347883\,,\ 0.1847176\,,\ 0.1480808378,\ 0.04216238])$

Pour la seconde procédure, on garde en mémoire la plus petite et la plus grande des valeurs rencontrées pour déterminer à la fin la taille des axes (10% de plus de chaque côté, vertical et horizontal). On calcule *ymin* et *ymax* dans la boucle en même temps que le calcul de y. Idem pour les valeurs de t. Enfin, on demande ici non pas de renvoyer Y mais d'afficher le graphe de la ligne brisée constituée par T et Y. On tape alors :

```
>>> import numpy as np; import matplotlib.pyplot as plt
>>> def Euler_Affich(f,t0,tn,n,y0) :
        h = (tn-t0)/float(n); ymin = y0; ymax = y0
        t = t0; y = y0; T = [t0]; Y = [y0]
        for k in range(n) :
            y = y + h*f(y,t); t = t+h; ymin = min(ymin, y)
            ymax = max(ymax, y); T.append(t); Y.append(y)
        deltaT = (tn-t0)/10; deltaY = (ymax-ymin)/10
        plt.plot(T, Y, color ='b'); plt.grid()
        plt.axis([t0-deltaT, tn+deltaT, ymin-deltaY, ymax+deltaY])
        plt.axhline(color = '0'); plt.axvline(color = '0'); plt.show()
```

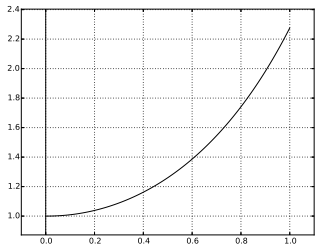
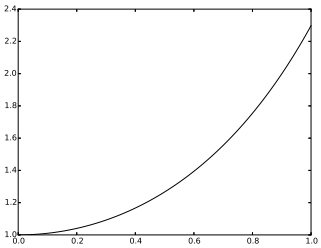
3. On écrit pour $t \in [0,1]$, $y'(t) = \dfrac{1}{t+1}\left(-y(t)+\cos t\right)$.

```
>>> Euler_Affich(f,0,10,35,1)
```



Exemple : *Déterminer sur* $[0,1]$ *la solution de* $y'(t) = t\,y(t) + t$ *qui vérifie* $y(0) = 1$ *en utilisant d'abord la procédure Euler de l'exemple précédent avec* $n = 100$ *puis odeint.*

```
>>> def f(x,t) : return(x*t+t) """ Avec Euler (n = 100) """
>>> Euler_Affich(f,0,1,100,1) (Ci-dessous la courbe de gauche.)
>>> import scipy.integrate as integr """ Avec odeint """
>>> T = np.linspace(0,1,100); Y = integr.odeint(f,1,T); plt.plot(T,Y)
```

Remarque : ***que peut-on faire ici avec*** *sympy* ***?*** On l'importe : $from\ sympy\ import\ *$
On utilise alors trois fonctions de *sympy* qui sont *dsolve*, *Function* et *Derivative*.
Donnons un exemple. On veut trouver les solutions de $y'(t) - y(t) = e^t$. On tape :

$>>>\ y\ =\ Function('y')\,;\ t\ =\ symbols('t')$
$>>>\ eq\ =\ dsolve(Derivative(y(t),t) - y(t) - exp(t), y(t))$

Si l'on tape *eq*, il s'affiche alors : $y(t)\ ==\ (C1\ +\ t) * exp(t)$
Notons la présence de la constante $C1$. Si l'on a de plus $y(0) = 1$:
$>>>\ t0\ =\ 0\,;\ y0\ =\ 1\,;\ C1\ =\ symbols('C1')$
$>>>\ eq1\ =\ (eq.rhs).subs(t,t0) - y0\,;\ S\ =\ solve([eq1],[C1])\,;\ eq.subs(S)$

$eq.subs(S)$ est une égalité dont la droite est la solution unique.
On s'inspire de la **méthode 1.14**. Notons que *eq.rhs* (respectivement *eq.lhs*) donne le membre de droite (respectivement de gauche) de l'égalité *eq*.

❑ **Méthode 1.21.— Comment résoudre de façon numérique un système différentiel linéaire d'ordre** 1 **avec Python**

Nous allons traiter le cas du système différentiel $\begin{cases} x'(t) &= a\,x(t) + b\,y(t) \\ y'(t) &= c\,x(t) + d\,y(t) \end{cases}$ avec la condition initiale $x(t_0) = x_0$ et $y(t_0) = y_0$. Ici a, b, c et d sont bien entendu constantes. On cherche une solution $t \mapsto (x(t), y(t))$ pour $t \in [t_0, t_n]$ de pas h.

- ***Première piste : on utilise*** *odeint* et on tape :

  $>>>\ import\ numpy\ as\ np\,;\ import\ scipy.integrate\ as\ integr$
  $>>>\ def\ F(x,t)\ :\ return(np.array([a*x[0] + b*x[1], c*x[0] + d*x[1]]))$
  $>>>\ T\ =\ np.arange(t0, tn, h)$
  $>>>\ X\ =\ integr.odeint(F, np.array([x0, y0]), T)$

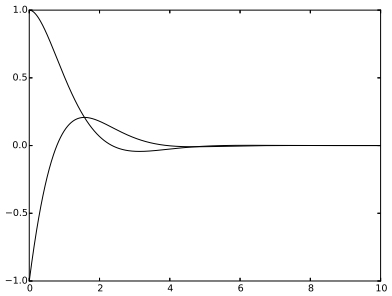
  X est un tableau de deux colonnes. Chaque $k^{\text{ème}}$ ligne correspond à $x(tk)$ et $y(tk)$.
  $X[:,0]$ (respectivement $X[:,1]$) donne $t \mapsto x(t)$ (respectivement $t \mapsto y(t)$).
  Pour afficher ces deux fonctions, on appliquera respectivement les commandes $plt.plot(T, X[:,0])$ et $plt.plot(T, X[:,1])$.
  Pour afficher $t \mapsto (x(t), y(t))$, on appliquera $plt.plot(X[:,0], X[:,1])$.

- ***Deuxième piste : on utilise la méthode d'Euler***. Voir le $2^{\text{ème}}$ exemple suivant.

Exemple : *Résoudre avec odeint le système différentiel* $\begin{cases} x'(t) &= -x(t) - y(t) \\ y'(t) &= x(t) - y(t) \end{cases}$ *sur* $[0, 10]$ *de pas* 0.01 *avec* $x(0) = 1$ *et* $y(0) = -1$. *On affichera ensemble* $t \mapsto x(t)$ *et* $t \mapsto y(t)$.

Méthodes

On tape :
>>> *import numpy as np ; import scipy.integrate as integr ; import matplotlib.pyplot as plt*
>>> *def F(x, t) : return(np.array([−x[0] − x[1], x[0] − x[1]]))*
>>> *T = np.arange(0, 10, 0.01) ; X = integr.odeint(F, np.array([1, −1]), T)*
>>> *plt.plot(T, X[:, 0]) ; plt.plot(T, X[:, 1]) ; plt.show( )*

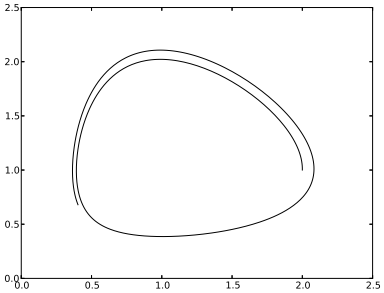


Exemple : 1. *Considérons le système différentiel* $\begin{cases} x'(t) &= f(x(t), y(t), t) \\ y'(t) &= g(x(t), y(t), t) \end{cases}$ *, où f et g sont des fonctions de $\mathbb{R}^3$ dans $\mathbb{R}$. Les conditions initiales sont $x(t0) = x0$ et $y(t0) = y0$.*
Ainsi en partant du cas du système linéaire de l'introduction de la **méthode 1.21** :
$f(x(t), y(t), t) = ax(t) + by(t)$ et $g(x(t), y(t), t) = cx(t) + dy(t)$.
Adapter $Euler\_Affich$ en $EulerSyst\_Affich$ qui a pour arguments f, g, $t0$, tn, n et $x0$, $y0$ et qui affiche la courbe intégrale $t \mapsto (x(t), y(t))$ pour $t \in [t0, tn]$ avec un pas $h = (tn - t0)/n$.

2. *Appliquer à* $\begin{cases} x'(t) &= x(t)(1 - y(t)) \\ y'(t) &= y(t)(x(t) - 1) \end{cases}$ *avec $[t0, tn] = [0, 10]$, $x(0) = 2$, $y(0) = 1$ et $n = 500$.*

1. >>> *import matplotlib.pyplot as plt*
>>> *def EulerSyst_Affich(f, g, t0, tn, n, x0, y0) :*
    t = t0 ; x = x0 ; y = y0 ; h = (tn − t0)/float(n) ; T = [t0] ; X = [x0] ; Y = [y0]
    for k in range(n) :
        x, y = x + h ∗ f(x, y, t), y + h ∗ g(x, y, t) ; t = t + h
        T.append(t) ; X.append(x) ; Y.append(y)
    plt.plot(X, Y)

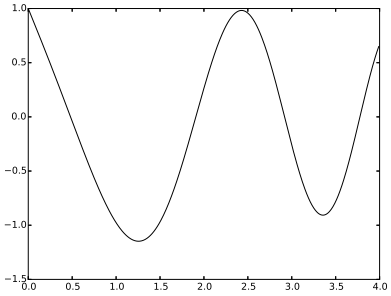
2. >>> *def f(x, y, t) : return(x ∗ (1 − y)) ; def g(x, y, t) : return(y ∗ (x − 1))*
>>> *EulerSyst_Affich(f, g, 0, 10, 500, 2, 1)*



❑ **Méthode 1.22.— Comment résoudre numériquement une EDL2**
Soit $y''(t)+a(t)y'(t)+b(t)y(t)=g(t)$ avec la condition initiale $y(t_0)=y_0$ et $y'(t_0)=y_1$.
On se ramène à un système différentiel d'ordre 1**.**
On utilise le cours avec $X(t)=\begin{pmatrix} y(t) \\ y'(t) \end{pmatrix}$, $X'(t)=A(t)X(t)+B(t)$ et en posant $A(t)=\begin{pmatrix} 0 & 1 \\ -b(t) & -a(t) \end{pmatrix}$ et $B(t)=\begin{pmatrix} 0 \\ g(t) \end{pmatrix}$. À partir de là, on peut utiliser la **méthode 1.21**.
Le plus rapide est *integr.odeint* de *scipy.integrate*, c'est-à-dire la première piste de la **méthode 1.21** avec ici :
>>> $def\ F(x,t)\ :\ return(np.array([x[1], -b(t)*x[0]-a(t)*x[1]+g(t)]))$

Remarque : Comme Python permet de travailler numériquement, il n'y a plus d'obstacle à résoudre un système différentiel du type $X'(t)=A(t)X(t)+B(t)$, où A est fonction de t. Dans le cours, où les résolutions sont exactes, c'est beaucoup plus limité. Illustrons avec l'exemple suivant.

Exemple : *Résoudre et afficher pour $t\in[0,4]$ l'unique solution de (E) $y''(t)+4ty(t)=0$, de condition initiale $y(0)=1$ et $y'(0)=-2$ en utilisant odeint du sous-module integrate du module scipy. On prendra un pas de $h=0.01$.*
Posons $X(t)=\begin{pmatrix} y(t) \\ y'(t) \end{pmatrix}$, l'équation (E) correspond au système $X'(t)=A(t)X(t)$ en posant $A(t)=\begin{pmatrix} 0 & 1 \\ -4t & 0 \end{pmatrix}$. On a : $F(X(t),t)=A(t)X(t)=\begin{pmatrix} y'(t) \\ -4ty(t) \end{pmatrix}$.
Donc $F(X(t),t)=[X[1],-4*t*X[0]]$. On tape alors :
>>> $import\ numpy\ as\ np\,;\ import\ scipy.integrate\ as\ integr\,;\ import\ matplotlib.pyplot\ as\ plt$
>>> $def\ F(X,t)\ :\ return(np.array([X[1],-4*t*X[0]]))$
>>> $T=np.arange(0,4,0.01)\,;\ X0=[1,-2]\,;\ XX=integr.odeint(F,X0,T)$
>>> $plt.plot(T,\,XX[:,0])\,;\ plt.show()$



Remarque : Python permet de résoudre directement des équations différentielles du second ordre, en ***utilisant le calcul formel et le module*** *sympy***.**
On reprend la remarque qui suit la **méthode 1.20** en la généralisant.

Exemple : *Résoudre $y''(t)+y'(t)+y(t)=e^{2t}\cos(2t)$, $y(0)=0$, $y'(0)=1$ avec sympy.*
>>> $from\ sympy\ import\ *\,;\ y=Function('y')\,;\ t=symbols('t')$
>>> $eq=dsolve(Derivative(y(t),t,t)+Derivative(y(t),t)+y(t)-exp(2*t)*cos(2*t),y(t))$

$>>>$ eq """ $C1$ et $C2$ étant les deux inévitables constantes. """
$y(t) == (C1 * sin(sqrt(3) * t/2) + C2 * cos(sqrt(3) * t/2))/sqrt(exp(t))$
$+(10 * sin(2 * t) + 3 * cos(2 * t)) * exp(2 * t)/109$
$>>>$ $t0 = 0\,;\ y0 = 0\,;\ y1 = 1$
$>>>$ $eq1 = (eq.rhs).subs(t, t0) - y0\,;\ eq2 = diff(eq.rhs, t).subs(t, t0) - y1$
$>>>$ $C1 = symbols('C1')\,;\ C2 = symbols('C2')$
$>>>$ $Sol = solve([eq1, eq2], [C1, C2])\,;\ eq.subs(Sol)$
$y(t) == (163 * sqrt(3) * sin(sqrt(3) * t/2)/327 - 3 * cos(sqrt(3) * t/2)/109)/sqrt(exp(t))$
$+(10 * sin(2 * t) + 3 * cos(2 * t)) * exp(2 * t)/109$

■ Autour des fonctions de plusieurs variables

❑ **Méthode 1.23.— Comment résoudre un système non linéaire avec Python**
On utilise le sous-module *scipy.optimize* en tapant : *import scipy.optimize as resol*
On cherche une solution $(x_0, ..., x_{n-1})$ d'un système de la forme :
$f1(x_0, ..., x_{n-1}) = 0$, $f2(x_0, ..., x_{n-1}) = 0$, ... , $fn(x_0, ..., x_{n-1}) = 0$.
On utilise *resol.fsolve* de premier argument *Syst* et de second argument une liste $[a1, ..., an]$ constituée de valeurs initiales $x_0 = a1, ..., x_{n-1} = a_n$ « pas trop loin » des bonnes solutions. On fera plusieurs choix de ces valeurs initiales.
$>>>$ $def\ Syst(X)\ :\ return(f1(X[0], ..., X[n-1]), ..., fn(X[0], ..., X[n-1]))$
$>>>$ $Sol = resol.fsolve(Syst, [a1, ..., an])\,;\ print(Sol)$
Attention, le nombre d'équations doit être le même que le nombre d'inconnues.

Exemple : *Résoudre* $\begin{cases} x^2 - y^2 &= 1 \\ x + 2y - 3 &= 0 \end{cases}$ *avec la fonction resol.fsolve.*

$>>>$ $import\ scipy.optimize\ as\ resol$
$>>>$ $def\ Syst(X)\ :\ return(X[0] * *2 - X[1] * *2 - 1\,,\ X[0] + 2 * X[1] - 3)$
$>>>$ $Sol1 = resol.fsolve(Syst, [0, 0])\,;\ Sol2 = resol.fsolve(Syst, [-3, 3])\,;\ Sol1\,,\ Sol2$
$array([1.30940108,\ 0.84529946])$ $array([-3.30940108,\ 3.15470054])$

Remarque : Une application courante de résolution de systèmes non linéaires est la recherche de points critiques. On doit commencer par trouver ce système avant de le résoudre. On peut déterminer les dérivées partielles à la main. On peut aussi ***utiliser*** *solve* ***du module*** *sympy* pour trouver l'expression littérale des dérivées d'une fonction de plusieurs variables. On rappelle que si f est une expression en x et y, $diff(f, x)$ donne $\dfrac{\partial f}{\partial x}$ sous forme d'une expression. *sympy* permet même de trouver de façon exacte les points critiques. Donnons un exemple qui utilise *sympy* dans un premier temps et où dans un second temps, on utilise la méthode numérique et la fonction *resol.fsolve* (qui est bien dans l'esprit du programme), ce qui permet de comparer les deux.

Exemple : *Trouver les points critiques de* $f : (x, y) \mapsto \dfrac{x}{x+y} + \dfrac{50 - x}{100 - (x + y)}$,
sur le fermé borné $D = [1, 49] \times [1, 49]$. *On pourra utilisera scipy.optimize et sympy, on comparera aussi avec une résolution sans Python. (**D'après Centrale-Supelec**.)*

>>> $from\,sympy\,import*$; $init\_printing()$; $x = symbols('x')$; $y = symbols('y')$
>>> $f = x/(x+y) + (50-x)/(100-(x+y))$; $p = diff(f,x)$; $q = diff(f,y)$; p , q

$$\frac{y}{(x+y)^2} + \frac{y-50}{(-x-y+100)^2},\ \frac{-x}{(x+y)^2} + \frac{-x+50}{(-x-y+100)^2}$$

Les points critiques sont solutions de (1) :

$$\begin{cases} y(-x-y+100)^2 - (50-y)(x+y)^2 &= 0 \\ -x(-x-y+100)^2 + (x+y)^2(-x+50) &= 0 \end{cases}.$$

>>> $solve([p,q],[x,y])$ """Voyons déjà nos points critiques avec *sympy* """
$[(25,25)]$ """ Ainsi, $(25,25)$ semble être le seul point critique à l'intérieur de D """
>>> $import\ scipy.optimize\ as\ resol$ """ On tente numériquement """
>>> $def\ Syst(X)$:
$return(X[1]*(-X[0]-X[1]+100)**2-(50-X[1])*(X[0]+X[1])**2\,,\ -X[0]*(-X[0]-X[1]+100)**2+(50-X[0])*(X[0]+X[1])**2)$
>>> $resol.fsolve(Syst,[30,28])$, $resol.fsolve(Syst,[1,1])$, $resol.fsolve(Syst,[49,49])$
$array([25.,25.])$, $array([0.,0.])$, $array([50.,50.])$ """ Seule la première convient"""

Reprenons (1). En multipliant la première ligne par x et la seconde par y et en sommant, on aboutit à $x=-y$ ou à $x=y$. En remplaçant, le seul point critique dans $[1,49]\times[1,49]$ est bien $(25,25)$. Rajoutons que sur $\mathbb{R}_+^\star \times \mathbb{R}_+^\star$, il n'y en a que deux : $(25,25)$ et $(50,50)$.
D'où la cohérence entre les résultats donnés par le calcul et par Python.

❑ **Méthode 1.24.— Comment tracer une ligne de niveau ou une surface**

- Pour ***tracer des lignes de niveau*** $\{(x,y)\in[a,b]\times[c,d],\ f(x,y)=k\}$, où k est un réel (qui peut prendre plusieurs valeurs), on fait une grille en X et en Y de pas h sur laquelle on calcule les valeurs de f. On emploie ensuite *contour* du module *matplotlib.pyplot* en mettant dans une liste L les valeurs de k pour lesquelles on désire le tracé. On tape :

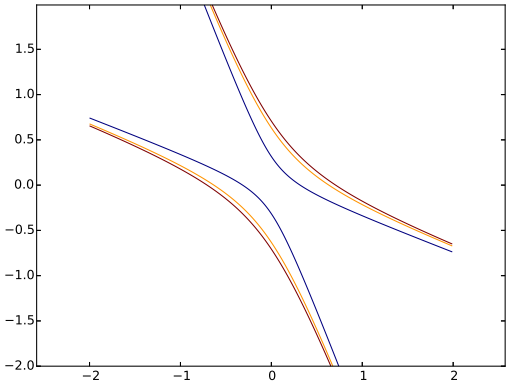
  >>> $import\ numpy\ as\ np$; $import\ matplotlib.pyplot\ as\ plt$
  >>> $X = np.arange(a,b,h)$; $Y = np.arange(c,d,h)$
  >>> $X,Y = np.meshgrid(X,Y)$
  >>> $Z = f(X,Y)$; $plt.axis('equal')$; $plt.contour(X,Y,Z,L)$; $plt.show()$

- Pour ***tracer une surface***, d'équation $z=f(x,y)$, on réalise d'abord une grille en (x,y) en supposant $x\in[a,b]$ et $y\in[c,d]$ avec un pas h puis on calcule les valeurs de z correspondants aux points de cette grille. On fait ensuite le tracé avec la fonction $plot\_surface$ issue du module *mpl_toolkits.mplot3d*. On tape :

  >>> $import\ numpy\ as\ np$; $import\ matplotlib.pyplot\ as\ plt$
  >>> $from\ mpl\_toolkits.mplot3d$; $import\ Axes3D$
  >>> $X = np.arange(a,b,h)$; $Y = np.arange(c,d,h)$
  >>> $X,Y = np.meshgrid(X,Y)$; $Z = f(X,Y)$
  >>> $ax = Axes3D(plt.figure())$; $ax.plot\_surface(X,Y,Z)$; $plt.show()$

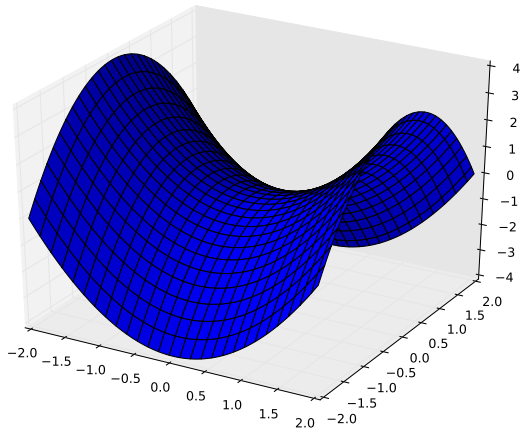
Exemple : *Tracer* $\{(x,y) \in [-2,2]^2,\ x^2+y^2+3xy=k\}$ *pour* $k \in \{0.1, 0.4, 0.5\}$. *(Pas* $h = 0.01$*).*

```
>>> import numpy as np; import matplotlib.pyplot as plt
>>> def f(x,y) : return x**2+y**2+3*x*y
>>> X = np.arange(-2,2,0.01); Y = np.arange(-2,2,0.01); X, Y = np.meshgrid(X,Y)
>>> Z = f(X,Y); plt.axis('equal'); plt.contour(X,Y,Z, [0.1,0.4,0.5]); plt.show()
```



Exemple : *Tracer la surface* $z = x^2 - y^2$ *pour* $(x,y) \in [-2,2]^2$ *avec un pas* $h = 0.015$.

```
>>> import numpy as np; import matplotlib.pyplot as plt
>>> from mpl_toolkits.mplot3d import Axes3D
>>> def f(x,y) : return x**2-y**2
>>> X = np.arange(-2,2,0.015); Y = np.arange(-2,2,0.015); X, Y = np.meshgrid(X,Y)
>>> ax = Axes3D(plt.figure()); Z = f(X,Y); ax.plot_surface(X,Y,Z); plt.show()
```



■ Probabilités et variables aléatoires réelles discrètes

❑ **Méthode 1.25.— Comment exploiter un exercice de probabilité et notamment simuler une suite de tirages avec Python**

Plusieurs modules ou sous-modules permettent soit de « bricoler » sa loi de probabilité, par exemple, si l'on veut simuler une loi discrète uniforme, soit d'utiliser des procédures qui donnent directement les lois usuelles classiques.

▸ ***1. Sous-module*** *numpy.random* :
le module *random* existe en dehors de *numpy* mais comme le module *numpy* est un des incontournables du programme officiel, on utilisera ici le sous-module *numpy.random*. On tape :

>>> import numpy.random as rd

• $rd.randint(a, b)$ permet de choisir un entier au hasard dans $[\![a, b-1]\!]$. La fonction *randint* peut prendre un troisième argument optionnel permettant d'effectuer plusieurs tirages et de renvoyer le résultat sous forme de tableau ou de matrice.
• $rd.random()$ renvoie un réel dans $[0, 1[$. Comme la fonction $rd.randint$, cette fonction accepte un argument supplémentaire optionnel permettant de réaliser plusieurs tirages et de les renvoyer sous forme de tableau ou de matrice.
• La commande $rd.binomial(n, p)$ permet de simuler une loi binomiale de paramètres n et p. Dans le cas $n = 1$, elle simule donc une loi de Bernoulli.
Un troisième argument otionnel donne le nombre de valeurs que l'on veut.
• $rd.geometric(p)$ permet de simuler une loi géométrique de paramètre p.
Un deuxième argument optionnel donne le nombre de valeurs voulues.
• La commande $rd.poisson(p)$ permet de simuler une loi de Poisson de paramètre p. Un deuxième argument optionnel donne le nombre de valeurs.

▸ ***2. Sous-module*** *stats* ***de*** *scipy* :

>>> from scipy.stats import ∗

scipy.stats possède les mêmes fonctions que *numpy.random* notées ici :

binom poisson bernoulli geom randint

On peut ajouter des attributs qui fournissent $P(X = k)$, $P(X \leq k)$ (fonction de répartition), $E(X)$ et $V(X)$. Ces attributs sont *rvs*, *pmf*, *cdf*, *mean* et *sdt*.
Ainsi, si l'on tape :

>>> va = poisson(p)

$va.rvs(size = l)$ donne une liste de tirage de Poisson de taille l
$va.pmf(k)$ donne $P(X = k)$, $va.pdf(k)$ donne $P(X \leqslant k)$
$va.mean$ donne $E(X)$ et $va.rvs$ donne $V(X)$.

Remarque : Simuler une loi de probabilité c'est souvent calculer des proportions que l'on installe dans une liste. On rappelle que $L.count(i)$ donne le nombre de fois où apparait i dans L.

Exemple : *On considère l'expérience : « on lance 4 fois une pièce et l'on compte combien de fois l'on obtient pile ». Créer une fonction experience qui recoit un entier n et renvoie, pour la réalisation de n expériences précédentes, la proportion de lancers où l'on obtient respectivement* $0, 1, 2, 3$ *ou* 4 *fois pile. Appliquer pour* $n = 5$ *et* $n = 10000$. *Comparer avec les résultats théoriques.*

On tape :

```
>>> import numpy.random as rd
>>> def experience(n) :
        L = [ ]
        for j in range(n) :
            A = rd.randint(0, 2, 4) ; S = sum(A) ; L.append(S)
        return([L.count(i)/n  for i in range(5)])
>>> experience(5) , experience(10000)
 [0.0, 0.2, 0.2, 0.4, 0.2], [0.06363, 0.25085, 0.37279, 0.25042, 0.06231]
```

Commentaire sur cet exemple :

la probabilité d'obtenir 0 fois pile est $\binom{4}{0}\left(\frac{1}{2}\right)^0\left(\frac{1}{2}\right)^4$ soit $\frac{1}{16}$, celle d'obtenir 1 fois pile est $\binom{4}{1}\left(\frac{1}{2}\right)^1\left(\frac{1}{2}\right)^3$ soit $\frac{1}{4}$, celle d'obtenir 2 fois pile est $\binom{4}{2}\left(\frac{1}{2}\right)^2\left(\frac{1}{2}\right)^2$ soit $\frac{3}{8}$, etc. On compare et on remarque que les valeurs de la liste $experience(10000)$ s'en rapprochent.

Exemple : *On veut vérifier expérimentalement la loi faible des grands nombres.*
Écrire une fonction Python LoiFaible qui reçoit un entier n et qui renvoie un tableau de taille 5 *dont chaque élément est la réalisation d'un tirage aléatoire de la variable aléatoire* $S_n = \frac{1}{n}(X_1 + ... + X_n)$, *où les* X_k *sont des v.a.r.d indépendantes de même loi* $\mathcal{B}(20, 0.7)$.
Appliquer LoiFaible à $n = 20$ *puis* $n = 1000$. *Quelle valeur théorique doit-on trouver ?*

On tape :

```
>>> from scipy.stats import binom ; var = binom(20, 0.7)
>>> def LoiFaible(n) :
        a = var.rvs(size = 5)/n
        for j in range(n - 1) :
            a+ = var.rvs(size = 5)/n
        return(a)
>>> LoiFaible(20) , LoiFaible(1000)
 array([14.15, 14.05, 14.35, 13.75, 14.7]) array([13.89, 14.051, 14.006, 14.147, 13.936])
```

On doit trouver $20 \times 0.7 = 14$. La convergence n'est pas très rapide !

Exemple : *Création d'une variable de Bernoulli de paramètre p.*
Écrire une fonction lancer(p) *qui renvoie True avec une probabilité* $p \in]0, 1[$ *et False avec une probabilité* $q = 1 - p$. *On utilisera la fonction rd.random( ). Faire quelques essais.*

On tape :

```
>>> import numpy.random as rd ; def lancer(p) : return(rd.random( ) < p)
>>> lancer(0.9) lancer(0.9999) , lancer(0.0001)
 False  True  False
```