

UNIVERSITÉ DE NANTES

BUT1 INFO

Méthodes numériques

**2023-2024**

### Exercice 1

Soit  $(u_n)$  la suite définie par  $u_0 = 1$  et pour tout entier naturel  $n$ ,

$$u_{n+1} = \frac{u_n}{2u_n + 1}.$$

On admet que pour tout  $n \in \mathbb{N}$ ,  $u_n \neq 0$  et on définit ainsi la suite  $(v_n)$  par  $v_n = \frac{1}{u_n}$ .

1. La suite  $(v_n)$  est-elle arithmétique, géométrique ?
2. En déduire une expression de  $(u_n)$  en fonction de  $n$ .
3. Montrer que pour tout entier naturel  $n$  non nul :  $0 < u_n \leq \frac{1}{3}$ .
4. Montrer que la suite  $(u_n)$  est décroissante. Que peut-on en déduire sur  $(u_n)$  ?

### Exercice 2

**2.1.** Soit  $(u_n)_{\mathbb{N}}$  la suite définie par  $u_{n+1} = \frac{1}{5}u_n + 1$  et  $u_0 = 0$

- a. D'après le théorème du point fixe trouver la seule limite  $l$  possible pour  $(u_n)_{\mathbb{N}}$ .
- b. Soit  $w_n = u_n - l$ , montrer que  $\forall n \in \mathbb{N}$ ,  $w_{n+1} = \frac{1}{5}w_n$ .
- c. En déduire une formule pour calculer  $w_n$  directement en fonction de  $n$ .
- d. Calculer  $u_n$  en fonction de  $n$  et en déduire le comportement de la suite  $(u_n)_{\mathbb{N}}$  quand  $n \rightarrow +\infty$ .

**2.2.** Soit  $(u_n)_{\mathbb{N}}$  la suite définie par  $u_{n+1} = 2u_n + 1$  et  $u_0 = 0$

- a. D'après le théorème du point fixe trouver la seule limite  $l$  possible pour  $(u_n)_{\mathbb{N}}$ .
- b. Soit  $w_n = u_n - l$ , montrer que  $(w_n)_{\mathbb{N}}$  est une suite géométrique de raison 2.
- c. Calculer  $w_n$  en fonction de  $n$ .
- d. Calculer  $u_n$  en fonction de  $n$  et en déduire le comportement de la suite  $(u_n)_{\mathbb{N}}$  quand  $n \rightarrow +\infty$ .

**2.3.** Soit  $(u_n)_{\mathbb{N}}$  la suite définie par  $u_{n+1} = \frac{2u_n + 1}{u_n + 2}$  et  $u_0 = 0$

- a. Montrer que la seule limite possible pour  $(u_n)_{\mathbb{N}}$  est  $l$  (en admettant, même si on le démontre aisément, que  $\forall n, u_n \geq 0$ ).
- b. Soit  $w_n = \frac{u_n - 1}{u_n + 1}$ , montrer que  $w_{n+1} = (1/3)w_n$ .
- c. En déduire les expressions de  $w_n$  puis  $u_n$  pour tout  $n$ .
- d. Quel est le comportement de la suite  $(u_n)_{\mathbb{N}}$  quand  $n \rightarrow +\infty$  ?

**2.4.** Soit  $(u_n)_{\mathbb{N}}$  la suite définie par  $u_{n+1} = \frac{2u_n - 1}{u_n}$  et  $u_0 = 2$

- a. Quelle est la seule limite  $l$  possible pour  $(u_n)_{\mathbb{N}}$  ?
- b. Soit  $w_n = \frac{1}{u_n - l}$ , montrer que  $(w_n)_{\mathbb{N}}$  est une suite arithmétique de raison 1.
- c. En déduire les expressions de  $w_n$  puis  $u_n$  pour tout  $n$ .
- d. Quel est le comportement de la suite  $(u_n)_{\mathbb{N}}$  quand  $n \rightarrow +\infty$  ?

### Exercice 3 Théorème de monotonie

- 3.1.** Soit  $(u_n)_\mathbb{N}$  la suite définie par  $u_{n+1} = \sqrt{1 + u_n}$  et  $u_0 = 0$ .
- Montrer que  $\forall n \in \mathbb{N}, 0 \leq u_n \leq 2$  et en déduire que  $(u_n)_\mathbb{N}$  est bien définie.
  - Montrer par récurrence que  $(u_n)_\mathbb{N}$  est croissante (i.e.  $\forall n \in \mathbb{N}, u_{n+1} - u_n \geq 0$ ).  
On pourra s'appuyer sur une astuce de calcul classique en présence de racine avec la relation  $\sqrt{a} - \sqrt{b} = \frac{(\sqrt{a} - \sqrt{b})(\sqrt{a} + \sqrt{b})}{(\sqrt{a} + \sqrt{b})} = \dots$
  - En déduire que la suite  $u_n$  est convergente et trouver sa limite  $l$ .
- 3.2.** Soit  $(u_n)_\mathbb{N}$  la suite définie par  $u_{n+1} = \frac{2u_n + 1}{u_n + 2}$  et  $u_0 = 0$
- Montrer  $\forall n \in \mathbb{N}$  on a  $0 \leq u_n \leq 1$  et en déduire que  $u_n$  est bien définie.
  - Montrer par récurrence que  $u_n$  est croissante (i.e.  $u_{n+1} - u_n \geq 0$ ).
  - En déduire que la suite  $(u_n)_\mathbb{N}$  est convergente et trouver sa limite.
- 3.3.** Soit  $(u_n)_\mathbb{N}$  la suite définie par  $u_{n+1} = \frac{5u_n - 1}{u_n + 3}$  et  $u_0 = 2$
- Montrer  $\forall n \in \mathbb{N}$  on a  $u_n > 1$ .
  - Etudier la monotonie de  $(u_n)$ .
  - Soit  $(v_n)$  la suite définie par :  $v_n = \frac{1}{u_n - 1} \quad \forall n \in \mathbb{N}$ . Montrer que  $(v_n)$  est une suite arithmétique en précisant sa raison et en déduire l'expression de  $u_n$  en fonction de  $n$ .

### Exercice 4 Vitesse de convergence

- Soit  $(u_n)_\mathbb{N}$  la suite définie par  $u_{n+1} = \sqrt{6 + u_n}$  et la donnée  $u_0 = 4$  (vérifiant  $\forall n \in \mathbb{N}, u_n \geq 3$  d'après l'exercice précédent).
  - Démontrer par récurrence que  $\forall n \in \mathbb{N}, u_n \geq 3$ .
  - Démontrer que  $\forall n \in \mathbb{N}, u_{n+1} - 3 \leq \frac{1}{6}(u_n - 3)$ . (utiliser que  $\sqrt{u_n + 6} \geq 3 +$  astuce de calcul déjà rencontrée...)
  - En déduire, par récurrence, que  $\forall n \in \mathbb{N}, 0 \leq u_n - 3 \leq \frac{1}{6^n}$ .
  - Que peut-on dire de la limite de  $(u_n)_\mathbb{N}$  quand  $n$  tend vers l'infini ? Si cette limite existe la donner et déterminer à partir de quel rang  $n$  peut-on être certain que le terme  $u_n$  est une valeur approchée de cette limite à  $10^{-8}$  près ?

$$2. \text{ Soit } (u_n)_\mathbb{N} \text{ la suite définie par récurrence } \begin{cases} u_{n+1} &= \frac{1}{2} \left( u_n + \frac{2}{u_n} \right) \\ u_0 &= 1 \end{cases}$$

### Exercice 5

- 5.1.** La suite des nombres  $\sqrt{2}, \sqrt{2\sqrt{2}}, \sqrt{2\sqrt{2\sqrt{2}}}, \dots$  converge-t-elle ? Et si oui, quelle est sa limite ?

**Exercice 6**

Étudier la suite définie par  $u_0 = 4$  et, pour tout  $n \in \mathbb{N}$ ,  $u_{n+1} = \frac{4}{u_n + 2}$ .

1. Montrer que la suite  $(u_n)$  est bien définie.
2. Étudier la monotonie de  $f$ .
3. Déterminer les points fixes de  $f \circ f$ .
4. Déterminer la nature de la suite  $(u_n)$ .

**Exercice 7**

On considère la fonction  $f$  définie sur  $\mathbb{R}$  par

$$f(x) = \frac{1}{2}x(x^2 - 3x + 4)$$

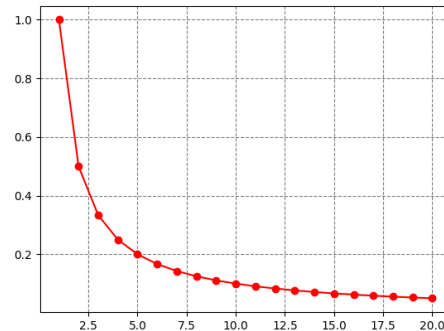
et on définit une suite  $(u_n)$  par  $u_0 \in \mathbb{R}$  et  $u_{n+1} = f(u_n)$ .

1. Étudier la monotonie de la fonction  $f$ .
2. Déterminer les points fixes de la fonction  $f$ .
3. Étudier le signe de  $f(x) - x$  sur  $\mathbb{R}$ .
4. Étudier la nature de la suite  $(u_n)$  selon la valeur de  $u_0$ .

### Exercice 8 Étude de suites

Pour afficher une suite en `python`, par exemple la suite définie par  $u_n = 1/(n+1)$ , on utilise la suite de commandes suivante (en supposant que `u1` implémente la fonction  $u1 : x \mapsto 1/(x+1)$ ) :

```
x = np.arange(1,20+1)
y = np.array(list(map(u1,x)))
print("valeur de la suite : ",y)
plt.grid(color='grey',
        linestyle = '--')
plt.plot(x,y,'ro-')
plt.show()
```



On peut conjecturer que :

- $0 \leq u_n \leq 1$
- $u_n$  est décroissante,
- si  $u_n$  est décroissante et minorée, elle est donc convergente
- elle converge vers 0.

8.1. Pour chacune des suites suivantes :

- Tracer le graphe des valeurs de  $u_n$  pour  $n = 1$  jusqu'à 20 (au moins).
- Conjecturer si la suite  $(u_n)_{\mathbb{N}}$  est bornée. Si oui donner un encadrement valable  $\forall n \in \mathbb{N}$ .
- Conjecturer si la suite  $(u_n)_{\mathbb{N}}$  est monotone. Si oui est-elle croissante ou décroissante.
- Conjecturer si la suite  $(u_n)_{\mathbb{N}}$  a une limite quand  $n \rightarrow \infty$ .

a.  $u_n = \frac{n^2 - 4}{2n^2 + 1}$

b.  $u_n = \frac{\cos(n)}{n + 1}$

c.  $u_n = \frac{1}{n + 1 - (-1)^n \sqrt{n}}$

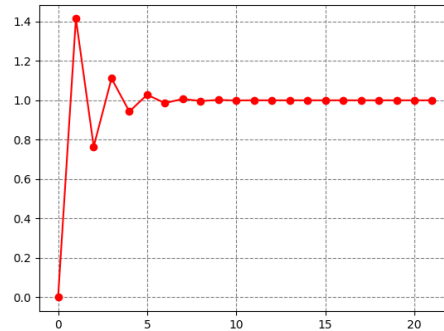
### Exercice 9 Étude de suites récurrentes

Pour afficher une suite définie par récurrence en `python`, par exemple la suite définie par :

$$\begin{cases} u_0 = 0 \\ u_{n+1} = \sqrt{2 - u_n} \end{cases}$$

on utilise la suite de commandes suivante (en supposant que `f1` implémente la fonction  $f1 : x \mapsto \sqrt{2 - x}$ ) :

```
y = [0]      # terme de départ
for i in range(20) :
    y.append(f1(y[-1]))
print("valeur de la suite : ",y)
plt.grid(color='grey',
        linestyle = '--')
plt.plot(y,'ro-')
plt.show()
```



On peut conjecturer que :

- $\forall n \in \mathbb{N}, 0 \leq u_n \leq 2$
- $(u_n)$  n'est pas monotone,
- $(u_n)$  converge vers 1

9.1. Pour chacune des suites suivantes :

- Tracer le graphe des valeurs de  $u_n$  pour  $n = 1$  jusqu'à 20 (au moins).
- Conjecturer si la suite  $(u_n)_{\mathbb{N}}$  est-elle bornée? Si oui donner un encadrement valable  $\forall n \in \mathbb{N}$ .
- Conjecturer La suite  $(u_n)_{\mathbb{N}}$  est-elle monotone? Si oui est-elle croissante ou décroissante.
- La suite  $(u_n)_{\mathbb{N}}$  a-t-elle une limite quand  $n \rightarrow \infty$ ? Justifier?

a. 
$$\begin{aligned} u_{n+1} &= 1 - (u_n - 1)^3 \\ u_0 &= 2 \end{aligned}$$

c. 
$$\begin{aligned} u_{n+1} &= 1 - (u_n - 1)^3 \\ u_0 &= 1/2 \end{aligned}$$

b. 
$$\begin{aligned} u_{n+1} &= 1 - (u_n - 1)^3 \\ u_0 &= -2 \end{aligned}$$

## Exercice 10 Suites et conjectures

10.1. Pour chaque suite ci-après :

- Tracer le graphe des valeurs de  $v_n$  pour  $n = 1$  jusqu'à 20
- Conjecturer l'expression de  $v_n$  en fonction de  $n$
- Conjecturer l'expression de  $u_n$  en fonction de  $n$
- Démontrer par récurrence le résultat obtenu pour  $u_n$ .

a.  $u_1 = 0$  et  $u_{n+1} = \frac{u_n+2}{5-2u_n}, v_n = \frac{u_n+2}{1-u_n}$

b.  $u_n = \sum_{k=1}^n k^2 = 1^2 + 2^2 + 3^2 + \dots + n^2,$   

$$v_n = \frac{\sum_{k=1}^n (k-1)k}{n \times (n+1)} = \frac{0 \times 1 + 1 \times 2 + \dots + (n-1) \times n}{n \times (n+1)}$$

### Exercice 11

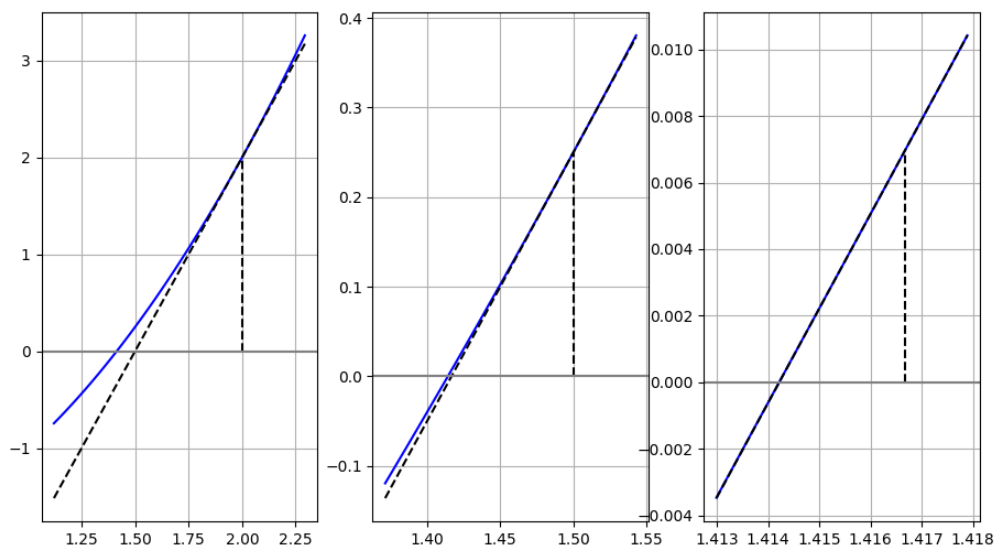
On veut pouvoir réaliser une approximation de la valeur de  $\sqrt{2}$  par une méthode de résolution approchée d'une équation du type  $f(x) = 0$ .

**11.1.** Quel choix doit être réalisé pour  $f$  ?

**11.2.** Mettre en oeuvre les méthodes de dichotomie (`dicho(a,b,f,e)`), fausse position (`fausse_pos(a,b,f,e)`) et Newton (`newton(a,f,df,e)`) en renvoyant une solution approchée  $\bar{x}$  dès lors que  $f(\bar{x}) < 10^{-8}$  et en stockant les valeurs successives des suites étudiées (initialisées en considérant que la valeur cherchée se situe entre 1 et 2).

**11.3.** Présenter un tableau comparatif montrant la rapidité de convergence des différentes méthodes et les écarts entre les suites manipulées et la valeur de  $\sqrt{2}$  renvoyée par Python.


**11.4.** Réaliser des graphiques (comme celui-ci dessous pour la méthode de Newton) pour montrer les premières étapes de chaque méthode :



## Exercice 12 Polynômes de Lagrange




Le but de cet exercice est de construire des approximations polynomiales de fonctions avec les polynômes interpolateurs de Lagrange.

- On reprend la présentation du cours afin d'implémenter des fonctions permettant la représentation de polynômes interpolateurs de Lagrange.




-  Écrire une fonction python `BaseLagrange(x,listX,i)` qui prend en entrée une valeur `x`, une liste de nombres `listX` et un entier positif `i` strictement plus petit que la longueur de `listX` et qui renvoie l'image de `x` par la fonction polynomiale

$$\begin{aligned}\mathbb{R} &\longrightarrow \mathbb{R} \\ x &\longmapsto \prod_{\substack{j=1 \\ j \neq i}}^n \frac{x-x_j}{x_i-x_j} .\end{aligned}$$

où les  $x_i$  sont les éléments de `listX`.



-  Utiliser la fonction précédente pour visualiser ces polynômes avec un choix raisonnable de `listX`, par exemple `listX = [0,1,2]`.
  -  Écrire une fonction python `InterLagrange1(x,listX,listY)` qui prend en entrée une valeur `x` et deux listes de nombres `listX` et `listY` de même taille  $n$  et qui renvoie l'image de `x` par le polynôme interpolateur de Lagrange de degré au plus  $n$  passant les points  $(x_i, y_i)$  pour  $x_i = \text{listX}[i]$  et  $y_i = \text{listY}[i]$ .
  -  Utiliser la fonction précédente pour visualiser ces polynômes avec des choix raisonnables de `listX` et `listY`, comme par exemple `listX = [0,1,2]` et `listY = [-1,3,2]`.
- Interlude : l'algorithme de Horner.* On présente ici une méthode afin d'optimiser le temps de calcul pour l'évaluation d'un polynôme. On considère un polynôme

$$P(X) = a_0 + a_1X + \cdots + a_NX^N.$$

-  Implémenter une fonction (naïve, sans utiliser d'opérateur ou de fonction de calcul de puissances intégrés à python) `EvalP(b,listCoef)` qui prend en entrée un nombre `b` et une liste de coefficients `listCoef = [a0,...,aN]` et qui renvoie  $P(b)$ .
-  Estimer le nombre d'opérations (sommes et produits) nécessaires au calcul dans la fonction précédente.
-  Constater que l'on a la factorisation suivante :

$$P(X) = a_0 + X(a_1 + X(a_2 + \cdots X(a_{N-1} + Xa_N) \cdots)).$$

Estimer le nombre d'opérations (sommes et produits) nécessaires à l'évaluation du polynôme  $P$  en  $b$  en utilisant la factorisation précédente.

-  Implémenter une fonction `Horner(b,listCoef)` qui utilise la factorisation pour évaluer le polynôme  $P$ .
-  Implémenter une fonction `compare` qui compare le temps d'exécution de `Horner` et de `EvalP`. On prendra une liste de coefficient relativement grande et on pourra itérer  $N$  fois chacune fonction avec  $N$  grand afin de mettre en lumière une différence notable.



3. *Approximation de fonctions.* Dans cette partie, on va procéder à des approximations de fonctions par des polynômes d'abord pour la fonction exponentielle sur  $[-1; 1]$  puis pour la fonction suivante :

$$f: [-5, 5] \longrightarrow \mathbb{R} \\ x \longmapsto \frac{1}{1+x^2} ,$$

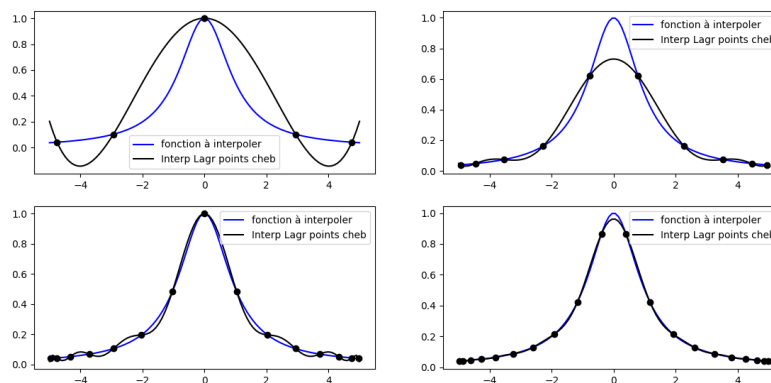
On va approcher ces fonctions avec des polynômes interpolateurs de Lagrange. Pour cela, on va choisir un nombre de points de la courbe représentative de chacune de ces fonctions et tracer le polynôme de degré minimal passant par ces points. On veut implémenter une fonction `approch(fonc,a,b,listN)` affichant le courbe représentative de la fonction `fonc` sur l'intervalle  $[a, b]$ , la courbe représentative du polynôme interpolateur défini pour un nombre `N` d'éléments appartenant à `listN` avec un graphique réalisé pour chaque élément de `listN`.

- (a) On commence par choisir des points répartis uniformément dans l'intervalle  $[a, b]$ .
- En considérant les points d'abscisses donnés par `absc= np.linspace((a,b,N))` et d'ordonnées `ordo = f(absc)`, exécuter la fonction `approch` pour la fonction exponentielle sur  $[-1; 1]$ . Vous utiliserez `[5, 10, 15, 20]` comme liste de nombre de points utilisés .
  - Faire de même avec la fonction  $f$  sur  $[-5; 5]$ . Que constatez-vous lorsque  $N$  grandit avec la fonction  $f$  ?
- (b) Le choix de prendre des points équirépartis dans l'intervalle de définition est complètement arbitraire et non optimal. En réalité, on peut choisir les abscisses des points d'interpolations afin de réduire une certaine distance entre la fonction  $f$  et le polynôme interpolateur  $P$ .
- Adaptez la fonction `approch` pour approcher la fonction  $f$  en se basant sur les abscisses des points d'interpolations suivants sur  $[-1; 1]$  à adapter pour un intervalle  $[a, b]$  :

$$x_k = \cos\left(\frac{2k-1}{2N}\pi\right), \quad k = 1, \dots, N$$

avec  $N$  le nombre de points d'interpolation. Cette liste de nombres renvoie l'interpolation dite au sens de Tchebychev.

- Représenter les interpolations obtenues comme ci-dessous

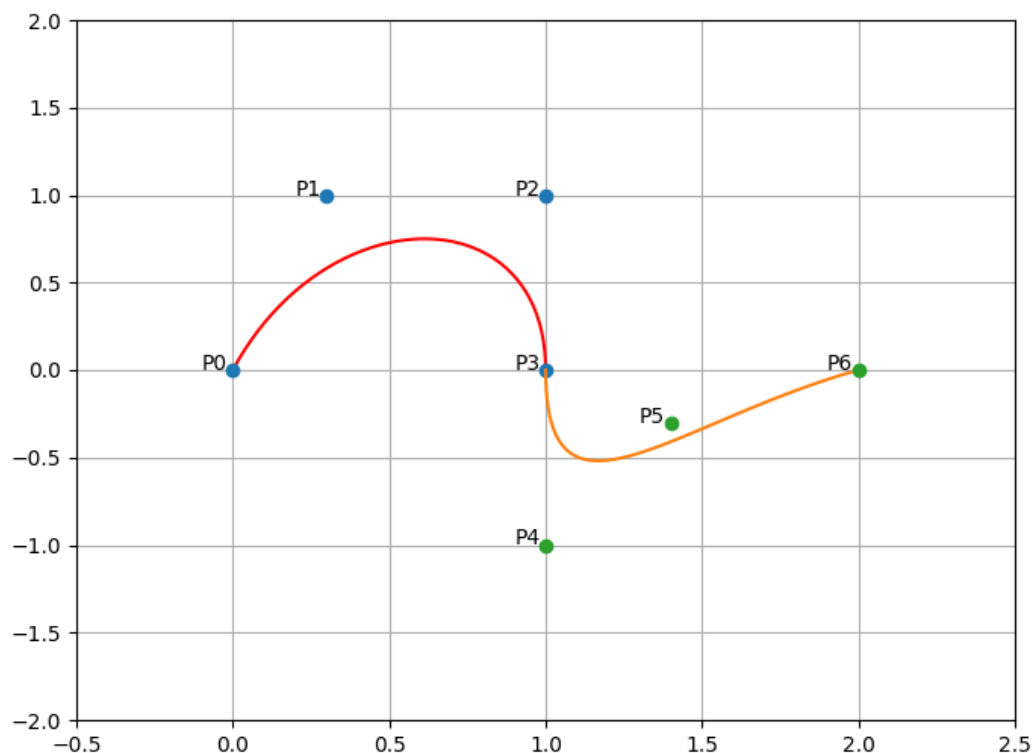


### Exercice 13 Courbes de Bézier d'ordre 3

Pour l'ensemble de l'exercice une fonction `plot_points(points_courbe,style)` est fournie permettant de représenter une liste de points (par exemple `points_courbe=[(0,0),(1,2)]` représentant l'origine avec le point de coordonnées (1,2)) selon un style à définir (par exemple `style='b-'` représenterait le segment entre les deux points représenté en bleu alors que `style='o'` va représenter séparément les deux points par un petit rond).

PREMIÈRE PARTIE : RACCORDEMENTS DE DEUX COURBES DE BÉZIER AVEC MÉTHODES NON RÉCURSIVES

On veut réaliser la représentation conjointe des deux courbes de Bézier d'ordre 3 définies respectivement par les 4 premiers points et les 4 derniers points de la liste donnée dans le fichier `bezier.py`



Pour cela on va dans un premier temps considérer que le réel  $t$  appartenant à  $[0, 1]$  par lequel on paramètre les points de chaque courbe de Bézier va être discrétisé avec  $N$  valeurs séparées à intervalles réguliers (on va choisir arbitrairement  $N=50$ ) et chaque courbe sera donc approchée par les segments reliant les 50 points obtenus (cette approximation de la courbe attendue par un ensemble de segments est d'autant meilleure que le nombre  $N$  est grand). Le lissage entre les deux courbes (de points de contrôle de  $P_0$  à  $P_3$  pour la première et de  $P_3$  à  $P_6$  pour la deuxième) est assuré par le fait que les vecteurs  $\overrightarrow{P_2P_3}$  et  $\overrightarrow{P_3P_4}$  sont colinéaires et de même sens.

**13.1.** Réaliser ceci en positionnant les points en s'appuyant sur les polynômes de Bernstein d'ordre 3 (on pourra utiliser une liste donnant ses coefficients `coefs_bern=[1,3,3,1]`). On rappelle que les coordonnées  $(x(t), y(t))$  des points  $P(t)$ , pour  $t$  variant entre 0 et 1, d'une courbe de Bézier sont données par les relations suivantes :

$$P(t) = (1-t)^3 \times P_0 + 3t \times P_1 + 3t^2 \times P_2 + t^3 \times P_3 \text{ soit :}$$

$$\begin{cases} x(t) &= (1-t)^3 \times x_0 + 3t \times x_1 + 3t^2 \times x_2 + t^3 \times x_3 \\ y(t) &= (1-t)^3 \times y_0 + 3t \times y_1 + 3t^2 \times y_2 + t^3 \times y_3 \end{cases}$$

La fonction `courbe_bezier_bern(points_control, N)` à implémenter doit effectuer ceci en prenant en entrée une liste de points de contrôle (avec le format précisé ci-dessus) et un nombre `N` définissant le nombre de points de la courbe dont on doit calculer les coordonnées et elle doit renvoyer en sortie la liste de ces points dont les coordonnées ont été calculées (avec toujours le même format attendu pour les représenter). Les `N` points correspondent aux `N` valeurs du paramètre  $t$  définies avec un espacement régulier entre 0 et 1 avec la fonction `linspace` de `numpy`.

## DEUXIÈME PARTIE : TRACÉ D'UNE COURBE DE BÉZIER AVEC MÉTHODE GÉOMÉTRIQUE

Dans un deuxième temps on souhaite réaliser le tracé d'une courbe de Bézier par une approche récursive exploitant la méthode illustrée dans le cours où la courbe de Bézier de degré 3 est obtenue à partir du parcours d'un point obtenu à partir du calcul des positions de barycentres successifs à partir des points de contrôle et ce pour chaque valeur d'un paramètre  $t$  pour lequel on choisira toujours ici des valeurs uniformément réparties sur l'intervalle  $[0; 1]$ .

**13.2.** Implémenter une première fonction `bary(A, B, t)` doit permettre de renvoyer le tuple des coordonnées du point défini comme barycentre de `A` affecté du coefficient `1-t` avec le point `B` affecté du coefficient `t`

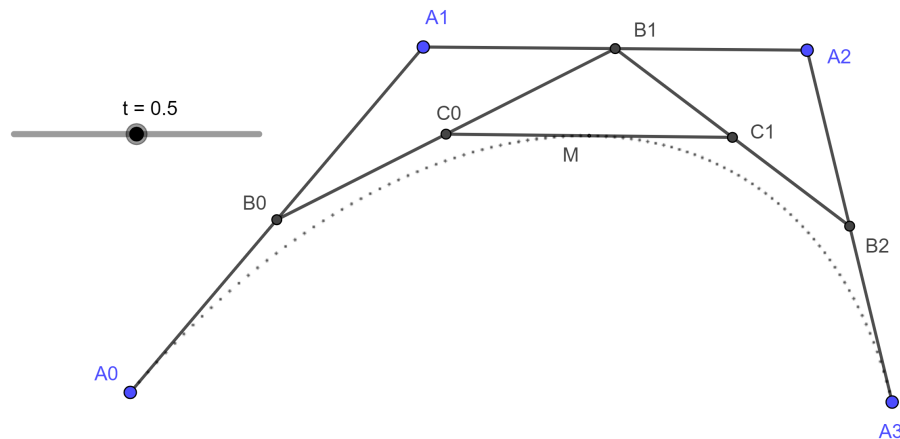
**13.3.** Implémenter la fonction `reduction(points_control, t)` définissant pour une liste de points et un certain coefficient `t`, une nouvelle liste, où chaque point correspond au barycentre de deux points successifs de la liste en entrée, affectés des coefficients `1-t` et `t` (la longueur de la liste en sortie est donc réduite d'une unité par rapport à la longueur de la liste en entrée).

**13.4.** Implémenter la fonction `point_bezier_geom(points_control, t)` donnant le point de la courbe obtenu avec une valeur de `t` donnée, après avoir suffisamment appliqué la fonction `reduction` aux points de contrôle.

**13.5.** Implémenter enfin la fonction `courbe_bezier_geom(points_control, N)` approchant la courbe de Bézier voulue à partir de la liste `points_control` en utilisant `N` valeurs pour le paramètre `t`, réparties régulièrement sur l'intervalle  $[0; 1]$  (0 et 1 devant être représentés dans ces valeurs).

TROISIÈME PARTIE : TRACÉ D'UNE COURBE DE BÉZIER AVEC ALGORITHME DE CASTELJAU

L'intérêt de l'approche présentée ici réside notamment dans le fait de pouvoir densifier le nombre de points là où la courbure est la plus forte. L'algorithme de Casteljau s'appuie sur le fait qu'on peut démontrer (en prenant exemple sur la figure suivante pour les notations des barycentres) que la courbe de Bézier de points de contrôle  $A_0, A_1, A_2, A_3$  correspond à la réunion de la courbe de Bézier de points de contrôle  $A_0, B_0, C_0, M$  avec celle de points de contrôle  $M, C_1, B_2, A_3$ . Le choix du paramètre  $t = 0.5$ , ici représenté, est régulièrement effectué pour effectuer le partage de la courbe recherchée en 2 et on suivra ce choix dans cette partie.



On peut observer que les segments définis par les points de contrôle  $A_0, B_0, C_0, M$  et  $M, C_1, B_2, A_3$  sont plus proches de la courbe attendue que les segments reliant les points de contrôle  $A_0, A_1, A_2, A_3$ . Ainsi en procédant récursivement en partageant la courbe en 2 on va pouvoir s'approcher de plus en plus de la courbe jusqu'à pouvoir considérer que la portion de courbe recherchée peut être considérée comme suffisamment proche du segment formé par le premier point de contrôle ( $A_0$  sur l'exemple) avec le dernier ( $M$  sur l'exemple). Dès lors on enregistrera le premier point de contrôle de cette portion de courbe comme représentant de celle-ci.

On va donc devoir décider d'un critère d'arrêt pour définir que le segment  $[A_0M]$  représente assez bien ce qui doit se passer entre ces deux points. Pour cela on attend à ce que les points  $A_0, B_0, C_0, M$  soient quasiment alignés dans cet ordre. On traduit ça encore en demandant que  $\overrightarrow{A_0B_0}$  et  $\overrightarrow{B_0C_0}$  soient quasiment colinéaires (avec même sens) de même que  $\overrightarrow{B_0C_0}$  avec  $\overrightarrow{C_0M}$ .

Pour cela on s'appuie sur le produit scalaire  $\vec{u} \cdot \vec{v}$  de deux vecteurs :

Pour  $\vec{u} = \begin{pmatrix} x \\ y \end{pmatrix}$  et  $\vec{v} = \begin{pmatrix} x' \\ y' \end{pmatrix}$

On a alors :  $\vec{u} \cdot \vec{v} = xx' + yy'$  et en notant  $\theta$  l'angle  $(\vec{u}, \vec{v})$ , on a un résultat sur

lequel baser notre critère de colinéarité avec sens identique (angle nul) :

$$\vec{u} \cdot \vec{v} = \|\vec{u}\| \times \|\vec{v}\| \times \cos(\theta)$$

Ainsi deux vecteurs sont colinéaires de sens identique si et seulement si  $\vec{u} \cdot \vec{v} = \|\vec{u}\| \times \|\vec{v}\|$

En considérant à un moment donné de la récursivité 4 points de contrôle  $P_0, P_1, P_2,$

$$P_3, \text{ on considère les vecteurs unitaires suivants : } \begin{cases} \vec{u}_{01} = \frac{\overrightarrow{P_0P_1}}{\|\overrightarrow{P_0P_1}\|} \\ \vec{u}_{12} = \frac{\overrightarrow{P_1P_2}}{\|\overrightarrow{P_1P_2}\|} \\ \vec{u}_{23} = \frac{\overrightarrow{P_2P_3}}{\|\overrightarrow{P_2P_3}\|} \end{cases}$$

$$\text{Ainsi par exemple } \vec{u} = \begin{pmatrix} \frac{x_{P_1} - x_{P_0}}{\sqrt{(x_{P_1} - x_{P_0})^2 + (y_{P_1} - y_{P_0})^2}} \\ \frac{y_{P_1} - y_{P_0}}{\sqrt{(x_{P_1} - x_{P_0})^2 + (y_{P_1} - y_{P_0})^2}} \end{pmatrix}$$

La condition d'arrêt s'exprimera alors de la manière suivante :

$$2 - \vec{u}_{01} \cdot \vec{u}_{12} - \vec{u}_{12} \cdot \vec{u}_{23} < \epsilon$$

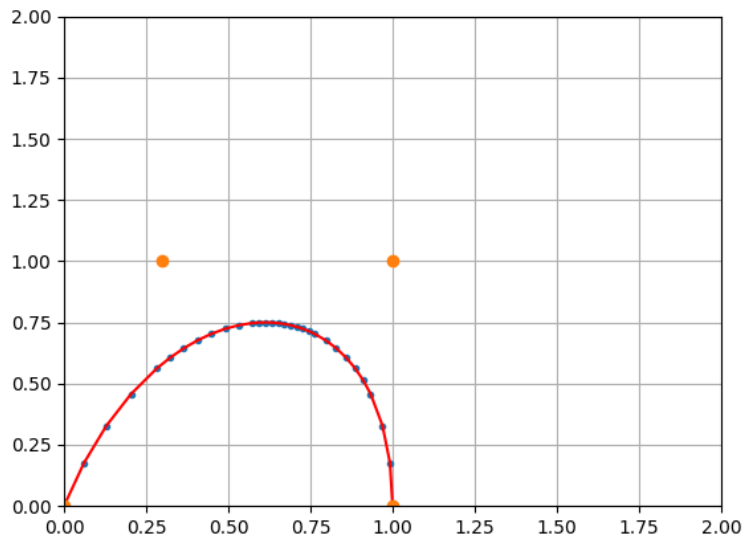
Cependant si les deux points de contrôle  $P_1, P_2$  sont confondus on doit dans ce cas particulier exprimer la condition d'arrêt de la manière suivante :

$$1 - \vec{u}_{01} \cdot \vec{u}_{23} < \epsilon$$

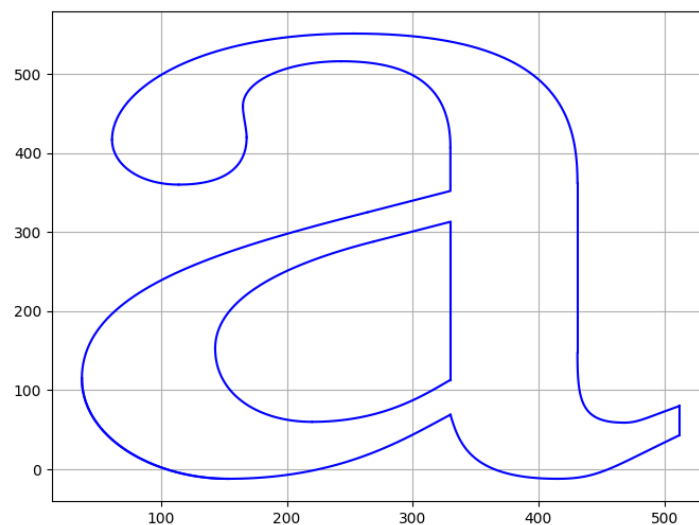
**13.6.** Réaliser la courbe suivante à partir des points définis dans le fichier fourni en appliquant la méthode décrite ci-dessus et en s'appuyant sur le squelette de code proposé. On remarquera comment le critère d'arrêt ainsi défini permet de concentrer les efforts de représentation sur les lieux de la courbe où la courbure est la plus forte. Les fonctions du squelette implémentent les idées suivantes :

- La fonction `vecteur_unitaire(P_1,P_2)` renvoie les coordonnées du vecteur unitaire  $\vec{u}_{12}$  défini ci-dessus si les points ne sont pas confondus et la valeur **False** dans le cas contraire.
- La fonction `test_alignement_4pts(points,epsilon)` renvoie la valeur de vérité correspondant à la condition d'arrêt définie précédemment pour une liste de 4 points.
- La fonction `division_courbe_bezier(points_control)` renvoie à partir d'une liste de 4 points de contrôle un tuple donnant les deux liste de points de contrôle qui vont séparer la courbe de Bézier en deux courbes dont on fera la réunion. A partir de l'exemple illustré ci-dessus `division_courbe_bezier([A_0, A_1, A_2, A_3])` va renvoyer `([A_0,B_0,C_0, M], [M, C_1, B_2, A_3])`
- La fonction `courbe_bezier_cast_helper(points_control,epsilon,points_courbe)` va effectuer le travail récursif attendu pour l'algorithme de Casteljau en ajoutant à la liste `points_courbe` le premier point de la liste `points_control` lorsque ceux-ci sont considérés alignés et va sinon rediviser la liste `points_control` pour effectuer deux appels récursifs pour approcher la courbe de Bézier attendue définie par `points_control` par la jonction de deux courbes définies par cette division de `points_control` en deux listes.

- La fonction `courbe_bezier_cast(points_control,epsilon)` est celle qu'on exécutera pour obtenir l'ensemble des points de la courbe. Elle initialise une liste `points_courbe` avec une liste vide, appelle la fonction `courbe_bezier_cast_helper` et finit par ajouter le dernier point de la liste `points_control` à la liste `points_courbe` pour enfin renvoyer cette liste donnant l'ensemble des points de la courbe approchant la courbe de Bézier attendue.



Le caractère suivant est obtenu en suivant les consignes du fichier `bezier.py` (un commentaire comme `#P0->P3` demande la réalisation de la courbe de Bézier de points de contrôle  $P_0$ ,  $P_1$ ,  $P_2$  et  $P_3$  et un commentaire comme `#P6P7` demande le tracé du segment  $[P_6P_7]$ ).



Exercice 14 Calcul des termes d'une suite récurrente

**14.1.** Écrire une fonction python `suite1(f,u0,n)` qui retourne un tableau contenant les valeurs  $[u_0, \dots, u_n]$  de la suite définie par la récurrence  $u_{n+1} = f(u_n)$ .

**14.2.** Écrire une fonction python `suite2(f,u0,l, $\varepsilon$ )` qui retourne `n0`, le premier rang à partir duquel  $|u_n - l| < \varepsilon$ .

**14.3.** Tester avec  $f(x) = \sqrt{2-x}$  et  $u_0 = 0$  en prenant  $\varepsilon = 10^{-8}$  et pour  $l$  la limite de la suite donnée par le théorème du point fixe, on doit obtenir :

```
def f1(x) :  
    return math.sqrt(2-x)
```

```
suite2(f1,0,1,10**(-8))  
-> 27.
```

```
suite1(f1,0,10)  
-> [0, 1.4142135623730951, 0.7653668647301795, 1.1111404660392046, 0.9427934736519,  
    1.0282054883864433, 0.9857963844595682, 1.007076767451435, 0.9964553339455638,  
    1.0017707652224817, 0.9991142250901637, 1.0004427894236814]
```

### Exercice 15 Utilisation de matplotlib

Python propose un package, `matplotlib`, permettant d'afficher simplement des graphiques.

Dans (quasiment) tous les programmes `python` dans lesquels nous ferons de l'affichage graphique, nous utiliserons :

- `numpy` que l'on importera par la commande :

```
import numpy as np
```

- `matplotlib.pyplot` que l'on importera par la commande :

```
import matplotlib.pyplot as plt
```

**Premier exemple** On souhaite afficher le graphique de la fonction

$$f : x \mapsto 3x^4 + 8x^3 - 6x$$

sur l'intervalle  $[-3, 2]$  :

- a. on commence par discrétiser l'intervalle  $[-3, 2]$  :

```
x = np.linspace(-3, 2, 100)
```

- b. on calcule les ordonnées correspondantes :

```
y = 3*x**4+8*x**3-6*x
```

- c. on calcule l'image qui va être affichée :

```
plt.plot(x, y, 'b:', linewidth=5)
```

- d. enfin, on affiche l'image :

```
plt.show()
```

Remarques :

- Pour afficher un graphique, `matplotlib` utilise 2 tableaux :
  - un tableau d'abscisses : ici représenté par `x`
  - un tableau d'ordonnées : ici représenté par `y`
- Rappel : la commande `linspace(a,b,n)` du package `numpy` permet de placer `n` valeurs régulièrement réparties dans l'intervalle  $[a, b]$ .
- Rappel : les tableaux de base de python ne permettent pas de faire des opérations terme à terme :

```
[1, 2, 3] * 2 -> [1, 2, 3, 1, 2, 3]
```

On utilise donc des tableaux `numpy`, qui se manipulent facilement.

Attention cependant, lorsque vous devez utiliser des fonctions comme `cos`, `ln`, `exp`, ... à utiliser les fonctions correspondantes dans `numpy` : `np.cos`, `np.log`, `np.exp`

- l'affichage se fait en deux temps :

1. calcul de l'image à afficher (avec une ou plusieurs commandes `plot`)
2. affichage de l'image à l'écran avec la commande `show`



- la commande `plot` peut prendre plusieurs arguments. En considérant que  $\mathbf{x}=[x_0, x_1, \dots, x_n]$  et  $\mathbf{y}=[y_0, y_1, \dots, y_n]$  sont deux tableaux (`numpy` ou non) de même taille, les principales utilisations sont :
  - `plot(x,y)` : affiche un ensemble de points  $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$  avec le style de ligne par défaut.
  - `plot(y)` : affiche un ensemble de points  $(0, y_0), (1, y_1), \dots, (n-1, y_n)$  avec le style de ligne par défaut.
- plusieurs options sont disponibles :
  - `color`, qui correspond à la couleur de la ligne et des marqueurs, qui peut valoir `'blue'` (ou `'b'`), `'red'` (ou `'r'`), `'green'` (...), `'orange'`, `'yellow'`, ... et aussi d'autres représentations (en hex RGB par exemple : `'#f6a154'`)
  - `marker`, pour mettre en évidence les points dessinés, qui peut valoir `'o'` (cercles), `'x'` (croix en 'x'), `'+'` (croix en '+'), `'s'` (carrés), `'v'` (triangles vers le bas), `'^'` (triangles vers le haut) ...
  - `linestyle`, qui correspond au style de ligne, qui peut valoir `'solid'` ou `'-'`, `'dashed'` ou `'--'`, `'dashdot'` ou `'-.'`, `'dotted'` ou `':'`
  - `linewidth`, qui correspond à l'épaisseur de la ligne,
  - `markersize`, qui correspond à la taille des marqueurs
  - ainsi on pourra par exemple exécuter la commande :

```
plot(x, y, color = 'r', marker = '+', linestyle = ':',
      linewidth = 4, markersize = 20)
```

- certaines options peuvent être utilisées en raccourci : si l'on souhaite une ligne en pointillés, rouge avec des marqueurs en `'x'`, on peut écrire : `plot(x,y,'rx:')`

**15.1.** Affichez, avec le style de votre choix, et sur **un seul** graphique, les courbes représentant les fonctions suivantes sur l'intervalle  $[-5, 5]$  :

- $f(x) = x + \sin(x)$
- $g(x) = \frac{x^3 + 5}{x^2 + 2}$
- $h(x) = \ln(x^4 + 1) - 4$

**15.2.** Dans notre cas, étant donné que nous souhaitons afficher des fonctions, il est souvent utile d'ajouter des informations au graphique :

- la commande `plt.grid()` permet l'affichage d'une grille adaptée en fonction de la taille de la fenêtre, on peut lui passer en argument une couleur, un style de ligne et une épaisseur de ligne :

```
plt.grid(color='grey', linestyle = '--')
```

- lorsque plusieurs fonctions sont dessinées, on peut dessiner une petite légende avec le nom de chaque fonction :

— récupérez la valeur de retour du `plot` dans une variable :

```
c_f, = plt.plot(x, f, ...)
```

*On ne récupère pas toute la valeur de retour de `plot`, seulement la première partie, d'où la `','` après le `c_f`.*

— ajoutez une étiquette à la courbe (ici `'f'`) :

```
c_f.set_label('f')
```

— affichez la légende :

`plt.legend()`

Dans certains cas, il peut être intéressant d'afficher les fonctions les unes à côté des autres. Pour cela, on peut utiliser la commande `subplot`.

Avant le `plot` de chacune des fonctions, on fait appel à une commande `subplot`. `subplot` permet de diviser la fenêtre d'affichage en une 'grille' de cases dans lesquelles on dessinera les fonctions. Par exemple, la commande :

`plt.subplot(231)`

signifie qu'on divise la fenêtre d'affichage en une grille de 2 lignes par 3 colonnes et que le `plot` qui suivra sera dessiné dans la case n°1.

1	2	3
4	5	6

Ordre du `subplot`

Ainsi si l'on souhaite prendre une grille de 2 par 3, et afficher les fonctions  $f$ ,  $g$  et  $h$  respectivement dans les cases 1, 2 et 4, on écrira :

```
plt.subplot(231)
plt.plot(x,f,...)
plt.subplot(232)
plt.plot(x,g,...)
plt.subplot(234)
plt.plot(x,h,...)
```

Et on obtient un graphique ressemblant à :

