1 Calculs sur une série statistique univariée

1.1 Calcul de moyenne ♥

La **moyenne** arithmétique des réels $x_1,\ x_2,\ \ldots,\ x_n$, notée $\overline{x}$, est définie par :

$$\boxed{\overline{x} = \frac{1}{n}\sum_{k=1}^{n} x_k.}$$

Lorsque la série statistique est donnée sous la forme d'une liste, on peut calculer la moyenne de cette série en n'effectuant un seul parcours, comme illustrer ci-dessous.

```
def moyenne(serie):
    """renvoie la moyenne des valeurs de la liste de nombres serie"""
    sigma = 0
    for nb in serie: # la variable nb décrit tous les éléments de serie
        sigma += nb
    return sigma/len(serie)
```

Moyenne des valeurs d'une liste de nombres

On peut remarquer que l'opération de division (par la taille de la liste) n'est effectuée qu'après le calcul de la somme des termes de la liste de façon à réduire le nombre d'opérations dites élémentaires.

1.2 Calcul de variance ♥

On appelle **variance** des réels $x_1,\ x_2,\ \ldots,\ x_n$ le réel noté σ^2 défini par :

$$\sigma^2 = \frac{1}{n}\sum_{k=1}^{n} (x_k - \overline{x})^2.$$

On remarque que chacune des valeurs de la série est appelée deux fois dans cette formule : une fois explicitement et une fois dans le calcul de la moyenne $\overline{x}$ de la série. Or, le **théorème de König-Huygens** assure que :

$$\sigma^2 = \left(\frac{1}{n}\sum_{k=1}^{n} x_k^2\right) - \overline{x}^2.$$

On peut alors optimiser le calcul de la variance d'une série statistique en couplant "à la volée" le calcul de la moyenne à celui de la variance, i.e. en un seul parcours de la liste, comme l'illustre le code ci-dessous.

```
def variance(serie):
    """renvoie la variance des valeurs de la liste de nombres serie"""
    n = len(serie)
    sigma, sigma_carre = 0, 0
    for nb in serie: # la variable nb décrit tous les éléments de serie
        sigma += nb
        sigma_carre += nb**2
    return (sigma_carre/n) - (sigma/n)**2
```

Variance des valeurs d'une liste de nombres

2 Algorithmes de recherche séquentielle

2.1 Recherche du maximum dans une liste de nombres ♥

L'algorithme de recherche du maximum d'une liste de nombres noté `liste` n'effectue qu'**un seul** parcours, **dans sa totalité** de la liste. Le principe est qu'on stocke la première valeur de de la liste dans une variable appelée `max_actuel` puis qu'on parcourt le reste des nombres de la liste et, qu'à chaque itération, on stocke le maximum des valeurs parcourues jusque là dans `max_actuel`.

```
def maximum(liste):
    """renvoie l'élément maximal d'un liste (non-vide) de nombres"""
    n = len(liste)
    # initialisation
    max_actuel = liste[0]
    # parcours par indice de la liste
    for x in liste[1:]:
        if x > max_actuel:
            max_actuel = x
    return max_actuel
```

Algorithme de recherche du maximum dans une liste de nombres (version parcours par indice)

```
def maximum(liste):
    """renvoie l'élément maximal d'un liste (non-vide) de nombres"""
    n = len(liste)
    # initialisation
    max_actuel = liste[0]
    # parcours par indice de la liste
    for k in range(1,n):
        if liste[k] > max_actuel:
            max_actuel = liste[k]
    return max_actuel
```

Algorithme de recherche du maximum dans une liste de nombres (version parcours par indice)

2.2 Recherche d'un élément dans une liste ♥

Le but de cet algorithme est de rechercher un élément `x` dans une liste `seq`.

Le principe de l'algorithme est de parcourir la liste (au plus une seule fois et pas nécessairement en totalité) jusqu'à trouver l'élément recherché. À chaque étape, on compare l'élément recherché avec un élément d'indice `k` de la liste `seq`.

- S'il y a correspondance, on "**lève un drapeau**" pour signaler que l'élément a bien été trouvé en donnant à la variable `found` la valeur `True`.
- Sinon, on passe à l'élément "suivant" dans la liste, en incrémentant la variable d'indexation `k`.

Ne connaissant pas le nombre d'itérations nécessaires à trouver l'élément, on utilise une boucle conditionnelle **`while`**. La recherche continue tant que l'élément `x` n'a pas été trouvé et qu'il reste à "tester" des éléments de la liste `seq`.

Initialement, on n'a pas encore trouvé l'élément recherché, donc la variable `found` prend la valeur `False`.

On a construit la variable `found` de façon à ce qu'elle indique si l'élément `x` appartient ou non à la liste `seq`. On renvoie alors la valeur de cette variable booléenne.

On donne ci-dessous un exemple de fonction codant l'algorithme de recherche d'un élément dans une liste à l'aide de la notion de drapeau.

```
def recherche(x,seq):
    """renvoie True si x appartient à la liste seq, False sinon"""
    # initialisation
    n, k = len(seq), 0
    found = False
    # parcours de la liste
    while not found and k < n:
        if seq[k] == x:
            found= True
        else:
            k += 1
    return found
```

Algorithme de recherche d'un élément dans une liste (à l'aide d'une variable drapeau)

On peut aussi se passer d'une variable drapeau en exploitant le fait que l'instruction **`return`** arrête l'exécution d'une fonction. On peut alors réaliser ici un parcours par élément.

```
def recherche_bis(x,seq):
    """renvoie True si x appartient à
        la liste seq, False sinon"""
    # parcours de la liste
    for y in seq
        if x == y:
            return True
    return False
```

Variante de l'algorithme de recherche

Remarque 1

> On peut adapter cet algorithme à la recherche d'un élément dans n'importe quel séquence (liste, chaîne de caractères, array, etc).

2.3 Recherche d'un mot dans une chaine de caractères ♥

Le but de cet algorithme est de vérifier si une chaine de caractères, qu'on notera `chaine`, contient une sous-chaine donnée qu'on notera `mot`.
Pour faciliter le travail, on propose d'écrire deux fonctions :

- une fonction `frecherche_mot_dans_chaine_ici` qui recherche le mot dans la chaîne à une place donnée ;
- une fonction `recherche_mot_dans_chaine` qui recherche le mot à toutes les positions valides (à déterminer), en utilisant la fonction précédente.

```
def recherche_mot_dans_chaine_ici(mot,chaine,i):
    """fonction booléenne qui renvoie si mot = chaine[i:i+len(mot)]"""
    long_mot = len(mot)
    for j in range(long_mot) :
        if chaine[i+j] == mot[j]:
            j += 1
        else:
           return False
    return True
```

Algorithme de recherche d'un mot dans une chaine à l'indice i

En notant `long_chaine` la longueur de la chaîne et `long_mot` la longueur du mot, on peut alors chercher le mot dans la chaîne à tous les indices `i` entre 0 et `long_chaine`−`long_mot` (compris).

```
def recherche_mot_dans_chaine(mot,chaine):
    """fonction booléenne qui renvoie si chaine contient mot"""
    long_chaine, long_mot = len(chaine), len(mot)
    for i in range(long_chaine - long_mot + 1):
        if recherche_mot_dans_chaine_ici(mot,chaine,i):
            return True
    return False
```

Algorithme de recherche d'un mot dans une chaine

Pour les puristes, on peut envisager de n'écrire qu'une seule fonction :

```
def recherche_mot_dans_chaine(mot,chaine):
    """fonction booléenne qui renvoie si chaine contient mot"""
    long_chaine, long_mot = len(chaine), len(mot)
    for i in range(long_chaine - long_mot + 1):
        j = 0
        while j < long_mot and chaine[i+j] == mot[j]:
            j += 1
        if j == long_mot:
            return True
    return False
```

Algorithme de recherche d'un mot dans une chaine

3 Fonctions récursives

3.1 Calcul de factoriels ♥

La fonction suivante prend en argument un entier n et calcule $n!$ récursivement en utilisant que $0! = 1$ et la relation de récurrence : $\forall n \in \mathbb{N}^*,\ n! = n \times (n-1)!$.

```
def factoriel(n):
    if n == 0:
        return 1
    else:
        return n * factoriel(n-1)
```

3.2 Algorithme d'exponentiation rapide

La fonction suivante prend en argument un nombre x et entier n et calcule x^n récursivement en utilisant que $x^0 = 1$ et la relation de récurrence :

$$\forall n \in \mathbb{N}^*, x^n = \begin{cases} \left(x^2\right)^{\frac{n}{2}} & \text{si } n \text{ est pair} \\ x \times \left(x^2\right)^{\frac{n-1}{2}} & \text{si } n \text{ est impair.} \end{cases}$$

```
def exp_rapide(x,n):
    if n == 0:
        return 1
    elif n%2 == 0:
        return exp_rapide(x**2,n//2)
    else:
        return x * exp_rapide(x**2,n//2)
```

On remarquera que la valeur de `n//2` *est* $\frac{n}{2}$ *si* n *est pair et* $\frac{n-1}{2}$ *si* n *est impair, justifiant son utilisation dans les deux cas de la définition de la fonction ci-dessus.*

4 Algorithmes de recherche dichotomique

4.1 Recherche dichotomique dans un tableau trié ♥

Le but de cet algorithme est de rechercher un élément `x` dans un tableau `tab` (préalablement) trié (selon l'ordre des réels ou l'ordre lexicographique par exemple).

Le principe de cet algorithme est de rechercher cet élément selon le paradigme "*diviser pour mieux régner*" : à chaque étape, on divise par deux (environ) la taille du tableau dans lequel on effectue la recherche de l'élément `x` (d'où le nombre de recherche **dichotomique**).

A chaque étape, si on cherche dans le tableau parmi les éléments d'indice `g` et `d`, on compare l'élément recherché `x` à l'élément médian du sous-tableau `tab[g:d]`, d'indice $m = \left\lfloor \frac{g+d}{2} \right\rfloor$.

- S'il y a égalité, on renvoie `True` pour signifier que l'élément `x` appartient bien au tableau `tab`.
- Sinon, on réduit "l'intervalle de recherche" au sous-tableau `tab[g:m]` ou `tab[m+1:d+1]`, en modifiant la valeur des variables `g` et `d` (indices des bornes gauche et droite de l'intervalle de recherche).

```
m = (g+d)//2
if x == tab[m]:
    return True
elif x < tab[m]:
    d = m - 1
else:
    g = m + 1
```

Cette recherche continue tant que l'élément recherché n'a pas été trouvé et qu'il reste des éléments à comparer, d'où la nécessité d'utiliser une boucle conditionnelle.

Dans le cas où `g=d`, il ne reste plus qu'un élément à tester et on a `m=g=d`. S'il ne correspond pas à celui recherché, la recherche doit terminer et l'une des alternatives du test conditionnel entraine `g>d`.

Puisqu'on peut sortir d'une boucle à l'aide de l'instruction **`return`**, la condition (suffisante) de sortie de boucle correspond au cas où il ne reste plus d'élément à comparer. On continue donc la recherche tant que `g<d` (ce qui peut paraitre anecdotique voire paradoxal au premier abord, mais qui se comprend parfaitement après réflexion).

A la sortie de boucle, on renvoie `False` : en effet, si l'algorithme n'a pas terminé mais qu'on est sorti de la boucle, c'est que l'élément recherché n'a pas été trouvé.

Le code ci-dessus présente une fonction booléenne de recherche dichotomique d'un élément `x` dans un tableau trié `tab`.

```
def recherche_dichotomique(x,tab):
    """fonction booléenne de recherche dichotomique
       de l'élément x dans le tableau trié tab"""
    # initialisation
    g, d = 0, len(tab) - 1
    # parcours dichotomique
    while g <= d:
        m = (g+d)//2
        if x == tab[m]:
            return True
        elif x < tab[m]:
            d = m - 1
        else:
            g = m + 1
     return False
```

Algorithme de recherche dichotomique dans un tableau trié

4.2 Algorithme de recherche dichotomique d'un zéro d'une fonction ♥

Le but de cet algorithme est de rechercher une **valeur approchée** d'un zéro d'une fonction **continue** sur un intervalle donné, **lorsqu'il existe**. La fonction `f`, les bornes `a` et `b` de l'intervalle ainsi que l'erreur d'approximation `epsilon` sont fournis par l'utilisateur comme arguments d'entrée.

Le principe est qu'à chaque itération, on divise par deux la longueur de l'intervalle dans lequel on recherche le zéro éventuel : si on le recherche dans l'intervalle $[g,d]$ et si on note $m = \dfrac{g+d}{2}$, il se situe dans l'intervalle $[g,m]$ ou dans l'intervalle $[m,d]$, par monotonie de la fonction `f`. Il suffit donc, selon les cas, d'affecter à la variable `g` ou la variable `d`, la valeur de `m` de façon à réduire l'intervalle de recherche à l'étape suivante.

```
m = (g+d)/2
if f(g)*f(m) <= 0:
    d = m
else:
    g = m
```

La recherche dichotomique continue tant que la longueur de l'intervalle de recherche est supérieure au double de l'erreur d'approximation `epsilon`. Une fois cette condition fausse, on renvoie en sortie de boucle la valeur médiane du dernier intervalle $[g,d]$: la distance de ce réel à tout entier de l'intervalle $[g,d]$ est bien inférieure à `epsilon`.

Enfin, on commence par s'assurer de se trouver bien dans les conditions de réalisation de l'algorithme :

- les valeurs données définissent bien un intervalle (`a < b`),
- le zéro existe bien (et est unique) par le théorème de la bijection (`f(a)*f(b) <= 0`),
- et l'erreur d'approximation est bien strictement positive (`epsilon > 0`).

Les hypothèses de continuité et de stricte monotonie ne sont pas vérifiées car elles ne sont pas vérifiables : l'ensemble des flottants (quelque soit la norme et la précision) ne peut avoir la puissance du continu puisqu'il est fini.

```
def recherche_zero_dichotomie(f,a,b,epsilon):
    """renvoie, si elle existe, une valeur approchée à epsilon près
       du zéro d'une fonction monotone sur [a,b], par dichotomie"""
    if a > b and f(a)*f(b) > 0 and epsilon <= 0:
        print("On ne peut pas conclure si f s'annule sur [a,b]")
    g, d = a, b
    while abs(d-g) > 2*epsilon:
        m = (g+d)/2
        if f(g)*f(m) <=0:
            d = m
        else:
            g = m
    return (g+d)/2
```

Algorithme de recherche du zéro d'une fonction monotone

5 Algorithmes de tri

Aucun algorithme n'est spécifiquement au programme. Un étudiant doit connaître au moins un algorithme de tri - de préférence (personnelle) le tri rapide ou le tri fusion.

5.1 Tri à bulles

On rappelle le principe du tri bulle :

- on parcourt la liste en comparant deux éléments consécutifs qu'on permute s'ils ne sont pas bien ordonnés ; on les permute ;
- à la fin du parcourt de la liste, l'algorithme répète l'étape précédente tant qu'il trouve des éléments à permuter ;
- lorsque le tableau a été parcouru sans qu'aucune permutation n'ait été effectuée, l'algorithme termine : le tableau est trié.

```
def tri_bulle(liste0):
    """trie liste0 en créant au préalable une copie superficielle """
    liste = liste0[:] # copie superficielle de liste0
    n = len(liste)
    exchange_flag = True # indicateur d'une permutation
    while exchange_flag:
        exchange_flag = False
        for k in range(n-1):
            if liste[k] > liste[k+1]:
                liste[k], liste[k+1]= liste[k+1], liste[k]
                exchange_flag = True
    return liste
```

Algorithme de tri à bulles d'une liste de nombres

5.2 Tri par insertion

On rappelle le principe du tri par insertion :

- à l'étape 0, la première valeur est triée (par rapport à elle-même) ;
- à l'étape i, toutes les valeurs du tableau d'indices compris entre 0 et $i-1$ sont triées et on cherche à "insérer" l'élément d'indice i parmi ces éléments triés (en respectant l'ordre choisi) de la façon suivante :
  - on stocke la valeur d'indice i dans une variable temporaire,
  - on décale les valeurs du tableau jusqu'à trouver la place de la valeur de la variable temporaire.

```
def tri_insertion(liste0):
    """ trie liste0 en créant
    une copie superficielle """
    liste = liste0[:] # copie
    n = len(liste)
    for i in range(1,n):
        x = liste[i]
        j = i
        while j>0 and liste[j-1] > x:
            liste[j] = liste[j-1]
            j -= 1
        liste[j]= x
    return T
```

Tri par insertion non en place

```
def tri_insertion_bis(liste):
    """trie sur place liste"""
    n = len(liste)
    for i in range(1,n):
        x = liste[i]
        j = i
        while j>0 and liste[j-1] > x:
            liste[j] = liste[j-1]
            j -= 1
        liste[j]= x
```

Tri par insertion en place

5.3 Tri par sélection

Le principe du tri par sélection est de placer à la k-ème itération le k-ème plus petit élément de la séquence à trier. Cela est facilement réalisable car il suffit de chercher le minimum des valeurs non encore triées.

```
def tri_selection(seq):
    """trie la liste seq par sélection (tri en place)"""
    n = len(seq)
    for k in range(n):
        # recherche du (k+1)-ème plus petit élément, i.e. min(seq[k:])
        i_min = k
        for i in range(k+1,n):
            if seq[i] < seq[i_min]:
                i_min = i
        seq[k], seq[i_min] = seq[i_min], seq[k]
    return seq
```

Algorithme de tri par sélection

5.4 Tri rapide (spé)

Le principe du tri rapide (*quicksort* en anglais) est de partitionner une liste de nombres en deux sous-listes dont les éléments sont respectivement inférieures et strictement supérieures à un pivot arbitrairement choisi (ici la première valeur de la liste). Les deux sous-listes sont ensuite triées récursivement ce qui permet de trier la liste initiale.

```
def partition(seq):
    """renvoie le triplet (seq_inf, pivot, seq_sup) où :
       - pivot est le premier élément de seq,
       - seq_inf (res. seq_sup) est la liste des éléments restant inférieurs
         (resp. strictement supérieurs) au pivot"""
    pivot, seq_inf, seq_sup = seq[0], [], []
    for x in seq[1:]:
        if x <= pivot:
            seq_inf.append(x)
        else:
            seq_sup.append(x)
    return seq_inf, pivot, seq_sup

def tri_rapide(seq):
    """trie récursivement la séquence seq par l'algorithme du tri rapide"""
    if len(seq) <= 1:
        return seq
    else:
        seq_inf, pivot, seq_sup = partition(seq)
        return tri_rapide(seq_inf) + [pivot] + tri_rapide(seq_sup)
```

Algorithme du tri rapide

5.5 Tri fusion (spé)

Le principe du tri fusion (*merge sort* en anglais) - est de partitionner une liste de nombres en deux sous-listes de longueurs égales (ou différents d'au plus un) qu'on trie récursivement puis qu'on fusionne une une seule liste triée.

```
def fusion(L1,L2):
    """prend en argument deux listes triées L1 et L2
         et renvoie la version triée de L1+L2"""
    liste = []
    while L1 != [] and L2 != []:
        x1, x2 = L1[0], L2[0]
        if x1 < x2:
            liste.append(x1)
            L1.pop(0)
        else:
            liste.append(x2)
            L2.pop(0)
    if L1 == []:
        return liste + L2
    else:
        return liste + L1


def tri_fusion(liste):
    """trie récursivement la séquence seq par l'algorithme du tri fusion"""
    if len(liste) <= 1:
        return liste
    else:
        n = len(liste)
        L1, L2 = tri_fusion(liste[:n//2]), tri_fusion(liste[n//2:])
        return fusion(L1,L2)
```

Algorithme du tri fusion

6 Simulation de variables aléatoires ♥♥♥

On importe le module `random` via l'instruction **`import`** `random as rd` et le module `numpy` via **`import`** `numpy as np`.

```
def loi_uniforme_discrete(a,b):
    return rd.randint(a,b)
```

simulation de la loi uniforme sur $[\![a,b]\!]$

```
def loi_Bernoulli(p):
    if rd.random() < p:
        return 1
    else:
        return 0
```

simulation de la loi de Bernoulli de paramètre p

```
def loi_binomiale(n,p):
    s = 0
    for k in range(n):
        if rd.random() < p:
            s += 1
    return s
```

simulation de la loi biomiale de paramètres n et p

```
def loi_geometrique(p):
    rang = 1
    while rd.random() > p:
        rang += 1
    return rang
```

simulation de la loi géométrique de paramètres p (spé)

```
def loi_uniforme_continue(a,b):
    return a + (b-a)*rd.random()
```

simulation de la loi uniforme sur $[a,b[$ (spé)

```
def loi_exponentielle(l):
    u = rd.random()
    return -np.log(1-u)/l
```

simulation de la loi exponentielle de paramètre λ (spé)

```
def loi_normale(m, sigma):
    return rd.gauss(mu, sigma)
```

simulation de la loi normale d'espérance m et d'écart-type σ (spé)

7 Méthodes numériques

7.1 Méthode des rectangles ♥

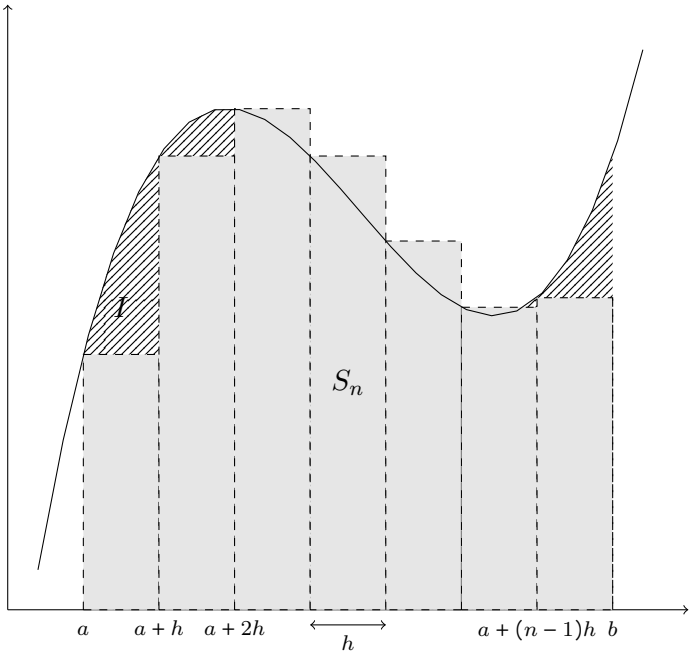
La **méthode des rectangles** est une méthode d'analyse numérique permettant d'approximer l'intégrale

$$I = \int_a^b f(x)\,\mathrm{d}x$$

d'une fonction f sur le segment $[a,b]$ par l'une des deux **sommes de Riemann** :

$$\boxed{S_n = \frac{b-a}{n}\sum_{k=0}^{n-1} f\left(a+k\frac{b-a}{n}\right) \quad \text{ou} \quad T_n = \frac{b-a}{n}\sum_{k=1}^{n} f\left(a+k\frac{b-a}{n}\right).}$$

On notera $h = \dfrac{b-a}{n}$ le **pas** de la subdivision.




```
def integration_rectangle(f,a,b,n):
    """approximation de l'intégrale de f entre a et b par
    la méthode des rectangles (n rectangles) à pas constant"""
    h = (b-a)/n
    somme = 0
    for k in range(n):
        somme += f(a + k*h)
    return h*somme
```

Méthode des rectangles

7.2 Méthode d'Euler ♥

Le but de la **méthode d'Euler** est de résoudre de manière approchée, sur un intervalle $[a, b]$, une équation différentielle dite ordinaire avec condition initiale), c'est-à-dire de la forme :

$$\begin{cases} y(a) = y_0 \\ \forall t \in [a, b],\ y'(t) = F(t, y(t)). \end{cases}$$

Pour cela, on veut construit une suite finie de $n+1$ points $(y_k)_{0 \leqslant k \leqslant n}$ approchant les valeurs exactes $(y(t_k))_{0 \leqslant k \leqslant n}$ où :

$$\forall k \in [\![0, n]\!],\ t_k = a + kh \quad \text{en notant} \quad h = \frac{b-a}{n}.$$

La construction des valeurs approchées se base l'approximation $y'(t) \approx \dfrac{y(t+h) - y(t)}{h}$ lorsque le **pas** h est suffisamment petit. Ainsi, on obtient :

$$y(t+h) \approx y(t) + hy'(t) = y(t) + hF(t, y(t)).$$

On peut donc prédire la valeur de la solution y à l'instant $t+h$ à partir de sa valeur à l'instant t.
La suite des valeurs approchées calculées par la méthode d'Euler est définie par la relation de récurrence ci-dessous, appelée, **schéma numérique de la méthode d'Euler** :

$$\boxed{\begin{cases} y_0 = y(t_0) \quad \textit{condition initiale (seule valeur exacte connue)} \\ \forall k \in [\![0, n-1]\!],\ y_{k+1} = y_k + hF(t_k, y_k). \end{cases}}$$

```
def euler(F,a,b,y_0,n):
    t = [a]
    y = [y_0]
    h = (b-a)/n
    for k in range(n):
        y.append(y[k]+h*F(t[k],y[k]))
        t.append(t[k]+h)
    return t, y
```

Méthode d'Euler

8 Algorithmes sur les graphes

On considère un graphe à n sommets numérotés de 0 à $n-1$ et donné par sa liste d'adjacence ou sa matrice d'adjacence.

8.1 Parcours en largeur

L'algorithme de parcours en largeur (*BFS - Breadth-First Search*) est utilisé pour explorer un graphe de manière itérative en commençant par un sommet source donné. Il fonctionne en visitant d'abord tous les voisins directs du sommet source, puis en visitant les voisins de ces voisins, et ainsi de suite, en suivant un niveau de profondeur à la fois.

Le processus commence par marquer le sommet source comme visité puis à le placer dans une file d'attente. Tant que la file d'attente n'est pas vide, on répète les deux étapes ci-dessous.

1. On retire le sommet en tête de la file d'attente.
2. On explore tous les voisins non visités de ce sommet ; on les marque comme visités et on les ajoute à la fin de la file d'attente.

```
def bfs(mat_adj,depart):
    nb_sommets = np.shape(mat_adj)[0]
    file = [depart]
    sommets_visites = [depart]
    while file != []:
        # sommet actuel
        sommet = file.pop(0)
        # recherche des voisins non visités
        for voisin in range(nb_sommets):
            if mat_adj[sommet,voisin]==1 and voisin not in sommets_visites:
                file.append(voisin)
                sommets_visites.append(voisin)
    return sommets_visites
```

Algorithme de parcours en largeur d'un graphe donné par sa matrice d'adjacence

```
def bfs(liste_adj,depart):
    nb_sommets = len(liste_adj)
    file = [depart]
    sommets_visites = [depart]
    while file != []:
        # sommet actuel
        sommet = file.pop(0)
        # recherche des voisins non visités
        for voisin in liste_adj[sommet]:
            if voisin not in sommets_visites:
                file.append(voisin)
                sommets_visites.append(voisin)
    return sommets_visites
```

Algorithme de parcours en largeur d'un graphe donné par sa liste d'adjacence

8.2 Parcours en profondeur (spé)

L'algorithme de recherche en profondeur (*DFS - Depth-First Search*) est également utilisé pour explorer un graphe, mais contrairement à l'algorithme BFS, il explore le graphe en descendant le plus loin possible le long d'une branche avant de faire marche arrière.
On propose ici une implémentation de cet algorithme à l'aide d'une pile (de sommets à traiter). L'algorithme de recherche en profondeur commence en choisissant un sommet source comme point de départ qu'on ajoute à la pile de sommets à traiter. Tant que la pile est non vide, on répète les étapes suivantes :

1. On extrait le sommet en haut de la pile.
2. S'il n'est pas marqué comme visité, on le marque comme visité et on ajoute ses voisins dans la pile des sommets à traiter. Si jamais aucun voisin non visité n'est trouvé à l'issue de l'étape 2, aucun sommet n'est marqué ; on dit qu'on réalise un retour en arrière (*backtracking*).

```
def dfs(mat_adj,depart):
    nb_sommets = np.shape(mat_adj)[0]
    pile = [depart]
    sommets_visites = []
    while pile != []:
        # sommet actuel
        sommet = pile.pop()
        if sommet not in sommets_visites:
            sommets_visites.append(sommet)
            # recherche des voisins non visités
            for voisin in range(nb_sommets):
                if mat_adj[sommet,voisin] == 1:
                    pile.append(voisin)
    return sommets_visites
```

Algorithme de parcours en profondeur d'un graphe donné par sa matrice d'adjacence

```
def dfs(liste_adj,depart):
    nb_sommets = len(liste_adj)
    pile = [depart]
    sommets_visites = []
    while pile != []:
        # sommet actuel
        sommet = pile.pop()
        if sommet not in sommets_visites:
            sommets_visites.append(sommet)
            for voisin in liste_adj[sommet]:
                pile.append(voisin)
    return sommets_visites
```

Algorithme de parcours en profondeur d'un graphe donné par sa liste d'adjacence