Algorithmes à connaître en TSI2

Minimum Maximum

Objectif : Déterminer le plus petit (ou le plus grand élément) d'une liste L

Principe mathématique ou algorithmique :

Stocker, par écrasement le cas échéant, la valeur minimale rencontrée en parcourant la liste.

Remarque : min([...]) et max([...]) sont des fonctions intégrées à Python.

Code Python :

```
def minimum(L):
    m=L[0] # initialisation du candidat
    for a in L: # parcours de la liste
        if m>a:
            m=a # écrasement de l'ancien candidat
    return m
```

Somme Produit

Objectif : Calculer $\sum_{k=0}^{n} a_k$ ou $\prod_{k=0}^{n} a_k$

Principe mathématique ou algorithmique :

Stocker, par écrasements successifs, les valeurs des sommes partielles formées en parcourant la liste des $(a_k)_{k\in[0;n]}$.

Remarque : sum(...) est une fonction intégrée à Python. prod(...) est une fonction du module numpy.

Rappel : $Moyenne(\mathrm{L})=\dfrac{1}{len(\mathrm{L})}\sum_{k=0}^{len(\mathrm{L})-1}\mathrm{L}[k]$

$Variance(\mathrm{L})=\left(\dfrac{1}{len(\mathrm{L})}\sum_{k=0}^{len(\mathrm{L})-1}(\mathrm{L}[k])^2\right)-(Moyenne(\mathrm{L}))^2$

Code Python :

```
def somme(L):
    s=0 # initialisation de s
    for a in L: # parcours de L
        s=s+a # écrasement de s
    return s

print(somme([2,3,8,1,7]))
```

```
def produit(L):
    p=1 # initialisation de p
    for a in L: # parcours de L
        p=p*a # écrasement de p
    return p

print(produit([2,3,8,1,7]))
```

Méthode de dichotomie

Objectif : Pour f continue sur $[a;b]$, telle que $f(a)f(b)\leqslant 0$, encadrer aussi précisément que désiré la solution de $f(x)=0$ pour $x\in[a;b]$

Principe mathématique ou algorithmique :

Découper en deux l'intervalle de départ, détecter dans quel sous-intervalle est la solution grâce au théorème des valeurs intermédiaires puis réitérer jusqu'à la précision désirée.

Remarque : la vitesse de convergence est exponentielle, $\mathrm{O}\left(\left(\frac{1}{2}\right)^n\right)$ où n est le nombre d'itérations donc la complexité est $\mathrm{O}(\ln(n))$.

Code Python :

```
def dichotomie(f,a,b,epsilon):
    while b-a>epsilon: # condition  d'arrêt sur la précision requise
        c=(a+b)/2 # c est le milieu de [a;b]
        if f(a)*f(c)<=0: # test du changement de signe des images
            b=c # la solution est dans [a;c] donc on écrase l'ancien b
        else:
            a=c # la solution est dans [c;b] donc on écrase l'ancien a
    return (a,b)

print(dichotomie(lambda x: x**2-2,0,2,0.00001))
```

Recherche dicho-tomique récursive

Objectif : Dans une liste triée, rechercher un terme plus rapidement qu'en les testant tous.

Principe mathématique ou algorithmique :

Si la liste est vide...
Si le terme central est celui recherché...
Sinon on cherche dans la demi-liste susceptible de contenir le terme recherché.

Complexité $\mathrm{O}(\ln(n))$

Code Python :

```
def recherche(L:list,x)->bool:
    n=len(L)
    m=n//2
    if n==0:
        return False
    elif L[m]==x:
        return True
    elif L[m]<x:
        return recherche(L[m+1:],x)
    else :
        return recherche(L[:m],x)
```

Méthode des rectangles (à droite)

Objectif : Approximation numérique de $\int_a^b f(x)\,\mathrm{d}x$

Principe mathématique ou algorithmique :

Sommes de Riemann : pour f continue sur $[a;b]$,

$$\sum_{k=1}^{n} f\left(a+k\frac{b-a}{n}\right)\frac{b-a}{n}\underset{n\to+\infty}{\to}\int_a^b f(x)\,\mathrm{d}x$$

Code Python :

```
def rectangles(f,a,b,N):
    return sum(f(a+k*(b-a)/N)*(b-a)/N for k in range(1,N+1))
print(rectangles(lambda x:x**2,0,1,1000))
```

Méthode des trapèzes

Objectif : Approximation numérique de $\int_a^b f(x)\,\mathrm{d}x$

Principe mathématique ou algorithmique :

Pour f continue sur $[a;b]$,

$$\left(\frac{f(a)}{2}+\sum_{k=1}^{n-1} f\left(a+k\frac{b-a}{n}\right)+\frac{f(b)}{2}\right)\frac{b-a}{n}\underset{n\to+\infty}{\to}\int_a^b f(x)\,\mathrm{d}x$$

Code Python :

```
def trapezes(f,a,b,N):
    return (f(a)/2+sum(f(a+k*(b-a)/N) for k in range(1,N))+f(b)/2)*(b-a)/N
print(trapezes(lambda x:x**2,0,1,1000))
```

Parcours d'un graphe en profondeur

Objectif : À partir d'un sommet suivre les arêtes arbitrairement, en marquant les sommets déjà visités pour ne pas les visiter à nouveau.

Principe mathématique ou algorithmique :

Un graphe orienté peut être manipulé à l'aide de la liste de ses arêtes [(sommet1,sommet2),...] ou d'un dictionnaire {sommet1:[sommets adjacents à sommet1],...}

Ligne 6, le booléen « new in L » peut aussi permettre de détecter la présence d'un cycle : le sous-chemin partant de la première occurrence de « new » jusqu'à la seconde occurrence est un cycle.

Code Python :

```
from numpy.random import choice
def parcours_en_profondeur(d,D:dict)->list:
    L=[d] #initialisation de la liste des sommets créant le chemin
    while D[L[-1]]!=[]: # tant que le dernier sommet n'est pas une impasse
        new=choice(D[L[-1]]) # choix aléatoire du sommet suivant
        if new in L: #si le nouveau sommet est déjà dans le chemin
            return L # alors le chemin est terminé
        else :
            L.append(new) #sinon le chemin se poursuit
    return L

print(parcours_en_profondeur('a',{'h': ['g'], 'a': ['b', 'c'],
'b': ['a', 'd', 'e'], 'e': ['b', 'd', 'f', 'g'], 'f': ['e', 'g'],
'g': ['e', 'f', 'h'], 'd': ['b', 'c', 'e'], 'c': ['a', 'd']}))
```

Parcours d'un graphe en largeur

Objectif : Visiter tous les sommets en « cercle concentriques » autour du sommet de départ jusqu'à une longueur donnée

Principe mathématique ou algorithmique :

Les chemins de longueur n partant du sommet d sont obtenus en reliant à d les chemins de longueur $n-1$ partants de sommets adjacents à d : on procède donc récursivement jusqu'aux chemins de longueur 0.

Remarque : dans un graphe orienté, $\mathrm{M}=\left(\mathrm{int}\left(\text{« }s_i\to s_j\ existe\ ?\text{ »}\right)\right)_{\substack{0\leqslant i\leqslant n-1\\0\leqslant j\leqslant n-1}}$ est la matrice d'adjacence alors $(\mathrm{M}^k)_{i,j}=$ le nombre de chemins de longueur k reliant s_i à s_j

Code Python :

```
def parcours_en_largeur(d,D:dict,n:int)->[list]:
    if n==0:
        return [[d]] #unique chemin partant de d et ayant 0 arête
    return([[d]+C for new in D[d] for C in parcours_en_largeur(new,D,n-1)])
#concaténation du chemin partant de d avec tous les autres chemins de
#longueur n-1 partant des sommets adjacents à d
print(parcours_en_largeur('a',{'h': ['g'], 'a': ['b', 'c'],
'b': ['a', 'd', 'e'], 'e': ['b', 'd', 'f', 'g'], 'f': ['e', 'g'],
'g': ['e', 'f', 'h'], 'd': ['b', 'c', 'e'], 'c': ['a', 'd']},4))
```

Tri par insertion

Objectif : Trier une liste comme un jeu de carte : un paquet trié dans une main, les autres cartes insérées au fur et à mesure au bon endroit.

Principe mathématique ou algorithmique :

$$\mathrm{L}=\left[\overbrace{\mathrm{L}[0],\ldots,\mathrm{L}[k-2],\mathrm{L}[k-1]}^{i\text{ éléments déja triés}},\overbrace{\mathrm{L}[k]}^{=v},\overbrace{\mathrm{L}[k+1],\ldots,\mathrm{L}[n-1]}^{\text{à trier}}\right]$$

décalages éventuels à droite

puis insertion

$$\mathrm{L}'=\left[\underbrace{\mathrm{L}[0],\ldots,v,\ldots,\mathrm{L}[k-1]}_{i+1\text{ éléments triés}},\underbrace{\mathrm{L}[k+1],\ldots,\mathrm{L}[n-1]}_{\text{à trier}}\right]$$

Réitérer la méthode d'insertion sur les termes L[1] à L[n-1] pour trier entièrement la liste L.

Complexité temporelle pour len(L)=n

meilleur des cas : $\mathrm{O}(n)$, pire des cas : $\mathrm{O}(n^2)$

Il faut être capable d'adapter le tri à une autre relation de comparaison ou à une autre structure de données.

Code Python :

```
def tri_insertion(L):
    n=len(L)
    for k in range(1,n): #L[:1] est déjà triée
        v=L[k] #stockage de la valeur à insérer
        i=k-1 # dernier indice de la liste déjà triée L[:k]
        while i>=0 and L[i]>v: # le décalage s'arrête quand L[i]<=v
            L[i+1]=L[i] #décalage des termes vers la droite
            i=i-1 # la liste déjà triée L[:k] est parcourue vers la gauche
        L[i+1]=v #insertion à la dernière position pour laquelle  L[i]>v

L1=[6,2,4,8,3,8,4,5]
tri_insertion(L1)
print(L1)
```

<u>Rappel sur la médiane d'une série statistique</u> : soit L une liste triée et n=len(L)

Si n est pair $\underbrace{\mathrm{L}[0]\leqslant\ldots\leqslant\mathrm{L}\left[\frac{n}{2}-1\right]}_{\frac{n}{2}\text{ termes}}\leqslant\underbrace{\mathrm{L}\left[\frac{n}{2}\right]\leqslant\ldots\leqslant\mathrm{L}[n-1]}_{\frac{n}{2}\text{ termes}}$

$$\text{Médiane(L)}=\frac{\mathrm{L}\left[\frac{n}{2}-1\right]+\mathrm{L}\left[\frac{n}{2}\right]}{2}$$

Si n est impair $\underbrace{\mathrm{L}[0]\leqslant\ldots\leqslant\mathrm{L}\left[\frac{n-1}{2}-1\right]}_{\frac{n-1}{2}\text{ termes}}\leqslant\mathrm{L}\left[\frac{n-1}{2}\right]\leqslant\underbrace{\mathrm{L}\left[\frac{n-1}{2}+1\right]\leqslant\ldots\leqslant\mathrm{L}[n-1]}_{\frac{n-1}{2}\text{ termes}}$

$$\text{Médiane(L)}=\mathrm{L}\left[\frac{n-1}{2}\right]$$

```
def mediane(L) :
    L.sort()
    n=len(L)
    if n%2==0:
        return (L[n/2-1]+L[n/2])/2
    return L[(n-1)/2]
```

<table>
<tr>
<td>Méthode du Pivot de Gauss</td>
<td>Échelonner et réduire la matrice augmentée pour résoudre un système linéaire à p inconnues</td>
<td>Algorithme de Gauss avec recherche partielle du pivot :
r = 0 # nombre de pivots rencontrés i.e. index de la ligne du pivot courant
Pour la colonne j (j allant de 1 à p) :
si l'un des termes de la colonne j, sur les lignes d'indice strictement supérieur à r, est non nul alors :
i) trouver l'indice i_0 du terme de la colonne j, sur les lignes d'indice strictement supérieur à r, le plus grand en valeur absolue : ce terme sera le nouveau pivot
ii) incrémenter r d'une unité
iii) permuter les lignes d'indice r et i_0
iv) multiplier la ligne d'indice r par l'inverse du nouveau pivot
v) annuler tous les autres termes de la colonne j en utilisant une combinaison linéaire judicieuse avec la ligne d'indice r
Si le système est compatible (a) et contient autant de pivots que de colonnes (b) alors l'unique solution est le contenu, à l'issue de l'algorithme, des r premières lignes de la colonne contenant le second membre.

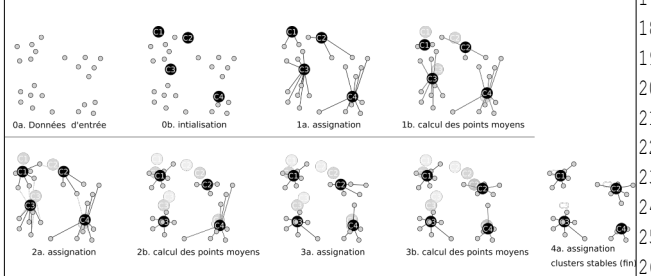
Remarque : avec le module sympy, M.rref() donne la matrice échelonnée réduite équivalente en lignes à M du type Matrix().</td>
<td>

```python
from numpy import array

def test(L:[array],j:int,r:int)->bool:
    for i in range(r+1,len(L)):
        if L[i][j]!=0:
            return True
    return False

def indice(L:[array],j:int,r:int)->bool:
    i0=r+1
    M=abs(L[i0][j])
    for i in range(r+2,len(L)):
        if abs(L[i][j])>M:
            i0=i
            M=abs(L[i][j])
    return i0

def Gauss(L:[array])->list:
    r=-1 # le premier pivot sera sur la ligen d'index 0
    for j in range(len(L[0])-1): # attention la dernière composante des lignes contient le second membre
        if test(L,j,r): # y a-t-il un pivot dans la clonne j sous la ligne r
            i0=indice(L,j,r) # indice de la ligne contenant le plus grand pivot en valeur absolue
            r=r+1 # incrémentation du compteur de pivots
            L[r],L[i0]=L[i0],L[r] # permutation des lignes d'index r et i0
            L[r]=L[r]/L[r][j] # pivot transformé en 1
            for i in range(len(L)): # opérations sur toutes les lignes
                if i!=r: # sauf la ligne d'index r
                    L[i]=L[i]-L[i][j]*L[r] # valide comme opération sur des arrays
    assert(all(L[i][-1]==0 for i in range(r+1,len(L)))) # les conditions de compatibilité sont-elles vérifiées
    assert(r==len(L[0])-2) # y a-t-il autant de pivot que de colonnes (r+1 pivots et len(L[0])-1 colonnes)
    return([L[i][-1] for i in range(r+1)]) # la colonne augmentée contient les solutions
```

</td>
</tr>
<tr>
<td>k_moyennes</td>
<td>Déterminer k clusters en minimisant la somme de leur moment d'inertie</td>
<td>Initialiser les k clusters
Tant que les clusters ne sont pas stabilisés :
Créer la liste des barycentres des k clusters
Créer k nouveaux clusters en associant chaque point au barycentre le plus proche de lui
 
</td>
<td>

```python
import numpy as np
from numpy import array

def barycentre(L:[array])->array:
    return sum(P for P in L)/len(L)

def centre(P:[array],LB:[array])->int:
    d=np.linalg.norm(P-LB[0])
    indice=0
    for k in range(1,len(LB)):
        if d>np.linalg.norm(P-LB[k]):
            d=np.linalg.norm(P-LB[k])
            indice=k
    return indice

def listes_differentes(L1:[array],L2:[array])->bool:
    for i in range(len(L1)):
        if not(all(L1[i]==L2[i])):
                return True
    return False

def k_moyennes(LP:[array],k:int)->{int:[array]}:
    DP={i:[] for i in range(k)} # initialisation du dictionaire {indice du cluster : liste des points}
    for P in LP:
        DP[np.random.randint(k)].append(P) # initialisation aléatoire des clusters
    LB=[barycentre(DP[i]) for i in range(k)] # initialisation de la liste des barycentres
    test=True
    while test:
        new_DP={i:[] for i in range(k)}
        for P in LP:
            new_DP[centre(P,LB)].append(P) # nouveau dictionnaire des clusters centrés sur les barycentres
        new_LB=[barycentre(new_DP[i]) for i in range(k)] # nouvelle liste des barycentres
        test=listes_differentes(LB,new_LB)
        DP=new_DP
        LB=new_LB
    return DP
```

</td>
</tr>
<tr>
<td>Interpolation polynomiale de Lagrange</td>
<td>Déterminer l'unique courbe représentative d'une fonction polynomiale de degré inférieur ou égal n passant par n+1 points fixés.</td>
<td>Soient n+1 réels $(x_i)_{0\leqslant i\leqslant n}$ distincts deux à deux, la base de Lagrange de $\mathbb{R}_n[X]$ aux abscisses $(x_i)_{0\leqslant i\leqslant n}$ est $(L_i(X))_{0\leqslant i\leqslant n}$ avec $L_i(X)=\prod_{\substack{0\leqslant j\leqslant n\\ j\neq i}}\frac{X-x_j}{x_i-x_j}$

Alors, $\forall P\in\mathbb{R}_n[X]$, $P(X)=P(x_i)L_i(X)$</td>
<td>

```python
from numpy.polynomial import Polynomial

def test(Lx:[float])->bool:
    for i in range(len(Lx)):
        for j in range(i+1,len(Lx)):
            if Lx[i]==Lx[j] :
                return False
    return True

def L(i:int,Lx:[float])->Polynomial:
    assert 0<=i<=len(Lx)-1
    assert test(Lx)
    Produit=Polynomial([1])
    for j in range(len(Lx)):
        if j!=i:
            Produit=Produit*Polynomial([-Lx[j],1])/(Lx[i]-Lx[j])
    return Produit

def P(Lx:[float],Ly:[float])->Polynomial:
    assert test(Lx)
    assert len(Lx)==len(Ly)
    Somme=Polynomial([0])
    for j in range(len(Lx)):
        Somme=Somme+Ly[j]*L(j,Lx)
    return Somme
```

</td>
</tr>
</table>

<table>
<tr>
<td>Méthode d'Euler</td>
<td>Approximation numérique de la fonction solution d'un problème de Cauchy d'ordre 1</td>
<td>Formule de Taylor à l'ordre 1 :
$$f(x_n+\varepsilon) \underset{\varepsilon\to 0}{=} f(x_n)+\varepsilon f'(x_n)+o(\varepsilon)$$
Pour $\varepsilon = x_{n+1}-x_n$ et $f(x_n)=y_n$ on pose :
$$y_{n+1}=y_n+\varepsilon f'(x_n)$$
Or $f'(x_n)$ s'exprime, grâce à l'équation différentielle, en fonction de x_n et de $f(x_n)=y_n$.
Remarque : odeint( ) du module scipy.integrate utilise ce procédé.</td>
<td>

```
def Euler(y_prime_de_x,x0,y0,epsilon,xmax):
    X,Y=[x0],[y0] # X et Y stockeront les valeurs de xn et yn
    while X[-1]<xmax:
        Y.append(Y[-1]+epsilon*y_prime_de_x(X[-1],Y[-1]))
        X.append(X[-1]+epsilon)
    return X,Y

# application pour y'=-2xy+3x et y(0)=1
sol=Euler(lambda x,y:-2*x*y+3*x,0,1,0.0001,2)
import matplotlib.pyplot as plt
plt.plot(sol[0],sol[1])
plt.show()
```

</td>
</tr>
<tr>
<td>Méthode d'Euler à l'ordre n</td>
<td>Approximation numérique de la fonction solution d'un problème de Cauchy scalaire linéaire d'ordre n</td>
<td>Il s'agit de transformer une équation différentielle linéaire d'ordre n en un système différentiel linéaire d'ordre 1
$$\text{Pour } Y(x)=\begin{pmatrix} y(x) \\ y'(x) \\ \vdots \\ y^{(n-1)}(x) \end{pmatrix} \in M_{n,1}(\mathbb{R}),$$
$$B(x)=\begin{pmatrix} 0 \\ \vdots \\ 0 \\ b(x) \end{pmatrix} \in M_{n,1}(\mathbb{R})$$
$$\text{et } A(x)=\begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 0 & 1 \\ -a_0(x) & \dots & \dots & \dots & -a_{n-1}(x) \end{pmatrix} \in M_n(\mathbb{R})$$
on a :
$$y^{(n)}+a_{n-1}(x)y^{(n-1)}+\dots+a_0(x)y=b(x)$$
$$\Leftrightarrow \quad Y'=A(x)Y+B(x)$$</td>
<td>

```
import numpy as np

def Euler_ordre_n(coef,second_membre,Y0,x0,pas,xmax):
    n=len(coef(x0))
    A=np.zeros([n,n]) # matrice carrée de taille n
    B=np.zeros([n,1]) # matrice colonne
    Y=np.array([[Y0[j]] for j in range(n)]) # matrice des conditions initiales
    for i in range(n-1):
        A[i,i+1]=1 # des 1 au dessus de la diagonale
    Lx=[x0]
    Ly=[Y[0,0]] # seule la première composante de Y est utile
    while Lx[-1]<xmax:
        A[n-1,:]=-1*np.array(coef(Lx[-1]))#la dernière ligne peut dépendre de x
        B[n-1,0]=second_membre(Lx[-1])
        Y=Y+pas*(np.dot(A,Y)+B) # opérations sur des arrays cf matrices
        Ly.append(Y[0,0]) #seule la première composante de Y est stockée
        Lx.append(Lx[-1]+pas)
    return Lx,Ly

import matplotlib.pyplot as plt
# application pour y''+y'=0 et y(0)=1 et y'(0)=0
sol=Euler_ordre_n(lambda x: [1,0],lambda x:0,[1,0],0,0.01,10)
plt.plot(sol[0],sol[1])
```

</td>
</tr>
</table>