T.P.E D'INFORMATIQUE 4

Tris et Fractales

Liste des participants :

- DJAHAPPI NZEMBIA ELISEE (Chef de groupe)
- DIFFOUO TIYO NESLY
- BILA ELIE DESIRE
- DINGUE WANDJI SERENA
- DZUGAING FOTSI EIFFEL
- DJANEA DOOH DJEMBA

Sous la supervision de :

Dr TAGOUDJEU

Mme WAMBA

Année 2023-2024

- ***<u>Nota sur le contenu de ce travail</u>:***

Dans ce compte rendu de notre travail portant sur l'implémentation en python de quelques algorithmes de tris et la représentation graphique de fractales, nous avons présenté : ***les tris bulles, insertion, rapide, fusion et insertion dichotomique*** ainsi que ***quelques cas de fractales de MANDELBROT, de JULIA et de SIERPINSKI.***

En ce qui concerne les algorithmes de tris, étant donné que les principes avaient déjà été très bien détaillés dans l'énoncé du T.P.E, nous nous sommes contentés ici d'en rappeler les points clés. Quant aux fractales, nous proposons quelques codes `python` pour leurs représentations. Nous avons à cet effet fait usage de la bibliothèque `Matplotlib`.

Document saisi par **DJAHAPPI**.

• • • Principe d'action

> Comme son nom le suggère, le tri à bulles consiste à faire *remonter* les éléments dans une liste à la manière des bulles d'air qui remonteraient à la surface d'un liquide. Son principe est donc de faire descendre, au fur et à mesure, les éléments les plus grands. On parcours donc $n-1$ fois la liste de taille n en plaçant au fond le plus grand parmi ceux qui n'ont pas encore été triés.

• • • Algorithme

```
procedure TRI_A_BULLES (var T : vecteur, n: entier);
var arret : booleen; i, j: entier; aux : reel;
debut
    Pour i = n(−1)2 faire
        arret ⟵ faux; j ⟵ 1;
        tantQue non arret faire
            si j < i et T[j] > T[j + 1]alors
                aux ⟵ T[j];    T[j] ⟵ T[j + 1];    T[j + 1] ⟵ aux;
            finsi
            j ⟵ j + 1;
            si j = i alors
                arret ⟵vrai;
            finsi
        fintantQue
    finPour
    STOP
```

• • • Implémentation en python

```python
def TRI_A_BULLES(T : list) -> list :
    n = len(T)
    for i in range(n-1,0,-1) :
        # debut de la descente de la bulle
        # l'indice de la bulle est d'abord j=0
        arret = False
        j = 0

        while not arret :
            if j < i and T[j] > T[j+1] :
                T[j], T[j+1] = T[j+1], T[j]
            # l'indice de la bulle est maintenant j = j+1
            j = j + 1
            if j == i :
                arret = True
    return T
```

TRI INSERTION

• • • Principe d'action

Le principe ici de parcourir la liste et d'insérer chaque élément de façon progressive à sa place dans la partie déjà triée du tableau.

• • • Algorithme

procedure TRI_INSERTION (**var** T : **vecteur**, n : **entier**);
var i, j : **entier**; val : **reel** ;
debut
  Pour $i = 2$ (1) n **faire**
    / le tableau est deja trie de 1 a $i-1$ */*
    / on insere alors $T[i]$ a sa place dans cette partie deja triee */*

    $j \longleftarrow i-1$; val $\longleftarrow T[i]$;
    tantQue $j \geqslant 1$ **faire**
      si $T[j] >$ val **alors**
        $T[j+1] \longleftarrow T[j]$;
        $T[j] \longleftarrow$ val ;
      finsi
      $j \longleftarrow j-1$;
    fintantQue
  finPour
STOP

• • • Implémentation en python

```python
def TRI_INSERTION(T : list) -> list :
    n = len(T)
    for i in range(1,n) :
        j = i-1
        val = T[i]
        while j >= 0 :
            if T[j] > val :
                T[j+1] = T[j]
                T[j] = val
            j = j-1
    return T
```

• • • Principe d'action

En ce qui concerne le tri rapide, on trie de façon récursive le tableau et ce, en triant les sous-listes de gauche et de droite d'un pivot qui est un élément du tableau. La sous-liste de gauche contenant les éléments de la liste qui sont inférieurs au pivot et la liste de droite, ceux qui lui sont supérieurs.
•→ Nous utiliserons une procédure auxiliare qui se chargera de la recursion et d'une procédure principale qui se contente d'initialiser la recursion et d'en restituer le résultat final.

• • • Algorithme

```
procedure permuter (var x : reel , y: reel) ;
var aux : reel ;
debut
    aux ⟵ x;    x ⟵ y;    y ⟵ aux ;
STOP
```

```
procedure TRI_RAPIDE (var T : vecteur, n : entier);
debut
    TRI_RAPIDE_rec (T, 1, n) ;
    /* fin algo */
STOP
```

```
procedure TRI_RAPIDE_rec (var T : vecteur, Idebut : entier, Ifin : entier);
/* Objectifs : Cette procedure effectue le trie rapide du vectuer T
               entre les indices Idebut et Ifin */
var Ipivot, i, j, k : entier ; aux, pivot : reel ;
debut
    Ipivot ⟵ (Idebut + Ifin)/2 ;
    pivot ⟵ T[Ipivot] ;

    si Idebut ⩾ Ifin alors
        /* fin algo */
    finsi
    si Idebut = Ifin alors
        si T[Idebut] > T[Ifin] alors
            permuter (T[Idebut], T[Ifin]);
        finsi
        /* fin algo */
    finsi
    /* Cas general */
    aux ⟵ Ipivot ;
    Pour i = ipivot − 1 (−1) Idebut faire
        si T[i] > pivot alors
            Pour k = i (1) Ipivot − 1 faire
                permuter(T[k], T[k + 1]) ;
            finPour
            Ipivot ⟵ Ipivot − 1 ;
        finsi
    finPour

    Pour j = aux +1 (−1) Ifin faire
    si T[j] < pivot alors
        Pour k = j (−1) Ipivot + 1 faire
            permuter(T[k − 1], T[k]) ;
        finPour
```

$Ipivot \longleftarrow Ipivot + 1$;
<u>finsi</u>
<u>finPour</u>

TRI_RAPIDE_rec $(T, Idebut, Ipivot - 1)$;
TRI_RAPIDE_rec $(T, Ipivot + 1, Ifin)$;
<u>**STOP**</u>

• • • Implémentation en python

```python
def TRI_RAPIDE_rec(T : list, Idebut : int, Ifin : int) -> list:
    Ipivot = (Idebut + Ifin) // 2
    pivot = T[Ipivot]

    if Idebut > Ifin :
        return T
    elif Idebut == Ifin  :
        return T

    elif Ifin == Idebut + 1:
        if T[Idebut] > T[Ifin] :
            T[Idebut], T[Ifin] = T[Ifin], T[Idebut]
            return T
        else :
            return T

    aux = Ipivot
    for i in range(Ipivot-1,Idebut-1,-1) :
        if T[i] > pivot :
            for k in range(i, Ipivot) :
                T[k], T[k+1] = T[k+1], T[k]
            Ipivot -= 1

    for j in range(aux+1, Ifin+1) :
        if T[j] < pivot :
            for k in range(j, Ipivot,-1) :
                T[k-1], T[k] = T[k], T[k-1]
            Ipivot += 1

    T = TRI_RAPIDE_rec(T, Idebut, Ipivot-1)
    T = TRI_RAPIDE_rec(T, Ipivot+1, Ifin)
    return T

def TRI_RAPIDE(T : list) -> list :
    T = TRI_RAPIDE_rec(T, 0, len(T)-1)
    return T
```

COMPLEXITE DU TRI RAPIDE

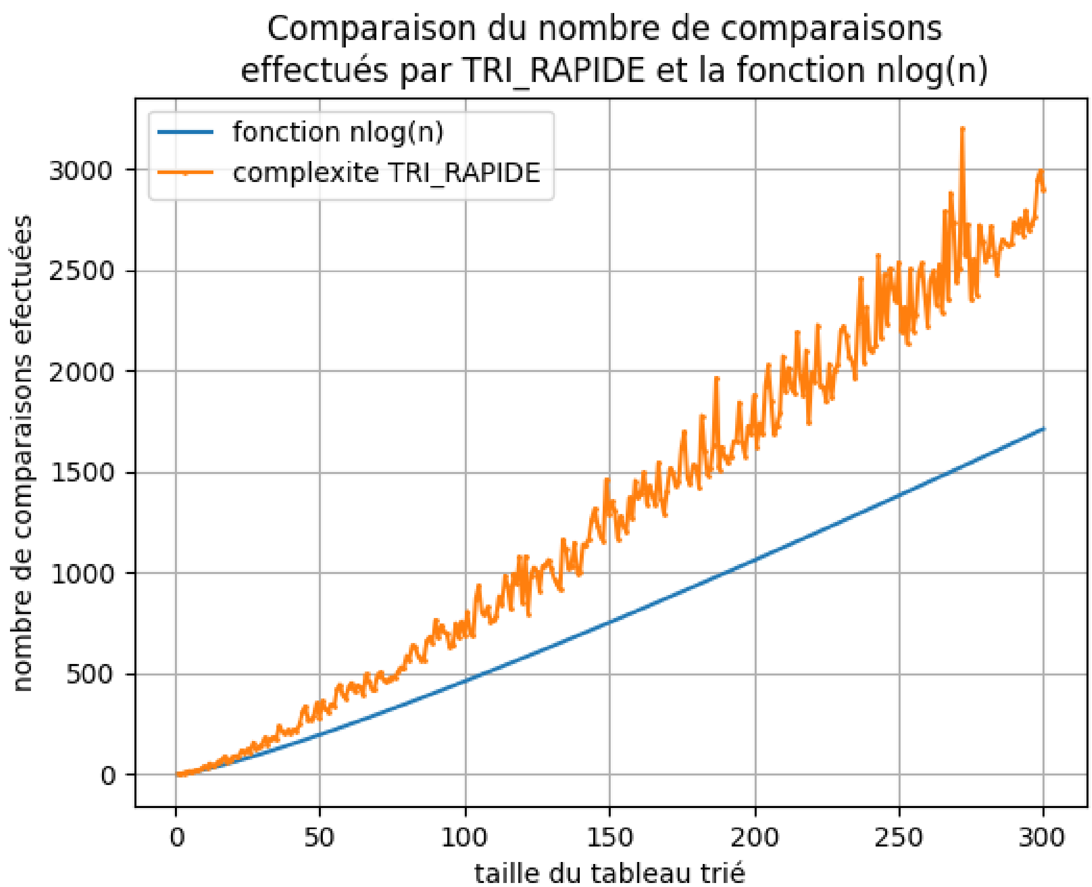
• • • Principe d'action

Il s'agit d'évaluer le nombre de comparaisons effectuées par notre TRI RAPIDE. Nous allons modifier notre code du tri rapide ci-dessus en ceci :

1. Nous allons passer en paramètre le nombre de comparaisons qui sera modifié à chaque fois;
2. Ainsi, à chaque fois qu'une comparasion veut être faite, on **incrémente** le nombre de comparaisons;
3. La fonction renverra au final, uniquement le nombre de comparasions .

Ensuite, nous allons faire une représentation graphique des courbes de la fonction $n\log(n)$ et celle du nombre de comparaisons en fonction de la taille du tableau.

► **N.B :** Nous avons pris des tableaux randomisés dont les éléments sont des entiers $\in [0,\ 100\,000]$
Voici un exemple de ce que l'obtient :



Le code est à la page suivante.

▶ **Modification des procédures**

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

def TRI_RAPIDE_rec(nb_compa : int, T : list, Idebut : int, Ifin : int) :
    Ipivot = (Idebut + Ifin) // 2
    pivot = T[Ipivot]

    nb_compa += 1
    if Idebut > Ifin :
        return T, nb_compa
    elif Idebut == Ifin  :
        return T, nb_compa
    elif Ifin == Idebut + 1:
        nb_compa += 1
        if T[Idebut] > T[Ifin] :
            T[Idebut], T[Ifin] = T[Ifin], T[Idebut]
            return T, nb_compa
        else :
            return T, nb_compa

    aux = Ipivot
    for i in range(Ipivot-1,Idebut-1,-1) :
        nb_compa += 1
        if T[i] > pivot :
            for k in range(i, Ipivot) :
                T[k], T[k+1] = T[k+1], T[k]
            Ipivot -= 1

    for j in range(aux+1, Ifin+1) :
        nb_compa += 1
        if T[j] < pivot :
            for k in range(j, Ipivot,-1) :
                T[k-1], T[k] = T[k], T[k-1]
            Ipivot += 1

    T, nb_compa = TRI_RAPIDE_rec(nb_compa,T, Idebut, Ipivot-1)
    T, nb_compa= TRI_RAPIDE_rec(nb_compa, T, Ipivot+1, Ifin)
    return T, nb_compa

def TRI_RAPIDE_complexite(Tab : list) :
    Tab, n = TRI_RAPIDE_rec(nb_compa=0, T= Tab, Idebut=0, Ifin=len(Tab)-1)
    return n
```

▶ Procédures principales de tracé

```python
def complexite(nb_max_taille) -> tuple:
    X = np.zeros(nb_max_taille)
    Y1 = np.zeros(nb_max_taille)

    for n in range(1, nb_max_taille + 1) :
        X[n-1] = n
        T = np.random.randint(100000, size=n)
        Y1[n-1]= TRI_RAPIDE_complexite(T)
    return X, Y1

def compare_complexity(n : int) :
    X, Y1= complexite(n)
    Y3 = np.zeros(n)
    for i in range(1,n+1) :
        Y3[i-1] = i*np.log(i)

    plt.xlabel('taille du tableau trié')
    plt.ylabel('nombre de comparaisons efectuées')
    plt.title('Nombre de comparaisons dans TRI_RAPIDE et la fonction nlog(n)')
    plt.plot(X, Y3, label='fonction nlog(n)')
    plt.plot(X, Y1, marker='o', markersize=1, label='complexite TRI_RAPIDE')
    plt.legend()
    plt.grid()
    plt.show()
```

TRI FUSION

• • • Principe d'action

Le principe du tri fusion consiste à diviser de façon récursive le tableau en deux parties, de trier ces deux parties et de les fusionner en conservant l'ordre croissant des éléments.
•→ Nous utiliserons une procédure de fusion des deux parties triées du tableau, une procédure qui sera récursive et dont le rôle est de diviser le tableau en deux parties puis les fusionner. Enfin, nous initialiserons ce processus par une procédure principale.

• • • Algorithme

```
procedure TRI_FUSION (var T : vecteur, n : entier);
debut
    si n < 2 alors
        /* fin algo */
    finsi
    TRI_FUSION_rec(T, 1, n) ;
    /* fin algo */
STOP
```

```
procedure TRI_FUSION_rec (var T : vecteur, Idebut: entier, Ifin : entier) ;
var Imil : entier ;
debut
    si Ifin = Idebut ou Ifin = Idebut + 1 alors
        FUSION(T, Idebut, Idebut, Ifin) ;
    finsi
    Imil ⟵ (Idebut + Ifin)//2 ;
    TRI_FUSION_rec(T, Idebut, Imil) ;
    TRI_FUSION_rec(T, Imil + 1, Ifin) ;
    FUSION(T, Idebut, Imil, Ifin) ;
STOP
```

```
procedure FUSION(var T : vecteur, Idebut : entier, Imil : entier, Ifin : entier);
/* Le tableau T est trie de Idebut a Imil et de Imil + 1 a Ifin */

var i, j, k, ℓ : entier ;      S : tableau[1 .. Ifin − Idebut + 1] de reel;
debut
i ⟵ Idebut ;     j ⟵ Imil + 1 ;     k ⟵ 1 ;
tantQue (i ⩽ Imil et j ⩽ Ifin) faire
    si T[i] > T[j] alors
        S[k] ⟵ T[j] ;
        j ⟵ j + 1 ;
    sinon
        S[k] ⟵ T[i] ;
        i ⟵ i + 1 ;
    finsi
    k ⟵ k + 1 ;
fintantQue
si i > Imil alors
    Pour ℓ = j (1) Ifin faire
        S[k] ⟵ T[ℓ] ;
        k ⟵ k + 1 ;
    finPour
finsi
si j > Ifin alors
    Pour ℓ = i (1) Imil faire
```

$S[k] \longleftarrow T[\ell]$;
$k \longleftarrow k+1$;
<u>finPour</u>
<u>finsi</u>
<u>Pour</u> $\ell = Idebut\ (1)\ Ifin$ **<u>faire</u>**
$T[\ell] \longleftarrow S[\ell - Idebut + 1]$
<u>finPour</u>
<u>STOP</u>

• • • Implémentation en python

```python
def FUSION (T : list, Idebut : int, Imil : int, Ifin : int) -> list :
    S = []
    i = Idebut
    j = Imil + 1

    while i <= Imil and j <= Ifin :
        if T[i] > T[j] :
            S.append(T[j])
            j += 1
        else:
            S.append(T[i])
            i += 1


if i > Imil :
    for l in range(j, Ifin + 1) :
        S.append(T[l])
if j > Ifin :
    for l in range(i, Imil + 1) :
        S.append(T[l])

for l in range(Idebut, Ifin + 1) :
    T[l] = S[l - Idebut]
return T

def TRI_FUSION_rec(T : list, Idebut : int, Ifin : int) -> list :
    if Ifin == Idebut + 1 or Ifin == Idebut:
        T = FUSION(T, Idebut, Idebut, Ifin)
        return T

    Imil = (Idebut + Ifin) // 2
    T = TRI_FUSION_rec(T, Idebut, Imil)
    T = TRI_FUSION_rec(T, Imil + 1, Ifin)
    T = FUSION(T, Idebut, Imil, Ifin)
    return T

def TRI_FUSION(T : list) :
    n = len(T)
    if n < 2 :
        return T
    T = TRI_FUSION_rec(T, 0, n-1)
    return T
```

TRI INSERTION DICHOTOMIQUE

• • • Principe d'action

Le principe est le même que celui du tri insertion classique à savoir: parcourir la liste et d'insérer chaque élément de façon progressive à sa place dans la partie déjà triée du tableau.
Seulement, le tri par insertion dichotomique se démarque au moment de l'insertion de chaque élément dans la liste déjà triée. En effet, on cherche par dichotomie la position que le nouvel élément doit occuper dans la liste déjà triée.

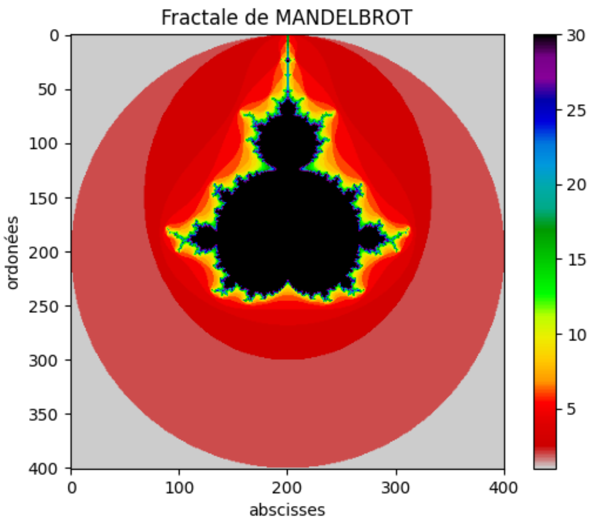
• • • Algorithme

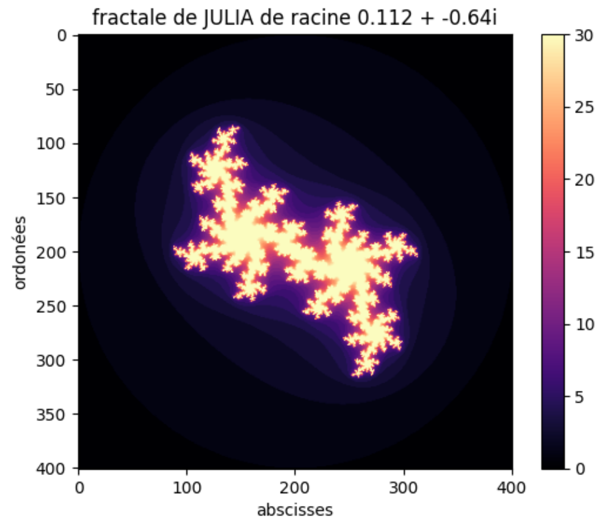
```
procedure inserer (var T : vecteur , newInd : entier, oldInd : entier) ;
    /* Objectifs : Inserer l'element T[oldInd] a la position d'indice newInd
                   avec oldInd > newInd */
    var i : entier,    aux : reel ;
    debut
        Pour i = oldInd (−1) newInd + 1 faire
            aux ⟵ T[i];    T[i] ⟵ T[i − 1] ;    T[i − 1] ⟵ aux ;
        finPour
    STOP
```

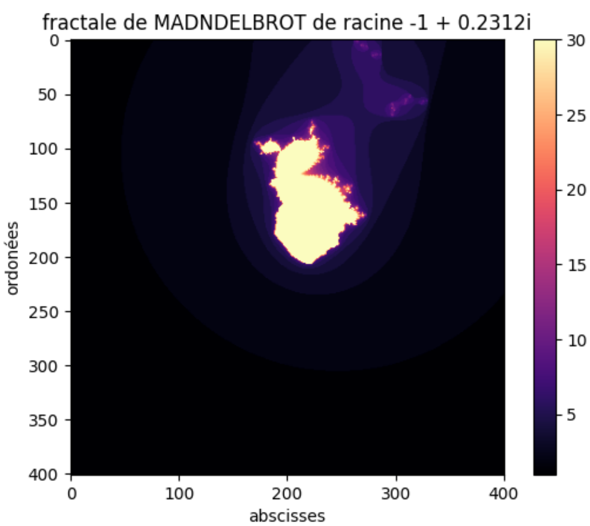
```
procedure TRI_INSERTION_DICHO(var T : vecteur, n : entier);
var i, inG, mil, inD : entier; flag : booleen ;
debut
    Pour i = 2 (1) n faire
        /* le tableau est deja trie de 1 a i − 1 */
        /* on insere alors T[i] a sa place dans cette partie deja triee */
        inG ⟵ 1 ;    inD ⟵ i − 1 ;
        flag ⟵ faux ;

        tantQue inG ⩽ inD faire
            mil ⟵ (inD + inG)//2 ;
            si T[i] = T[mil] alors
                inserer(T, mil, i) ;
            sinon si T[i] < T[mil]
                inD ⟵ mil − 1 ;
            sinon
                inG ⟵ mil + 1 ;
            finsi
        fintantQue

        si flag = faux alors
            si T[i] < T[mil] alors
                inserer(T, mil, i)
            sinon /*T[i] > T[mil]*/
                inserer(T, mil + 1, i)
            finsi
        finsi
    finPour
STOP
```

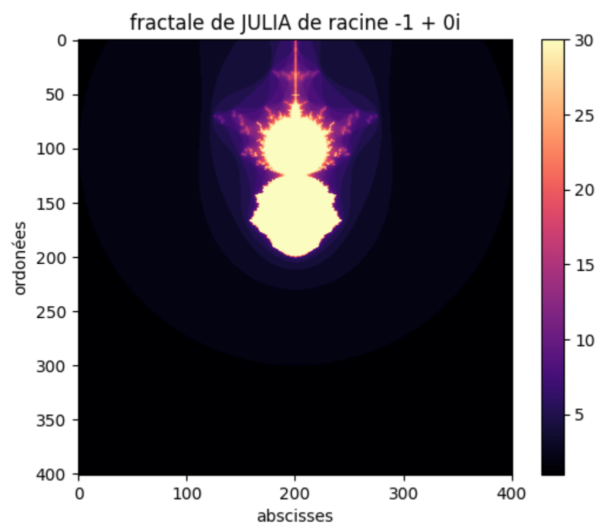
• • • Implémentation en python

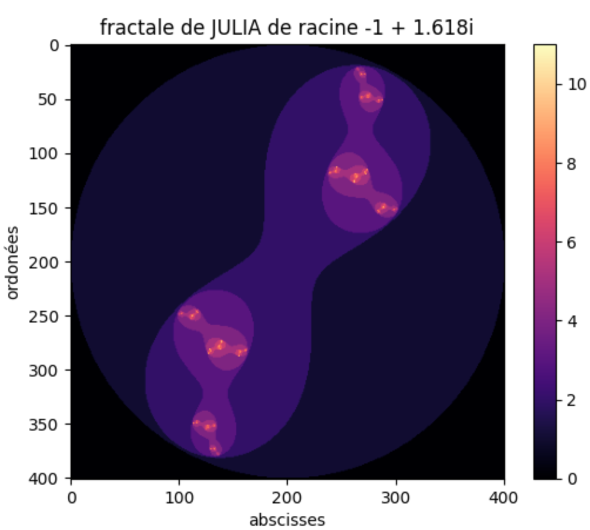
```python
def inserer(T : list, newInd : int, oldInd : int) -> list :
"""
On veut inserer l'element T[oldInd] a la position d'indice
newInd avec oldInd > newInd
"""
for i in range(oldInd, newInd, -1) :
    T[i], T[i-1] = T[i-1], T[i]
return T

def TRI_INSERTION_DICHO(T : list) -> list :
    n = len(T)
    if n < 2 :
        return T
    for i in range(1,n) :
        inG = 0
        inD = i-1
        flag = False
        while inG <= inD :
            mil = (inG + inD)//2
            if T[i] == T[mil] :
                T = inserer(T, mil, i)
                flag = True
            elif T[i] < T[mil] :
                inD = mil - 1
            else :
                inG = mil + 1
        if flag == False :
            if T[i] < T[mil] :
                T = inserer(T, mil, i)
            else : # T[i] > T[mil]
                T = inserer(T, mil + 1, i)
    return T
```

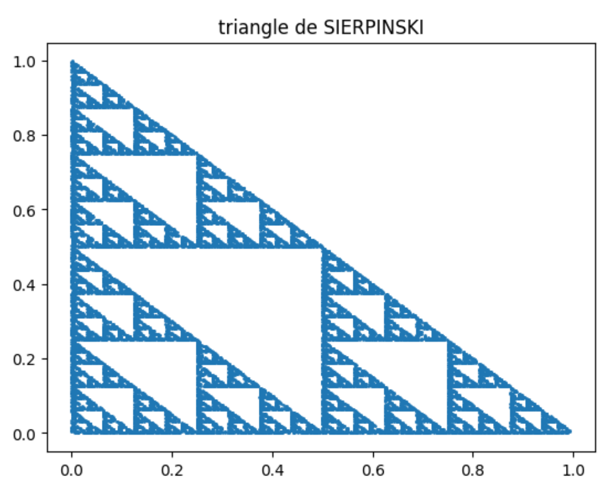

Fractale de MANDELBROT
ordonées
abscisses


fractale de JULIA de racine 0.112 + -0.64i
ordonées
abscisses


fractale de MADNDELBROT de racine -1 + 0.2312i
ordonées
abscisses


fractale de JULIA de racine -1 + 0i
ordonées
abscisses


fractale de JULIA de racine -1 + 1.618i
ordonées
abscisses


triangle de SIERPINSKI

• • • Principe général

On affiche à l'écran un certain nombre de points du plans définis par un procédé soit de rcurrence soit de convergence d'une certaine suite.
Nous illustrerons ci-dessous trois types de fractales bien connues : ***les fractales de Mandelbrot, de Julia et de Sierpinski.***

• • • Fractales de Mandelbrot

On affiche une matrice ***Mat*** à 2 dimensions et dont les éléments sont des indices $n(z)$ des suites $(u_n(z))$ définies par : $\boldsymbol{u_0 = 0}$ et $\boldsymbol{\forall n \geqslant 0,\ u_{n+1}(z) = u_n^2(z) + z}$ pour $\boldsymbol{z = x + iy}$ avec $\boldsymbol{x, y \in [-2,\ 2]}$.
Chaque $n(z)$ signifiant la potentielle convergence de la suite $(u_n(z))$ de la façon suivante :
On fixe un entier $\mathbf{N_{max}}$, un réel **seuil** d'avance à ne pas dépasser et des pas $\mathrm{d}x$ et $\mathrm{d}y$ pour x et y tels que $z = x + iy$.
Et on suit les étapes ci-dessous :

1. On calcule pour chaque $z = x + iy$, on calcule les termes $u_1(z),\ u_2(z), \ldots, u_{N_{max}}(z)$ tant que leurs modules ne dépassent pas le **seuil**.
2. Si le **seuil** est dépassé à un indice n, alors on considère que la suite diverge et on ajoute à la matrice ***Mat*** la valeur n. Dans le cas contraire, on suppose que la suite converge et on ajoute N_{max} à la matrice.
3. On avance x et y : $x = x{+}\mathrm{d}x$ et $y = y{+}\mathrm{d}y$. et on reprends le raisonnement jusqu'à avoir parcouru tout $[-2,\ 2] \times [-2,\ 2]$.

Le code python est le suivant :

```python
import numpy as np
import matplotlib.pyplot as plt
def mandelbrot() :
    Nmax = 30
    seuil = 2
    min = -2
    max = 2
    dx = 0.01
    dy = 0.01

    nb_lignes = int(1 + (max-min) / dx)
    nb_cols = int(1 + (max-min) / dy)

    X = np.arange(min,max+dx,dx)
    Y = np.arange(min,max+dy,dy)
    Mat = np.zeros((nb_lignes, nb_cols))

    for i, x in enumerate(X) :
        for j, y in enumerate(Y) :
            z = complex(x,y)
            u = 0
            n = 0
            while abs(u) < seuil and n < Nmax :
                n += 1
                u = u**2 + z
            Mat[i,j] = n

    plt.xlabel('abscisses')
    plt.ylabel('ordonées')
    plt.title("Fractale de MANDELBROT")
    plt.imshow(Mat, cmap='viridis')
    plt.colorbar()
    plt.show()
```

• • • Fractale de Julia

Le principe est le même que celui de Mandelbrot sauf que la suite est définie par :
$u_0 = c$ et $\forall n \geqslant 0,\ u_{n+1}(z) = u_n^2(z) + z$ pour $z = x + iy$ avec $x, y \in [-2,\ 2]$.
où **c** est un nombre complexe donné que nous appelerons ***racine de la fractale.***

Le code python est le suivant :

```python
import numpy as np
import matplotlib.pyplot as plt
def julia(re, im)  :
    Nmax = 30
    seuil = 2
    min = -2
    max = 2
    dx = 0.01
    dy = 0.01

    nb_lignes = int(1 + (max-min) / dx)
    nb_cols = int(1 + (max-min) / dy)

    X = np.arange(min,max+dx,dx)
    Y = np.arange(min,max+dy,dy)
    Mat = np.zeros((nb_lignes, nb_cols))

    for i, x in enumerate(X) :
        for j, y in enumerate(Y) :
            c = complex(re, im)
            u = complex(x,y)
            n = 0
            while abs(u) < seuil and n < Nmax :
                n += 1
                u = u**2 + c
            Mat[i,j] = n
    plt.xlabel('abscisses')
    plt.ylabel('ordonées')
    plt.title(f'fractale de JULIA de racine {re} + {im}i')
    plt.imshow(Mat, cmap='magma')
    plt.colorbar()
    plt.show()
```

• • • Fractale/ Triangle de Sierpinski

Cette fois-ci nous n'allons plus nous baser sur une construction analytique mais plutôt sur une construction géométrique du triangle de sierpinski. SOit donc trois points du plan : $A(0,0)$, $B(1,0)$ $C(0,1)$. Le principe est de suivant :

1. Choisir un point P au hasard à l'intérieur du triangle ABC. Pour ce faire, constatons que : $P(x,y) \in ABC \iff 0 \leqslant x \leqslant 1$ et $0 < y < 1-x$. On peut donc aisément se servir de la fonction `rand` pour trouver un tel P;
2. Choisir un des trois sommets $A, BetC$ au hasard qu'on nommera S;
3. Calculer le point M milieu de $[PS]$;
4. Redéfinir P par $P = M$. Stocker l'abscise de P dans une matrice X et son ordonnée dans une matrice Y.
5. Réitérer N fois à partir de l'étape 2.

Enfin, on affiche l'ensemble des points obtenus.

Le code python est le suivant :

```python
import numpy as np
import matplotlib.pyplot as plt
def sierpinski() :
    A = (0,0)
    B = (1,0)
    C = (0,1)
    N = 20000

    X = np.zeros(N)
    Y = np.zeros(N)

    # choisir le point P dans le trianle ABC
    x = np.random.rand(1)[0]
    y = np.random.rand(1)[0]*(1-x)
    P = (x,y)
    for i in range(N) :
        # choisir un sommet S entre A, B et C
        S = [A, B, C][np.random.randint(3, size=1)[0]]

        # M milieu de [PS]
        M = ((P[0] + S[0])/2, (P[1] + S[1])/2)

        # P = M
        P = M
        # conservation de P
        X[i] = P[0]
        Y[i] = P[1]

    plt.title('triangle de SIERPINSKI')
    plt.plot(X, Y, linestyle='', marker='o', markersize=1)
    plt.show()
```

Nous vous remercions pour votre aimable attention !