Ensemble de Mandelbrot et exploration algorithmique

Ensemble de Julia et de Mandelbrot

Pierre Fatou et Gaston Julia sont deux mathématiciens français du début du XXe siècle dont les travaux constituent la base de l'étude des variations des fonctions à valeurs complexes.

Ils s'intéressent à la variation des suites (z_n) définies pour tout nombre complexe de départ z_0 par la relation de récurrence suivante:

$$z_{n+1} = z_n^2 + c$$

où c est un nombre complexe choisi arbitrairement.

On s'intéresse à l'ensemble de Julia *rempli* (en un seul morceau), c'est-à-dire l'ensemble des z_0 et pour un même nombre complexe c tels que (z_n) soit bornée.

Grosso modo, en fonction de la valeur de départ une suite peut diverger vers l'infini ou pas

Benoît Mandelbrot (1924-2010) continue les recherches dans les années 1980 et y contribue grandement. Il est le premier à avoir représenter graphiquement les ensembles de Julia grâce aux ordinateurs IBM auxquels ils avaient accès. L'ensemble de Julia est renommé pour lui rendre hommage

Aujourd'hui, les belles images renvoyées par les ordinateurs stimulent toujours l'intérêt des mathématiciennes et mathématiciens pour ce domaine d'étude :)

Suite bornée

Comme on ne peut pas prévoir si $(z_i)_{i \in N}$ diverge vers l'infini ou pas quand i tend vers l'infini à partir des paramètres de départ, on utilise un algorithme qui calcule itérativement les valeurs de (z_i) où le nombre complexe c est donné.

Dans notre programme, **i** ne tend pas vers l'infini mais vers un nombre défini par **n_iter**.

Si à partir d'un certain rang **i** on a $|z_i| > 2$ alors on arrête l'itération car on est sûr que la suite diverge vers l'infini (admis) et on renvoie **i** qui est forcément inférieur au nombre d'itération max.

Si au bout de n_iter itérations on n'a toujours pas dépassé le seuil alors on considère que la suite ne diverge pas et on renvoie i qui est alors égal au nombre d'itérations max.

Notes pour plus tard

i varie entre 0 et n_iter.

La valeur de i montre la vitesse à laquelle la suite diverge vers l'infini ou pas : plus la valeur est petit et plus la suite diverge rapidement.

Ainsi on pourra colorier notre repère : les suites qui ne divergent pas seront toutes représentées par la même couleur parce qu'elles sont toutes associées au même i et les suites qui divergent seront représentées par différentes couleurs en fonction des différentes valeurs de i.

La fonction est_bornee prend en paramètres

- un nombre complexe c
- une valeur de départ z
- le nombre d'itérations n_iter
  Elle renvoie
- n_iter si la suite ne diverge pas
- le rang i à partir duquel la suite diverge

```
1  def est_bornee(z, c, n_iter):
2      i = 0 #compteur
3      w = z
4      while i < n_iter and abs(w) <= 2:
5          w = w * w + c #suite
6          i += 1
7      return i
```

Paramètres

Info

> On utilise un repère pour représenter graphiquement l'ensemble rempli. Chaque point du repère correspond à une unique suite de l'ensemble dont la valeur de départ est l'affixe de ce point.

Une instance de la classe **Paramètres** contient:

- un nombre complexe **c**
- le nombre maximal d'itérations **n_iter**
- les coordonnées du centre du repère affichée **(xc, yc)**
- le nombre de pixels améliore la netteté du dessin **n**
- un facteur de zoom **zoom**

Cette classe permet de créer itérativement l'ensemble de nos suites.

Plus les valeurs de **n_iter** et de **n** sont élevées et plus le temps de réponse est long (peut facilement dépasser 2min)

```
class Parametres:

    def __init__(self, c, xc=0, yc=0, n=300, zoom=60, n_iter=64):

        self.c = c #complexe
        self.xc = xc #centre axe abscisses
        self.yc = yc #centre axe ordonnées
        self.n = n #nombre de pixels (netteté)
        self.zoom = zoom
        self.n_iter = n_iter
```

Les attributs **zoom xc** et **yc** sont très utiles parce qu'ils permettent de définir précisément la taille du repère. Ainsi on peut se déplacer dans l'image.

```
        # portion affichée de l'image
        taille = 2 / zoom
        self.xmin = xc - taille
        self.xmax = xc + taille
        self.ymin = yc - taille
        self.ymax = yc + taille
```

Affixe d'un pixel

La fonction **affixe** calcule l'affixe d'un point du repère d'abscisse **i** et d'ordonné **j**.

```python
def affixe(pixel, parametre):
    i, j = pixel
    x = parametre.xmin + i * (parametre.xmax - parametre.xmin) / parametre.n
    y = parametre.ymax + j * (parametre.ymin - parametre.ymax) / parametre.n
    return x + 1j * y
```

> **Note**
>
> On peut écrire pour tourner l'image :
> x = parametre.xmin + j ...
> y = parametre.ymax + i ...

Matrice nulle

On représente notre repère par une matrice : chaque cellule représente un point du repère et chaque point du repère représente une suite de l'ensemble de Julia. Elle est au départ nulle.

La fonction **matrice_nulle** renvoie une matrice nulle de taille **n**.

```python
def matrice_nulle(n):
    m = [None] * n # lignes
    for ligne in range(n):
        m[ligne] = n * [0] # colonnes
    return m
```

Dessiner l'ensemble de Julia avec Matplotlib

Récapitulatif :

- fonction **est_bornee** vérifie si une suite diverge ou pas
- fonction **affixe** calcule l'affixe d'un point de l'image
- fonction **matrice_nulle** renvoie une matrice nulle qui représente numériquement l'image affichée sur l'écran
- classe **Paramètre** initialise différents paramètres de l'image comme par exemple l'origine du repère est sa taille.

1. Calculer l'ensemble des données

La fonction **paramètres_julia** remplit itérativement chaque cellule d'une matrice nulle avec les résultats de la fonction **est_bornee**.

Une cellule correspond à un point du repère et chaque point correspond à une unique suite de l'ensemble dont la valeur de départ est l'affixe de ce point.

```
def parametres_julia(parametre):
    matrice = matrice_nulle(parametre.n)
    for i in range(parametre.n):
        for j in range(parametre.n):
            z = affixe((i, j), parametre)
            k = est_bornee(
                z=z,
                c=parametre.c,
                n_iter=parametre.n_iter)
            matrice[i][j] = k
    return matrice
```

2. Afficher le graphique

La fonction **afficher_julia** colorie chaque point du repère en fonction des valeurs de la matrice.

Les couleurs "Blues" constitue une échelle de couleurs : plus la valeur d'un point est élevé et plus la couleur du point est foncé. Inversement plus la valeur est petite et plus la couleur est claire.

Cette variation de couleurs permet de distinguer visuellement les suites qui divergent de celles qui convergent.

```
def afficher_julia(matrice, parametre):
    plt.imshow(
        matrice, # les données
        interpolation='bicubic',
        cmap="Blues", # couleurs
        extent=(
            parametre.xmin,
            parametre.xmax,
            parametre.ymin,
            parametre.ymax)
        )

    plt.show()
```

3. Initialiser le programme

```
import matplotlib.pyplot as plt

...


#initialise le repère
figure, axes = plt.subplots()

#cache les axes, par esthétisme
axes.set_axis_off()

#initialise les paramètres globaux du programme
#ici on visuellement l'entiereté de l'image
parametre = Parametres(
    c = -0.744 + 0.145 * 1j,
    xc=0,
    yc=0,
    n=500,
    n_iter=300,
    zoom=1
)

#verifie si les suites sont bornées ou pas
image = parametres_julia(parametre)

#ouvre une nouvelle fenêtre de l'image
afficher_julia(image, parametre)
```