Modularité

1 Module

Un module en Python est un fichier .py, par exemple `turtle.py`, permettant d'utiliser des fonctions pour réaliser des programmes. Ainsi avec le module `turtle.py`, que l'on peut importer dans un autre fichier .py, on peut réaliser des programmes de dessin. Ce n'est pas le seul module de Python, permettant de faire des dessins.

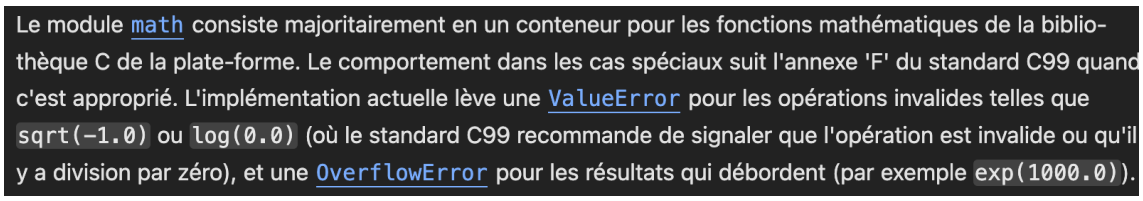
Un autre exemple, pour calculer la racine carrée d'un nombre on a utilisé la fonction `sqrt` du module `math`

L'idée générale est d'isoler dans un module un certain nombre d'outils réutilisables par tous via une `interface`, qui est une sorte de contrat, accessible en ligne qui assure à l'utilisateur le bon fonctionnement de l'outil à condition qu'il suive les recommandations de l'interface.

Un utilisateur n'a pas à savoir dans le détail le fonctionnement interne du module, quoiqu' en général le code source de ce dernier est accessible.

Exercice

1. Aller voir l'interface du module `turtle.py` ici `https://docs.python.org/fr/3.13/library/turtle.html#turtle-method` puis le code source ici `https://github.com/python/cpython/blob/3.13/Lib/turtle.py` et observer la présence de tests dans le module à la ligne 4015.
2. Voir ici le code source du module `math.py` qui renvoie vers -> `https://github.com/python/cpython/blob/main/Modules/mathmodule.c` .



Pourquoi utiliser le langage C pour le calcul numérique ? La réponse est ici `https://fr.wikipedia.org/wiki/C_(langage)`.

2 Imports de module

Dans la mesure du possible on évite de tout importer en faisant `from turtle import *`, ou `from math import *`.

On le fait avec des débutants en programmation mais il vaut mieux importer juste ce que l'on a besoin en faisant par exemple

`from turtle import Turtle` ou `from math import sqrt`

3 Package

`numpy`, pour le calcul scientifique, n'est pas un module mais un ensemble de modules rangés dans un dossier. On appelle cela un `package`.

Ainsi le module `polynomial.py` est rangé avec d'autres modules dans un dossier `Polynomial` lui-même rangé avec d'autres dossiers et d'autres fichiers dans le dossier `numpy`.

Ce qui explique dans l'interface les différents niveaux exposés :

`numpy.polynomial.polynomial` signifie le module `polynomial.py` dans le dossier `polynomial` dans le dossier `numpy`

Power Series (`numpy.polynomial.polynomial`)

This module provides a number of objects (mostly functions) useful for dealing with polynomials, including a `Polynomial` class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with polynomial objects is in the docstring for its "parent" sub-package, `numpy.polynomial`).

Classes

`Polynomial`(coef[, domain, window, symbol])	A power series class.

Quelques méthodes de la classe `Polynomial`

Arithmetic

`polyadd`(c1, c2)	Add one polynomial to another.
`polysub`(c1, c2)	Subtract one polynomial from another.
`polymulx`(c)	Multiply a polynomial by x.
`polymul`(c1, c2)	Multiply one polynomial by another.
`polydiv`(c1, c2)	Divide one polynomial by another.
`polypow`(c, pow[, maxpower])	Raise a polynomial to a power.
`polyval`(x, c[, tensor])	Evaluate a polynomial at points x.

Calculus

`polyder`(c[, m, scl, axis])	Differentiate a polynomial.
`polyint`(c[, m, k, lbnd, scl, axis])	Integrate a polynomial.

Dans l'interface de la classe `Polynomial`, l'explication de la méthode `polyadd` permettant de faire la somme de deux polynômes.

numpy.polynomial.polynomial.polyadd

`polynomial.polynomial.polyadd(c1, c2)` [source]

Add one polynomial to another.

Returns the sum of two polynomials *c1* + *c2*. The arguments are sequences of coefficients from lowest order term to highest, i.e., [1,2,3] represents the polynomial `1 + 2*x + 3*x**2`.

Parameters:

c1, c2 : *array_like*

1-D arrays of polynomial coefficients ordered from low to high.

Returns:

out : *ndarray*

The coefficient array representing their sum.

4 Un module outils.py

Dans le but de dessiner des fractales à partir de suites récurrentes on va réaliser un module appelé `outils.py`.

Dans ce module il y aura deux classes, une classe `Polynome` et une classe `Complexe`.

Ensuite dans deux autres fichiers `mandelbrot.py` et `julia.py` on importera le module `outils` pour dessiner des fractales.

Nous ne faisons pas cela pour des raisons d'efficacité sinon on aurait utilsé `numpy` et `matplotlib` mais pour illustrer la notion de **modularité**

4.1 La classe Polynome

1. Créer une classe `Polynome` dont les attributs sont `coeff` de type `list` contenant les coefficients du polynome et `deg` de type `int` le degré du polynôme. Il n'y a que la liste des coefficients du polynôme comme paramètre du constructeur.

```
>>> p = Polynome([1,2,0,4])
```

2. Ecrire la méthode `__str__` permettant de visualiser un objet de type Polynome.

```
>>> p = Polynome([1,2,0,4])
>>> print(p)
1+2x+4x^3
```

3. Ecrire une méthode `__add__ (self,other)` qui renvoie la somme de deux polynomes.

```
>>> q = Polynome([0,3,0,0,-1,0,1])
>>> print(p + q)
1+5x+4x^3-x^4+x^6
```

4. Ecrire la méthode `derive` qui renvoie le polynôme dérivé de self.

```
>>> print(p.derive())
2+12x^2
```

Rappel : $(x^n)' = nx^{n-1}$

4.2 La classe Complexe

1. Créer une classe `Complexe` dont les attributs sont `x` de type `float` et `y` de type `float`.

```
>>> z = Complexe(1,2)
```

2. Ecrire la méthode `__str__` permettant de visualiser un objet de type Complexe.

```
>>> z = Complexe(1,2)
>>> print(z)
1+2i
```

3. Ecrire une méthode `__add__ (self,other)` qui renvoie la somme de deux nombres complexes.

```
>>> z2 = Complexe(2,3)
>>> print(z + z2)
3 + 5i
```

4. Ecrire la méthode `__mul__ (self,other)` qui renvoie le produit de deux nombres complexes.

```
>>> print(z*z2)
-4 + 7i
```

5. Ecrire la méthode `__abs__ (self)` qui renvoie la norme de self.

```
>>> abs(z)
2.2360679774997 9
```

4.3 La classe Grille

Pour créer une grille de nombre complexes

```
def subdivision(amin,amax,nb_pts):
        pts = [amin]
        pt = amin
        pas = (amax - amin)/(nb_pts -1)
        for i in range(1,nb_pts):
            pt += pas
            pts.append(pt)
        return pts

class Grille:

    def __init__(self,xmin,xmax,ymin,ymax,
    nb_pts_x,nb_pts_y):

        pts_x = subdivision(xmin,xmax,nb_pts_x)
        pts_y = subdivision(ymin,ymax,nb_pts_y)

        self.grille_z = []
        for j in range(len(pts_y)):
            ligne = []
            for i in range(len(pts_x)):
                z = Complexe(pts_x[i],pts_y[j])
                ligne.append(z)
            self.grille_z.append(ligne)
```

5 Visualisation de données

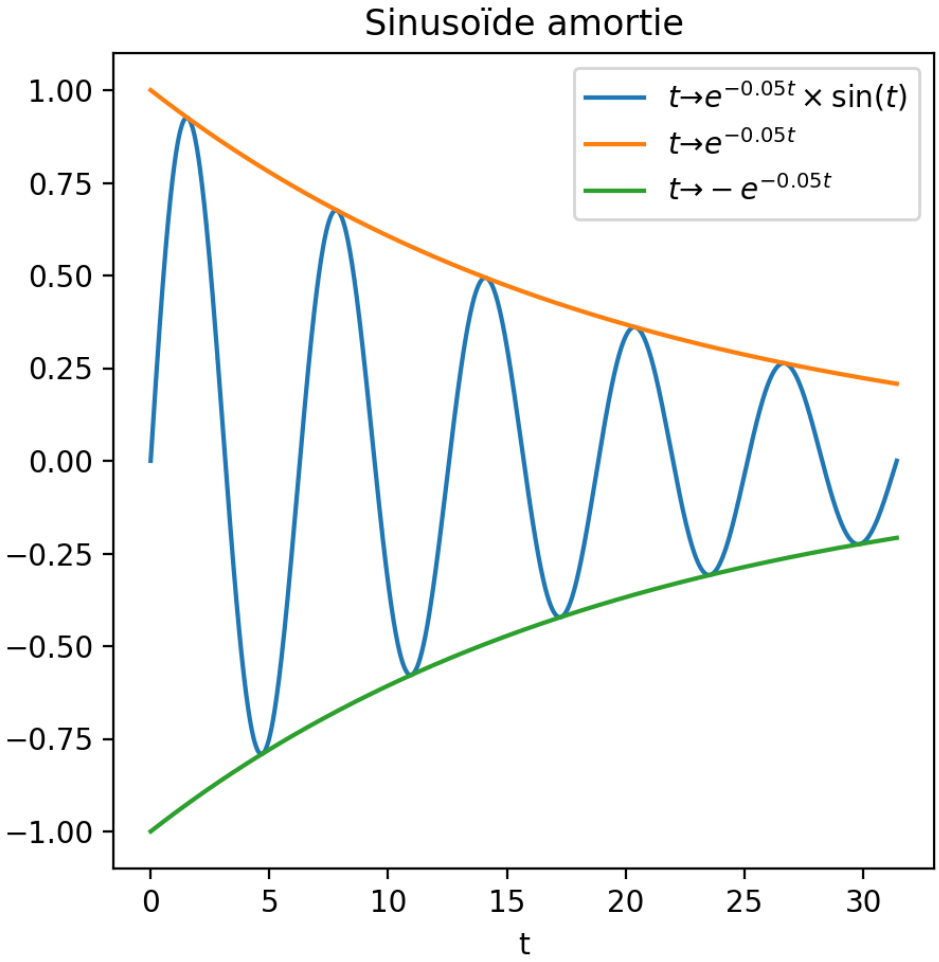
On importe le module `pyplot` du package `matplotlib` pour tracer des courbes et dessiner des images.

```
import matplotlib.pyplot as plt
from numpy import linspace
from math import pi,sin,exp
k = 0.05
x = linspace(0,10*pi,1000)

y = [sin(t)*exp(-k*t) for t in x]
z = [exp(-k*t) for t in x]
w = [-exp(-k*t) for t in x]

fig, ax = plt.subplots(figsize=(5, 5))
ax.plot(x, y,
label=r'$t \to e^{-0.05t}\times\sin(t)$')
ax.plot(x, z,
label=r'$t \to e^{-0.05t}$')
ax.plot(x, w,
label=r'$t \to -e^{-0.05t}$')
ax.set_xlabel('t')
ax.set_title("Sinusoide amortie")
ax.legend()
plt.show()
```

Ce qui permet de visualiser une sinusoïde amortie encadrée par deux exponentielles.



Dans un fichier texte on dispose de mesures de températures de 1864 jusqu'à 2018 pour la Suisse.

On souhaite visualiser l'évolution de la température annuelle sur cette période. On peut procéder comme précédemment et tracer la courbe des températures en fonction de l'année. Cependant dans un premier temps il faut ouvrir le fichier texte en lecture et extraire les données pour les transformer en listes, avant d'utiliser matplotlib.

En procédant ainsi, puis faire afficher les 5 premières et les 5 dernières années ainsi que les températures correspondantes :

```
donnees = []
with open('tempSuisse.txt',"r",encoding='utf-8') \
as fichier:
    for ligne in fichier:
        liste = ligne.split('\t')
        donnees.append((liste[0],
        liste[-1].replace('\n','')))

donnees = donnees[16:-1]
annee = [int(elt[0]) for elt in donnees]
temp = [float(elt[1]) for elt in donnees]
print(annee[:5]," ",temp[:5])
print(annee[-5:]," ",temp[-5:])
```

Ensuite on va utiliser ces listes pour tracer la courbe des températures en fonction des années

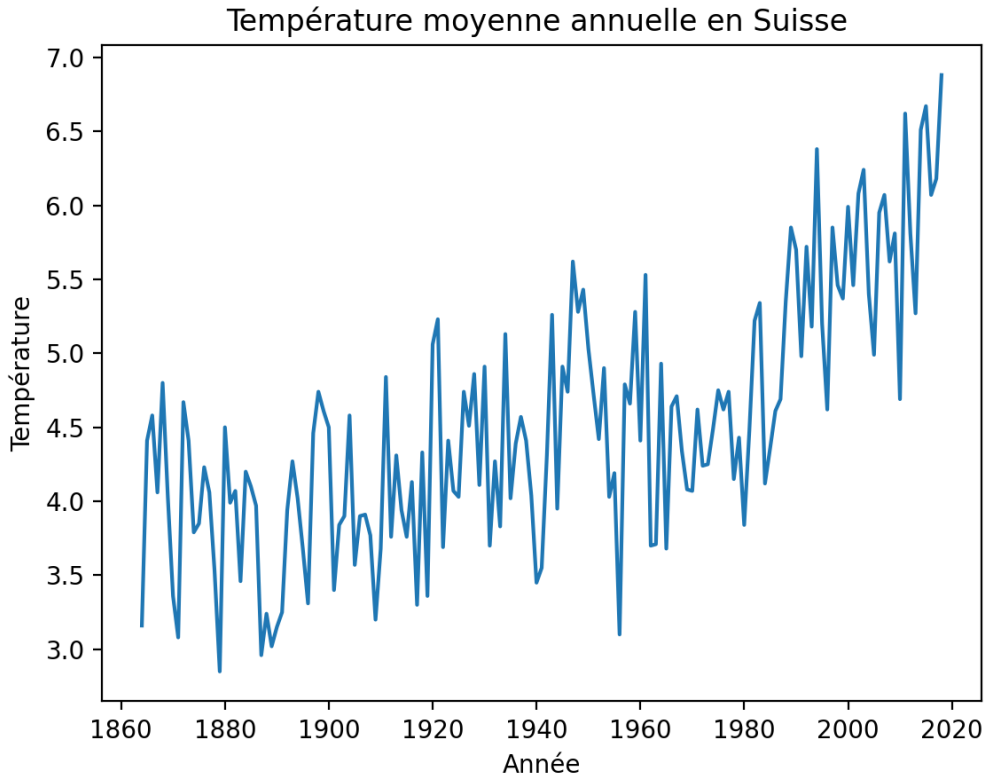
```
import matplotlib.pyplot as plt

donnees = []
with open('tempSuisse.txt',"r",encoding='utf-8') \
as fichier:
    for ligne in fichier:
        liste = ligne.split('\t')
        donnees.append((liste[0],
        liste[-1].replace('\n','')))

donnees = donnees[16:-1]
annee = [int(elt[0]) for elt in donnees]
temp = [float(elt[1]) for elt in donnees]
plt.plot(annee,temp)
plt.xlabel('Année')
```

```
plt.ylabel('Température (C)')
plt.title('Température moyenne annuelle en Suisse '
plt.show()
```

On obtient alors



Le climatologue Ed Hawkins a eu l'idée de traduire une échelle de température par une échelle de couleurs, allant du bleu (pour les températures les plus basses) au rouge.

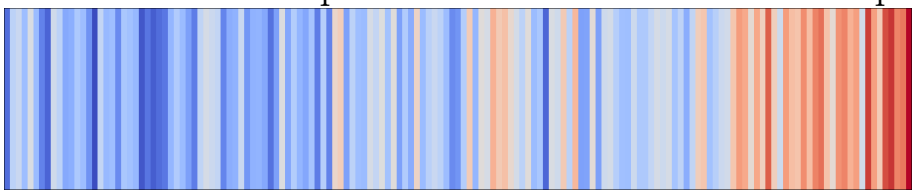
`matplotlib` a déjà des échelles de couleurs prédéfinies, (`cmap` pour colormap) par exemple `cmap='coolwarm'` ou `cmap='RdBu_r'` pour Red Blue reverse

On procède ainsi

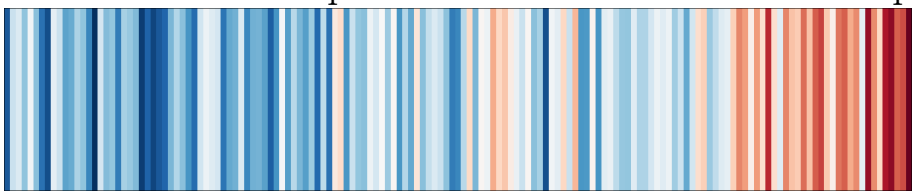
```
import matplotlib.pyplot as plt
from numpy import linspace,array
from matplotlib.patches import Rectangle
from matplotlib.collections import PatchCollection
from matplotlib.colors import ListedColormap
```

```
fig = plt.figure(figsize=(10, 2))
ax = fig.add_axes([0, 0, 1, 1])
col = PatchCollection([
    Rectangle((y, 0),1, 1)
    for y in range(1864, 2019)
])
col.set_cmap(cmap='coolwarm')
# la liste Python temp des températures
# est traduite en array de numpy
col.set_array(array(temp))
ax.add_collection(col)
ax.set_ylim(0, 1)
ax.set_xlim(1864, 2019)
plt.show()
```

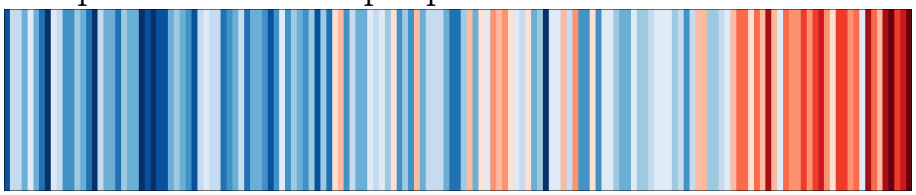
On obtient alors pour l'échelle de couleurs cmap='coolwarm'



On obtient alors pour l'échelle de couleurs cmap='RdBu_r'



On peut définir sa propre échelle de couleurs



```
cmap = ListedColormap([
    '#08306b', '#08519c', '#2171b5', '#4292c6',
    '#6baed6', '#9acab1', '#c6dbef', '#deebf7',
    '#fee0d2', '#fcbba1', '#fc9272', '#fb6a4a',
    '#ef3b2c', '#cb181d', '#a50f15', '#67000d',
])
```

6 Cartographier "l'instabilité " des orbites de nombres complexes

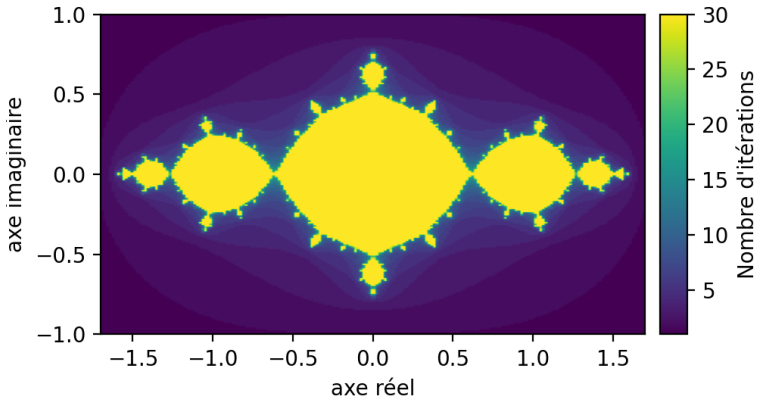
A partir d'une fonction $f : \mathbb{C} \ni z \to z^2 - 1 \in \mathbb{C}$ donnée et d'un point de départ z_0 appelé graine on suit l'orbite de ce point, c'est à dire $z_0 -> f(z_0) -> f(f(z_0)) -> ... -> f^{(n)}(z_0)$, où on note $f^{(n)}(z) = f(f(...(f(z)$ (on applique f n fois à z)

Quelques orbites :

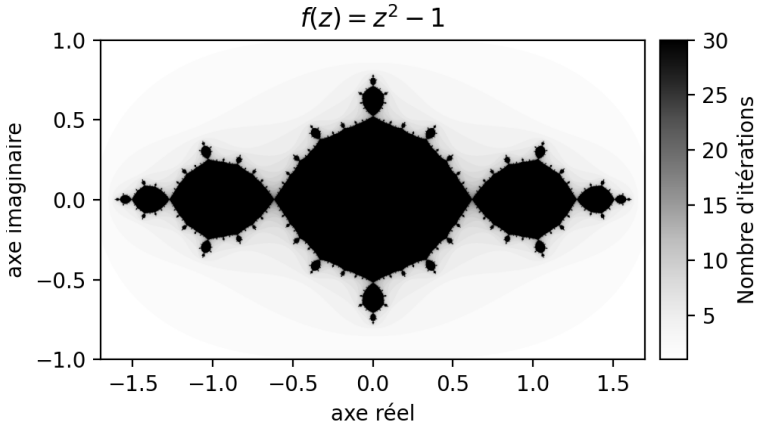
1. $z_0 = 0$, l'orbite est 0 -> -1 -> 0 -> -1, elle est 2-périodique et est stable, dans le sens où elle reste dans le voisinage de l'origine.
2. $z_0 = i$, l'orbite est i -> -2 -> 3 -> 8 -> ..., elle est considérée comme instable, car la suite des points s'éloigne de l'origine et tend en module vers $+\infty$
3. $z_0 = \phi = \dfrac{1+\sqrt{5}}{2}$. Il se trouve que $\phi^2 - 1 = \phi$ donc $f(\phi) = \phi$ donc l'orbite est fixe.
   $\phi -> \phi -> \phi ...$

Si on prend un millions de graines régulièrement réparties dans le rectangle [-1,7 ; 1,7] x [-1 ;1] et on "étudie" l'orbite de chacun d'eux en leur associant un nombre qui mesure l'instabilité de leur orbite, allant de 0 pour "très instable" à 30 "stable",

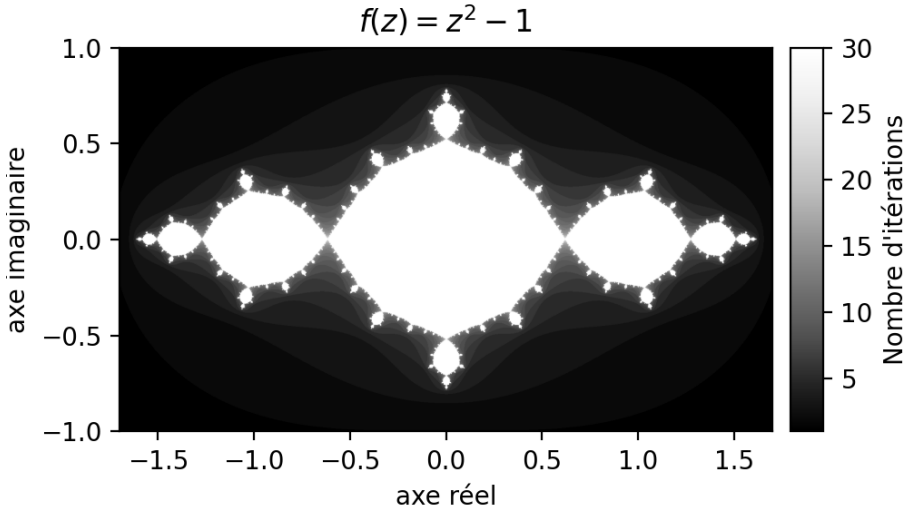
(voir détails plus loin) puis en convertissant cette échelle par une échelle de couleur uniforme de matplotlib cmap =’viridis’ on obtient



ou si cmap=’Greys’



ou si cmap='Greys_r'



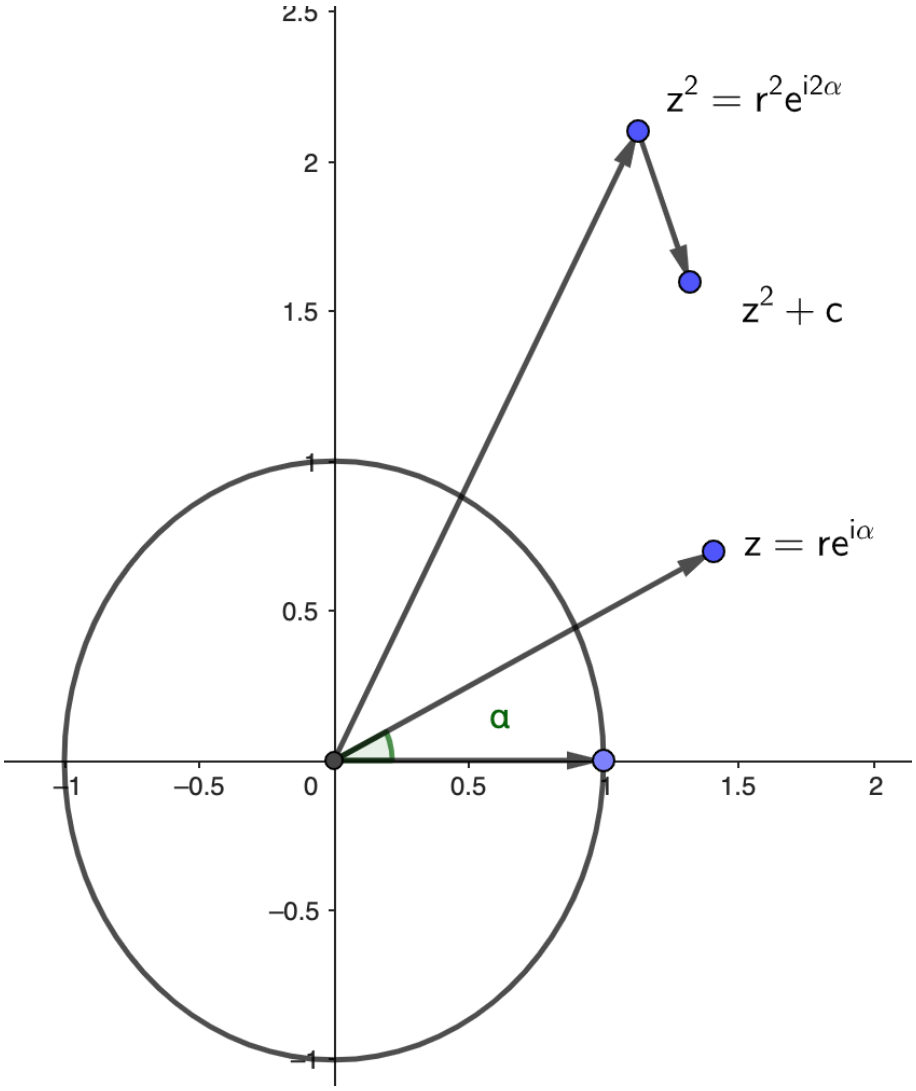
7 Détails

On appelle ensemble de Julia rempli K_c l'ensemble des graines $z_0 \in \mathbb{C}$ telles que la suite récurrente :

$z_{n+1} = z_n^2 + c$ avec la graine z_0 est **bornée**

Géométrie du problème :

$f(z) = z^2 + c$ peut être décomposée en une similitude (ie rotation suivi d'une homothétie) puis d'une translation



Propriété : On pose $r = max(|c|, 2)$ alors une suite récurrente du type précédent est bornée si et seulement si pour tout $n \in \mathbb{N}$ $|z_n| \leqslant r$

Lemme 1. *Pour tout nombre complexe* $z1$ *et* $z2$ *on a* $|z1| - |z2| \leqslant |z1 + z2| \leqslant |z1| + |z2|$

Lemme 2. $r^2 - |c| \geqslant r$

Preuve Supposons qu'il existe un rang m tel que $|z_m| > r$, c'est à dire il existe $h > 0$ tel que $|z_m| = r + h$

alors $|z_{m+1}| = |z_m^2 + c| \geqslant |z_m|^2 - |c| = (r + h)^2 - |c| = r^2 - |c| + 2rh + h^2 \geqslant r + 2rh$ Or $r \geqslant 2$ donc $r + 2rh \geqslant r + 4h$

Si on recommence

$|z_{m+2}| = |z_{m+1}^2 + c| \geqslant |z_{m+1}|^2 - |c| = (r+4h)^2 - |c| = r^2 - |c| + 8rh + h^2 \geqslant r + 8rh$ Or $r \geqslant 2$ donc $r + 2rh \geqslant r + 4^2h$

Par récurrence pour tout $k \in \mathbb{N}$ on a $|z_{m+k}| \geqslant r + 4^k h$ donc z_n n'est pas bornée.

On a donc un critère géométrique pour savoir si la suite est bornée, que l'on va programmer :

On fixe un nombre d'itérations maximal pour l'orbite, par exemple 100. Si au bout de ces 100 itérations l'orbite est restée à l'intérieur du disque de centre l'origine et de rayon r, on dit (avec un "léger risque de se tromper") que la suite est bornée.

Tous les cas sont possibles : au bout d'une ou deux itérations l'orbite peut sortir du disque limite, ou à la 30 ième.

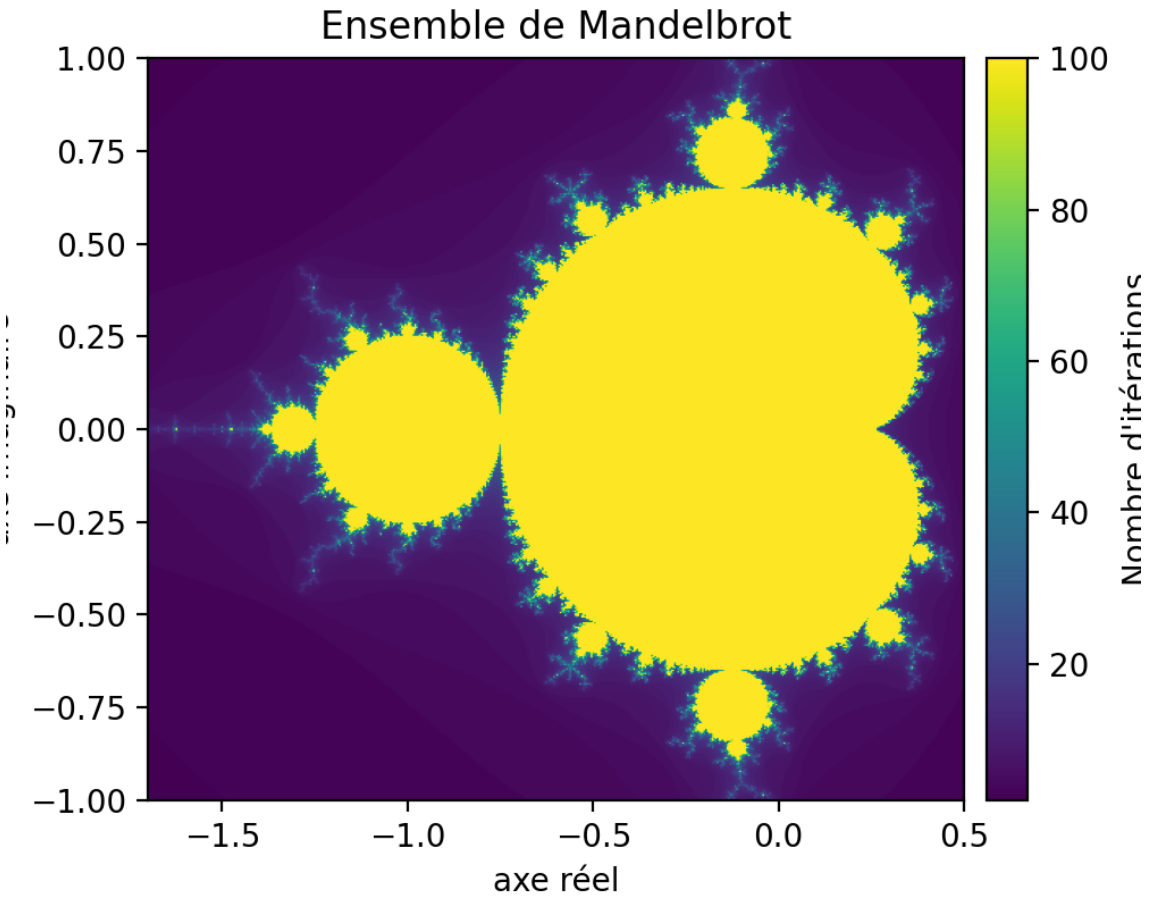
On a donc une échelle de 1 à 100 mesurant l'instabilité de la suite, que l'on va traduire ensuite par une échelle de couleurs.

8 Ensemble de Mandelbrot

On appelle ensemble de Mandelbrot l'ensemble des paramètres c tels que la suite récurrente :

$z_{n+1} = z_n^2 + c$ avec la graine $z_0 = 0$ est **bornée**

Il suffit de modifier légèrement le fichier `julia.py` pour obtenir un fihier `mandelbrot.py` pour obtenir les images suivantes :


Ensemble de Mandelbrot
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.5
−1.0
−0.5
0.0
0.5
axe réel
100
80
60
40
20
Nombre d'itérations