Théo Richard
Nathan Robin
Yann Rousseau
Louis Royet

Télécom Physique Strasbourg

General engineer training

Mathematics & IT project

2024 school year

First year

Reaction-diffusion equations and Turing's patterns

Contents

Part I
Introduction

Nowadays, reaction-diffusion systems are used in different fields, for example in pattern recognition, for ecological studies of species invasions, and in the health field (epidemiology, morphogenesis). But what is it exactly, and what will we study here ?

In 1952, Alan Turing, an English scientist, published one of the most famous paper in the history of biology. He hypothesized that common patterns seen on animals, such as zebras or leopards, are the consequence of two phenomena happening during the development of the embryo, leading to remarkable patterns. Those two phenomenons are simultaneous and in very particular initial conditions, lead to remarkable patterns. Turing explains this mechanism with a reaction between two reagents and simultaneously the diffusion of them.

Our goal here is to find out which are the initial conditions which will lead to some patterns, and to explain the code used to do so.
In this paper, we will focus on Gray-Scott's model to describe the reactions between the reagents U and V :

$$U + 2V \longrightarrow 3V$$
$$V \longrightarrow P$$

The equations describing this reaction but also the diffusion of the reagents are :

$$\frac{\partial u}{\partial t} = D_u \nabla^2 u - uv^2 + F(1-u)$$
$$\frac{\partial v}{\partial t} = D_v \nabla^2 v + uv^2 - (F+k)v$$

- $\frac{\partial u}{\partial t}$ (resp. $\frac{\partial v}{\partial t}$) is the temporal variation of u (resp. v) concentration $\in [0, 1]$
- $\nabla^2 u$ (resp. $\nabla^2 v$) is the laplacian of u (resp. v)
- D_u (resp. D_v) is the diffusion coefficient of u (resp. v)
- F is the feed rate
- k is the kill rate

Let's analyze the different terms of the equation :

→ $D_u \nabla^2 u$ is the **diffusion** term. The laplacian "looks around" to account for the local variation of u and v. If the quantity of u is higher around the position, then u in this position will increase. If this is not the case, u will decrease. Same for $D_v \nabla^2 v$.

→ $\pm uv^2$ is the **reaction** rate. It corresponds to the reaction above ($U + 2V \longrightarrow 3V$). That is why this term is negative in the first equation, positive in the second equation.

→ $F(1-u)$ is the **feeding** term. Because U disappears, it is necessary to supply new U. $(1-u)$ means that the system is fed more when their is low u concentration in a place, and fed less if their is high u concentration.

→ $(F+k)v$ is the **kill** term. The amount of V needs to be controlled because if not, it will just increase. This is why we subtract this term, which depends on the concentration of V (the more V there is, the more V is removed).

Part II
Ordinary differential equation establishment

Reminder of the equations :

$$\frac{\partial u}{\partial t} = D_u \nabla^2 u - uv^2 + F(1-u)$$
$$\frac{\partial v}{\partial t} = D_v \nabla^2 v + uv^2 - (F+k)v$$

We will use two mathematical approximations to be able to resolve them through IT code : Euler's method and the finite difference method.

Let's define h_t, the time interval.
Euler's method then gives us the following approximation of $\frac{\partial u}{\partial t}$:

$$\frac{\partial u}{\partial t} = \frac{u(x, y, \mathbf{t} + \mathbf{h_t}) - u(x, y, t)}{h_t} + O(h_t)$$

Let's define h_s, the spatial interval.
We can then discretize $\frac{\partial^2 u}{\partial x^2}$ (resp. $\frac{\partial^2 u}{\partial y^2}$) with the finite difference method :

$$\frac{\partial^2 u}{\partial x^2} = \frac{u(\mathbf{x} + \mathbf{h_s}, y, t) - 2u(x, y, t) + u(\mathbf{x} - \mathbf{h_s}, y, t)}{h_s^2} + O(h_s^2)$$

We then obtain the general formula for u and v at the moment $t + h_t$:

$$u(x, y, \mathbf{t} + \mathbf{h_t}) = u(x, y, t) + h_t \left[\frac{D_u}{h_s^2} \Big(u(\mathbf{x} + \mathbf{h_s}, y, t) + u(\mathbf{x} - \mathbf{h_s}, y, t) + u(x, \mathbf{y} + \mathbf{h_s}, t) + u(x, \mathbf{y} - \mathbf{h_s}, t) - 4u(x, y, t) \Big) - u(x, y, t)v^2(x, y, t) + F\big(1 - u(x, y, t)\big) \right]$$

$$v(x, y, \mathbf{t} + \mathbf{h_t}) = v(x, y, t) + h_t \left[\frac{D_v}{h_s^2} \Big(v(\mathbf{x} + \mathbf{h_s}, y, t) + v(\mathbf{x} - \mathbf{h_s}, y, t) + v(x, \mathbf{y} + \mathbf{h_s}, t) + v(x, \mathbf{y} - \mathbf{h_s}, t) - 4v(x, y, t) \Big) + u(x, y, t)v^2(x, y, t) - (F+k)v(x, y, t) \right]$$

Part III
Code implementation

Three programming languages are within our reach to answer this problem : C, Python and Matlab. Because of the size of our data, and the numerous calculus needed to be executed, speed is a serious topic. However, the main part of the job will rely on how functional is the data representation. Because it is not easy to represent the images, C is dismissed. For the resources required, Matlab is excluded. For his portability, the amount of knowledge of each individual member of the team in this language, and a good mix between quickness and good display, **Python** was the most logical choice.

1 Initial conditions

- **N** : The size of the images. The pictures are represented by a two-dimensional array, each elements (i, j) containing the concentration of a reagent in this particular pixel. The original suggestion for arrays shape was 512×512 to get a pretty satisfying image. However, for speed reasons, 256×256 is also a good option, reducing size by two and still providing really good images.
- the **feed** rate, the **kill** rate, the diffusion coefficients $\mathbf{D_u}$ and $\mathbf{D_v}$ (introduced in part one).
- the spatial step **dx** and the temporal step **dt**. (introduced in part two).
- **number of iterations** : for how long the system will evolve ($= nb_{iter} \times dt$).
- **frame** rate : necessary for the fluidity of the animation of the results.

Because the initial conditions have a considerable impact and are central in the problem, they will be discussed in detail later.

2 Functions

2.1 Pseudo-Random perturbation

The initialisation of the two arrays U and V will be discussed in detail in part **4**.

One of the main ideas we are trying to demonstrate is that for initial conditions fixed, the pattern we will obtain will not depend on the initial state of the system (the arrays U and V). For that, a randomize generation of the array U and the array V is needed.
However, initiating every pixel with a random number between 0 and 1 will not do the trick (see **IV.4.1**). We need some sort of shape at random place for the program to run properly.
In order to do so, a possible idea is to choose a pixel randomly, give it a high value, and assign a value that decreases according to a Gaussian around this reference pixel. It will create some sorts of stain at random places.

The Gaussian function used in two dimensions is the following :

$$g(x, y) = \exp\left[- \frac{(x - x_0)^2}{2\sigma_x^2} - \frac{(y - y_0)^2}{2\sigma_y^2} \right]$$

with x_0 and y_0 the coordinates of the "center" of the Gaussian.

Note : Because it was enough, the code implemented gives spots that look like stars more than stains.

```
#-------------------------------------------------

def gaussienne(x,x0,y,y0,sigmaX,sigmaY):

    e1 = (x-x0)**2 / (2 * sigmaX * sigmaX)
    e2 = (y-y0)**2 / (2 * sigmaY * sigmaY)

    return (np.exp(-e1 -  e2))

#-------------------------------------------------

def init_mat(nb_taches,taille_tache,N):
    U = np.ones((N,N))
    V = np.zeros((N,N))
    for k in range(nb_taches):
        i = rd.randrange(taille_tache+1,N-taille_tache-1) #random center of the star
        j = rd.randrange(taille_tache+1,N-taille_tache-1) #random center of the star
        for a in range(taille_tache):
            V[ i, j - a ] = gaussienne( i , i , j-a , j,7,7)      #voisin gauche
            V[ i, j + a ] = gaussienne( i , i , j+a , j,7,7)      #voisin droit
            V[ i + a, j ] = gaussienne( i+a , i , j , j,7,7)      #voisin bas
            V[ i - a, j ] = gaussienne( i-a , i , j , j,7,7)      #voisin haut
            V[ i - a, j - a] = gaussienne( i-a , i , j-a , j,7,7) #voisin haut gauche
            V[ i - a, j + a] = gaussienne( i-a , i , j+a , j,7,7) #voisin haut droite
            V[ i + a, j - a] = gaussienne( i+a , i , j-a , j,7,7) #voisin bas gauche
            V[ i + a, j + a] = gaussienne( i+a , i , j+a , j,7,7) #voisin bas droite

            U[ i, j - a ] = (1 - V[ i, j - a ])*0.8               #voisin gauche
            U[ i, j + a ] = (1 - V[ i, j + a ])*0.8               #voisin droit
            U[ i + a, j ] = (1 - V[ i + a, j ])*0.8               #voisin bas
            U[ i - a, j ] = (1 - V[ i - a, j ])*0.8               #voisin haut
            U[ i - a, j - a] = (1 - V[ i - a, j - a])*0.8         #voisin haut gauche
            U[ i - a, j + a] = (1 - V[ i - a, j + a])*0.8         #voisin haut droite
            U[ i + a, j - a] = (1 - V[ i + a, j - a])*0.8         #voisin bas gauche
            U[ i + a, j + a] = (1 - V[ i + a, j + a])*0.8         #voisin bas droite

    return U,V
#-------------------------------------------------
```

2.2 The Laplacian

The way to compute the Laplacian of an array has evolved over time.

This first way was a simple implementation of the equation seen in part II :

```
def discret_laplacien(f,x,y):
    h = f[x-1][y] #voisin du haut
    b = f[x+1][y] #voisin du bas
    g = f[x][y-1] #voisin de gauche
    d = f[x][y+1] #voisin de droite
    center = f[x][y]

    return ((h+b+g+d-4*center)/(hs*hs))
```

However, it was called for every pixel of the image and you can't calculate the laplacian in the corners and the edges of the image.

In order to optimize the code, make it cleaner and not setting conditions on the edges (which will result in the creation of a "toric" model), we are going to use the function *roll* from the numpy library :

```
def laplacien(M):
    L = -4*M
    L += np.roll(M, (0,-1), (0,1)) # add right neighbor
    L += np.roll(M, (0,+1), (0,1)) # add left neighbor
    L += np.roll(M, (-1,0), (0,1)) # add up neighbor
    L += np.roll(M, (+1,0), (0,1)) # add down neighbor

    return L/(hs*hs)
```

Note : how does the function roll works ? It creates a copy of the array in argument, shifted by one square in a given direction.

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} >> L = \texttt{np.roll(M,(0,-1),(0,1))} >> L = \begin{bmatrix} 2 & 3 & 1 \\ 5 & 6 & 4 \\ 8 & 9 & 7 \end{bmatrix}$$

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} >> L = \texttt{np.roll(M,(-1,0),(0,1))} >> L = \begin{bmatrix} 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

This way, we no longer call the function for each pixel. You only have to call the function once and it returns an array containing the Laplacian of each pixel at a time t. Moreover, we have just created a toric model: the right edge is a direct neighbor of the left edge, and the top edge is a direct neighbor of the bottom edge.
Despite this optimization, the results that follow will never be very satisfactory and very precise. It is therefore necessary to improve the discretization model of the Laplacian.

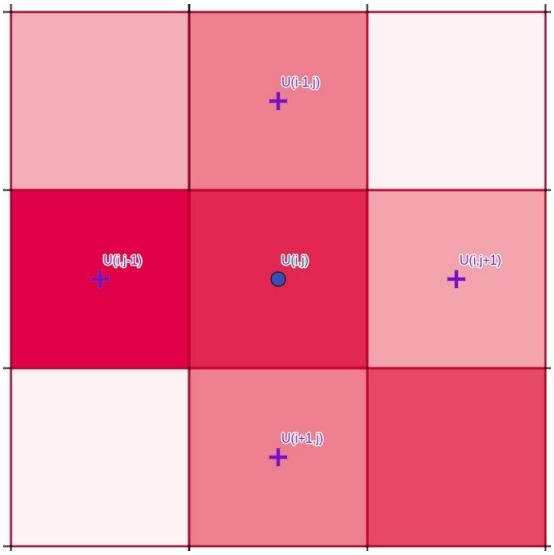


Figure 1: *Calculation of the laplacian using the 4 nearest neighbors*

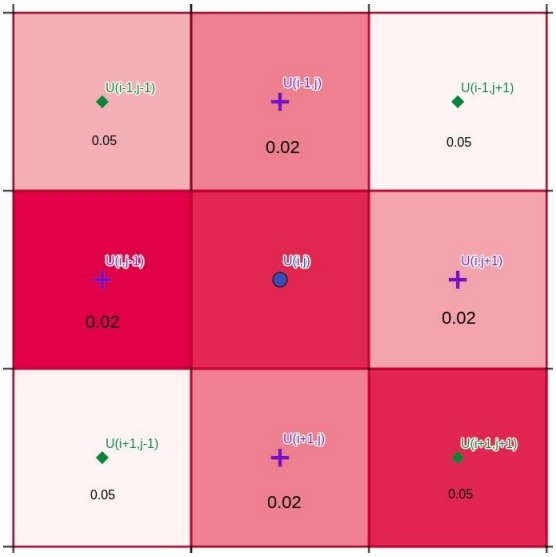


Figure 2: *Calculation of the laplacian using* ***every*** *neighbors*

To be more accurate, we are going to consider every neighbor surrounding the pixel. However, because some pixels are closer than others, they don't have the same weight. Therefore the four nearest neighbors will have a weight of 0.2 and the four diagonal neighbors, a little further away, will have a weight of 0.05. We do have $4 \times 0.2 + 4 \times 0.05 = 1$.
To conclude, the final function is :

```
def Laplacien(M):
        L = -1*M
        L += 0.2*np.roll(M, (0,-1), (0,1)) # voisin de droite
        L += 0.2*np.roll(M, (0,+1), (0,1)) # voisin de gauche
        L += 0.2*np.roll(M, (-1,0), (0,1)) # voisin du dessus
        L += 0.2*np.roll(M, (+1,0), (0,1)) # voisin du bas
        L += 0.05*np.roll(M, (-1,-1), (0,1)) #voisin diagonale droite bas
        L += 0.05*np.roll(M, (+1,-1), (0,1)) #voisin diagonale droite haut
        L += 0.05*np.roll(M, (+1,+1), (0,1))#voisin diagonale gauche haut
        L += 0.05*np.roll(M, (-1,+1), (0,1)) #voisin diagonale gauche bas

        return L/(dx*dx)
```

2.3 Update the state

The first approach did not take advantage of the numpy arrays. We were updating one pixel at a time. The very first version of the code was actually wrong because we didn't even use buffer arrays. However, it was necessary because the pixels were updated in the classic reading direction, meaning that the calculation of the laplacian was using some neighbors at the state t, and some neighbors at the state $t + dt$ ... The buffer arrays were there in order to have a copy of the array U and V at the state t.
That fixed, here is what the code looked like :

```
def resolution_ED():

    tamponU = np.zeros(taille_image)
    tamponV = np.zeros(taille_image)

        for t in range(nb_iter):
                for i in range(1,taille_image-1):
                        for j in range(1,taille_image-1):
                                uv2 = U[i][j]*V[i][j]*V[i][j]
                                tamponU = U
                                tamponV = V
                                U[i][j] += (D_u * discret_laplacien(tamponU,i,j)
                                - uv2 + F*(1-U[i][j]))*dt
                                V[i][j] += (D_v * discret_laplacien(tamponV,i,j)
                                + uv2 - (k+F)*V[i][j])*dt
```

Now, a smarter way to update the state is the following code :

```
    def update_etat(U, V, DU, DV, f, k, dt):

        LU = Laplacien(U)
        LV = Laplacien(V)

        diff_U = (DU*LU - U*V**2 + f*(1-U)) * dt
        diff_V = (DV*LV + U*V**2 - (k+f)*V) * dt

        U += diff_U
        V += diff_V
```

No more double-loop. We have already seen the improvement of the Laplacian function. Thanks to this improvement, no more buffer is needed and we take advantage of the calculations between numpy arrays.

3 Display and animation

First, let's introduce the display tools used.

3.1 plt.imshow

We have been using the `matplotlib.pyplot` library in order to go from array U to image U.
The main parameter is an **array** with RGB values for example, which means either floats between 0 and 1 or integers between 0 and 255.
The second parameter generally specified is **cmap** which stands for «colormap». It is a gradient of color and there are dozens available.

Example :

$$\begin{bmatrix} 0.000 & 0.033 & 0.067 & 0.100 & 0.133 & 0.167 \\ 0.167 & 0.200 & 0.233 & 0.267 & 0.300 & 0.333 \\ 0.333 & 0.367 & 0.400 & 0.433 & 0.467 & 0.500 \\ 0.500 & 0.533 & 0.567 & 0.600 & 0.633 & 0.667 \\ 0.667 & 0.700 & 0.733 & 0.767 & 0.800 & 0.833 \\ 0.833 & 0.867 & 0.900 & 0.933 & 0.967 & 1.000 \end{bmatrix}$$

Figure 3: `Array A`

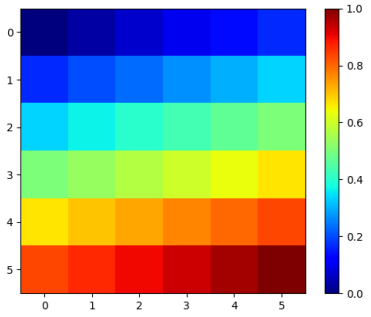


Figure 4: `plt.imshow(A,cmap='jet')`

3.2 GIF and Live modes

Let's take a look at the main function :

```
def main(feed,kill,Du,Dv):
        #plt.ion()
        #plt.figure()
        for k in range(nb_iter):
                update_etat(U,V,Du,Dv,feed,kill,dt)
                if k % frame == 0:
                        filename = "Uframe_{:02d}.png".format(k//frame)
                        print(filename)
                        plt.imshow(U, cmap="jet")
                        plt.savefig(filename)
                        #plt.pause(0.005)
```

This is the version that is used to generate GIFs. `frame` is a global variable defined at the beginning of the code which allows to manage the fluidity of the animation. `plt.imshow(U,cmap="jet")` creates the image as we just saw above. `plt.savefig(filename)` saves the image in the current program execution folder.
Once every image is generated, we use *ImageMagick* to convert those into a gif. Because this is not a one-step process, a small *bash* script is useful :

```
#!/bin/sh

#Go to the image-saving folder
cd /home/yann/Bureau/PMI/prog_Py/png_and_gif

#Execute the .py program
python3 ../main.py

#Convert the images into a GIF using ImageMagick
convert U*.png renameME.gif

#Once the conversion is done, clean by removing every image
rm U*.png
```

If we want to switch to the live animation, we need to comment some lines and uncomment others :

```
def main(feed,kill,Du,Dv):
        plt.ion()
        plt.figure()
        for k in range(nb_iter):
                update_etat(U,V,Du,Dv,feed,kill,dt)
                if k % frame == 0:
                        #filename = "Uframe_{:02d}.png".format(k//frame)
                        #print(filename)
                        plt.imshow(U, cmap="jet")
                        #plt.savefig(filename)
                        plt.pause(0.005)
```

`plt.ion()` turns on the interactive mode, which means we can now plot new things into the same figure. `plt.figure()` creates this particular figure that will stay open and will be refreshed. Finally, `plt.pause()` is necessary to slow-down just a little the program in order to let some time for plt.imshow() to actually plot the image.

3.3 Mentionable ideas.

The first thought was to find a way to represent U and V at the same time on the same figure. To do so, we created a third array, `W`, the **representation** array. How to fill it in? The idea was to look at each pixel of the two arrays, and compare the u and v concentration in this pixel (i,j). If $u(i,j) > v(i,j)$, then $u(i,j)$ "wins" this pixel, else if $v(i,j) > u(i,j)$ then $v(i,j)$ "wins" this pixel.
After deciding who won the pixel, we apply our own color gradient. If u wins, the pixel will be more or less red. If v wins, the pixel will be more or less blue.

However, we did realize that their is no need to do so. Representing a single array, that of U for example, perfectly captures the situation and `plt.imshow()` makes a much cleaner gradient.

Part IV
Initial conditions

4 Arrays initialization

It is important to define the initial matrix `U` and `V`. It will explain the convergence and the patterns formation in some cases. We can make different choices of initial matrix. For example matrix with random coefficients (see III.2.1). We can also add a perturbation on our own to see how it will diffuse through the whole matrix.

We will introduce simple tools such as the minimum, the average and the maximum of the matrix coefficients in order to study the convergence. We will only follow the evolution of the matrix U. We will calculate the minimum, the average and the maximum at each step/iteration and plot it.

```
abscisse = []
lmin = []
lmoy = []
lmax = []

abscisse.append(temps)
lmin.append(np.min(U))
lmoy.append(np.mean(U))
lmax.append(np.max(U))
```

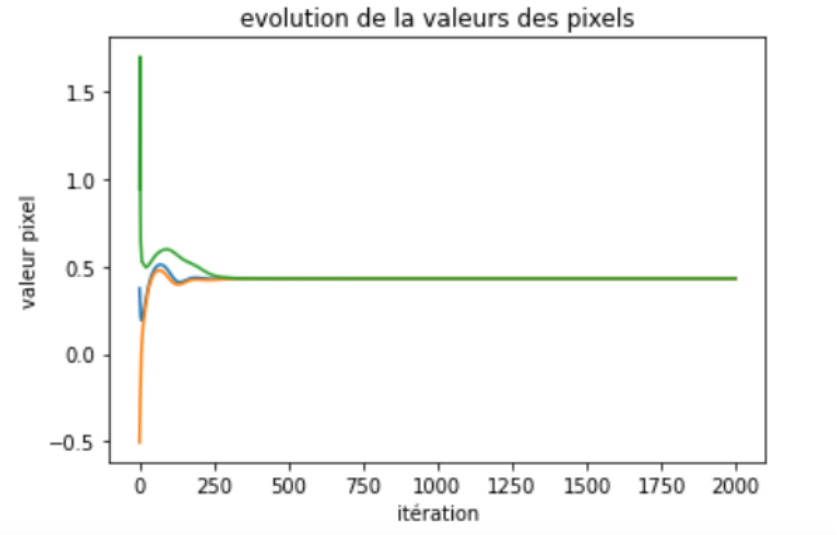
4.1 No perturbation and random matrix

We will first study the most simple case : two random matrix U and V with no perturbation. This choice is not insignificant because it is actually the best representation of the reality. The concentration of two species U and V is generally randomly distributed in space before the beginning of the chemical reaction.

```
import numpy as np
U = np.random.rand(N,N)
V = np.random.rand(N,N)
```

```
N = 512
F = 0.08
k = 0.06
Du = 0.16
Dv = Du/2
dx = 1
dt = 1
boucle = 2000
```

Here is the evolution of the pixels through the time :



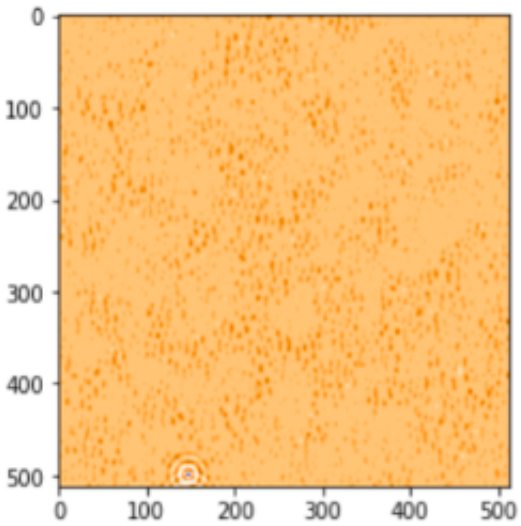
iteration	10	500	1000
minimum	0.116	0.429**27**	0.4292893**215**
average	0.203	0.429**29**	0.4292893**217**
maximum	0.523	0.429**33**	0.4292893**222**
span	4e-1	6e-5	7e-10

Observations :

- All the pixels quickly converge on the same value (0.429). Thus, this simulation doesn't allow patterns formation because all the pixels are at the same value (ie, same color) and as you can see, there is no evolution after the 400th iteration.

- At the very beginning, some pixels are above 1 while others are below 0. This is due to the calculation with the Laplacian method. However, this is not a big issue because the pixels quickly tend toward a common value between 0 and 1.

Let's look at the result with this following configuration. With no surprise, there is no specific pattern and the display doesn't evolve as the time goes on.



- Code improvement : we assign the value 0 to all the pixels that are below 0 and 1 to all of those that are above 1. We only change the value for the display.

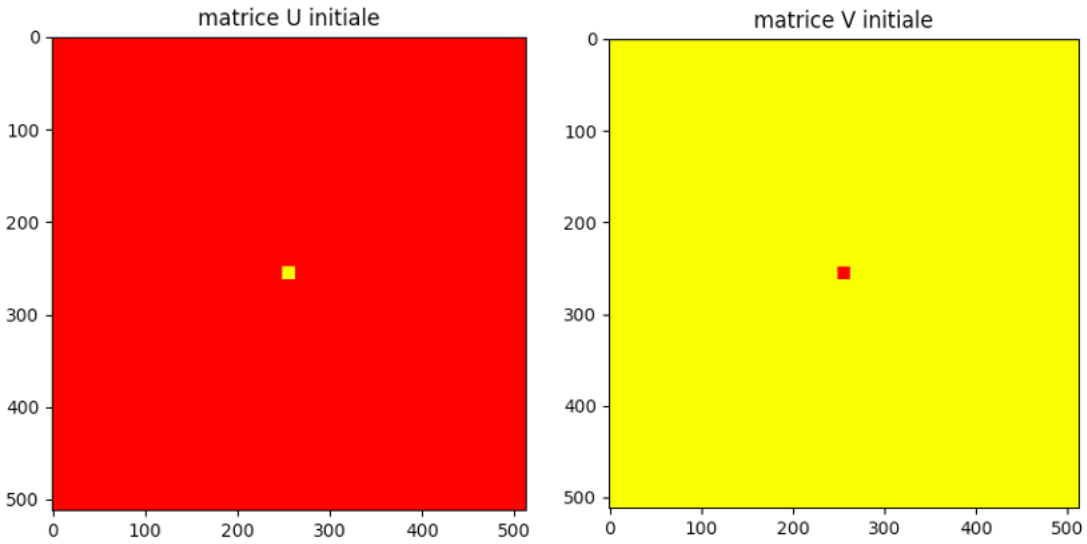
```
U_aff=np.copy(U)
        for i in range(N):
            for j in range(N):
                if U[i,j]<0:
                    U_aff[i,j]=0
                elif U[i,j]>1:
                    U_aff[i,j]=1

        plt.imshow(U_aff, cmap=plt.cm.PuOr)
```

4.2 Non-random matrix with perturbation

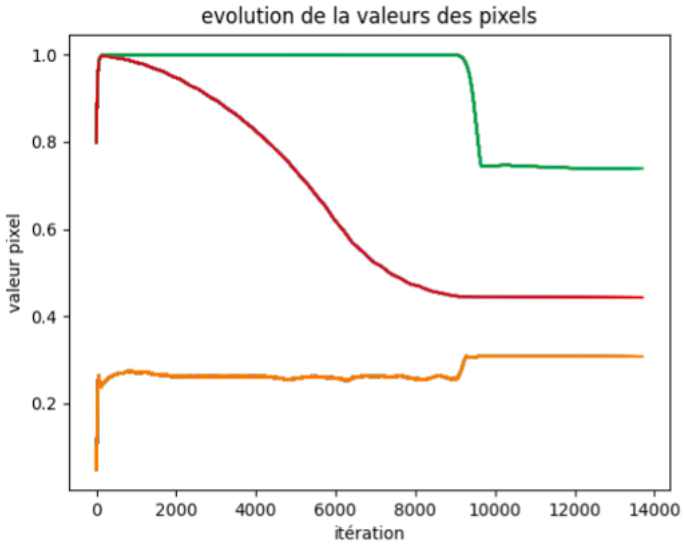
In order to obtain some patterns, we will use non-random matrix and we will add a central perturbation. We will chose a highly concentrated matrix U except at the center and a matrix V that has the opposite properties.

```
U = np.ones((N,N))*0.8
V = np.zeros((N,N))
U[( N // 2 ) - 7 : ( N // 2 ) + 7 , ( N // 2 ) - 7 : ( N // 2 ) + 7] = 0.1
V[( N // 2 ) - 7 : ( N // 2 ) + 7 , ( N // 2 ) - 7 : ( N // 2 ) + 7] = 0.9
```



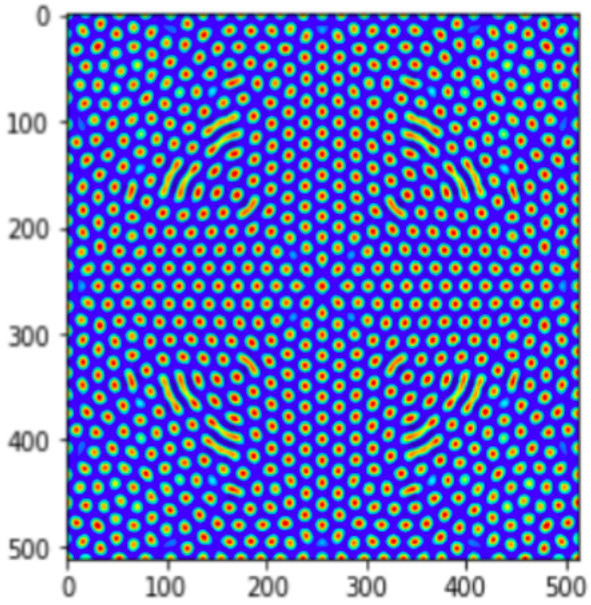

```
N = 512
f=0.039
k=0.058
Du=1
Dv=Du/2
dx=1
dt=1
nb_iter=100000
frame = 100
```

Let's look at the pixels evolution in this case.



We can see there that the difference between the maximum and the minimum is large. Thus, it allows

the pixels to take different values (ie to display different colors). The pixels values evolve as the time goes on and it creates beautiful patterns. Here is the picture we have after the 1500th iteration :



5 Parameters

We have already discussed the size of the image **N**, the number of iterations nb_{iter}, the **frame**.

5.1 Influence on the speed

- D_u and D_v : The higher they are, the faster the diffusion is. We did respect a rule every time though. It is $\frac{D_u}{D_v} = 2$.
- dt : The time increment. Increase it to speed up the animation. To be set with the number of iterations to get the desired study time. Generally, $dt = 1$ but we sometimes take $dt = 1/2$ or $dt = 2$ depending on the situation.
- dx : the spatial step involved in the Laplacian. Because of this link, it was a potential cause of many issues for a long time. Generally, $dx = 1$ but sometimes $dx = 2$ in order to slow down the diffusion.

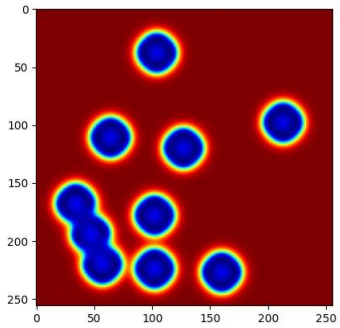


Figure 5: *Figure at iteration 150 for* $dx = 1$

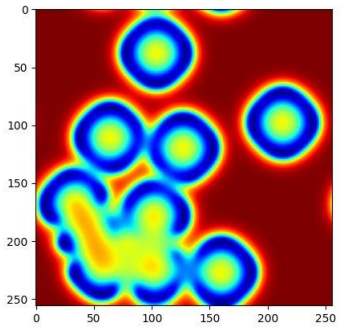


Figure 6: *Figure at iteration 450 for* $dx = 1$

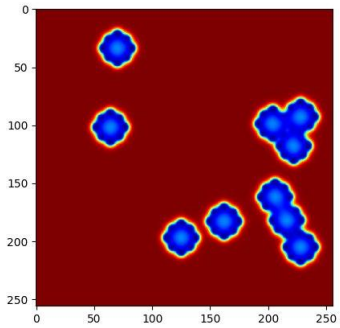


Figure 7: *Figure at iteration 150 for* $dx = 2$

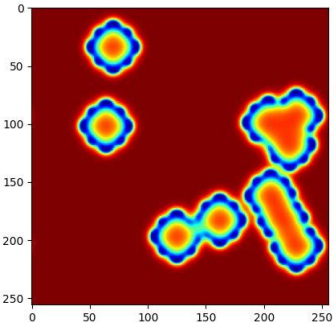


Figure 8: *Figure at iteration 450 for* $dx = 2$

5.2 Feed and Kill

Now this is the main event. Once the previous constants have been understood and somewhat fixed, it is on the **kill** and **feed** constants that everything is played out: from the appearance of a pattern to the total absence of reaction. Before exploring the many existing pattern families, let's try to frame these values to reduce our exploration.

5.2.1 Delimitation of k and f

As we witnessed divergence or a concentration overwhelming the other for f or k higher than 0.07, we narrowed down the problem to looking for "working k and f" between 0 and 0.07.
The method is the following: we set the other initial conditions and we will also set the arrays by defining specific initial concentration arrays.

```
U = np.ones((N,N))*0.8
V = np.zeros((N,N))
for i in range(int(0.45N),int(0.55N):
    for j in range(int(0.45N),int(0.55*N):
        V[i,j]=1
        U[i,j]=0
```

This means that we introduce a disturbance square of the reagent V in the middle. Then, we go through every combination of f and k possible between 0 and 0.07 with a 0.005 step and note every combination when patterns were witnessed.
With this method we could map and visualise 7 families of Turing patterns :

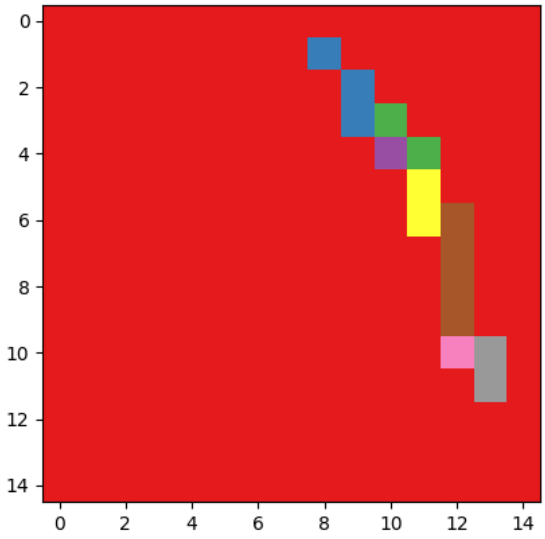


Figure 9: *feed as a function of kill. Each square of color represents one group of pattern (except red = no pattern) (Axes are not to scale)*

5.2.2 Pattern families

Now, this is the fun part : exploring through initial conditions in order to find new pattern, and be mesmerized by their formation and how they diffuse/react with each other.

Note : Because it is something very visual and moving, the reader is invited to explore the examples of GIFs accompanying this paper.

As we mentionned before, the results will depend on the initial arrays. Let's begin with arrays randomized by the gaussian function :

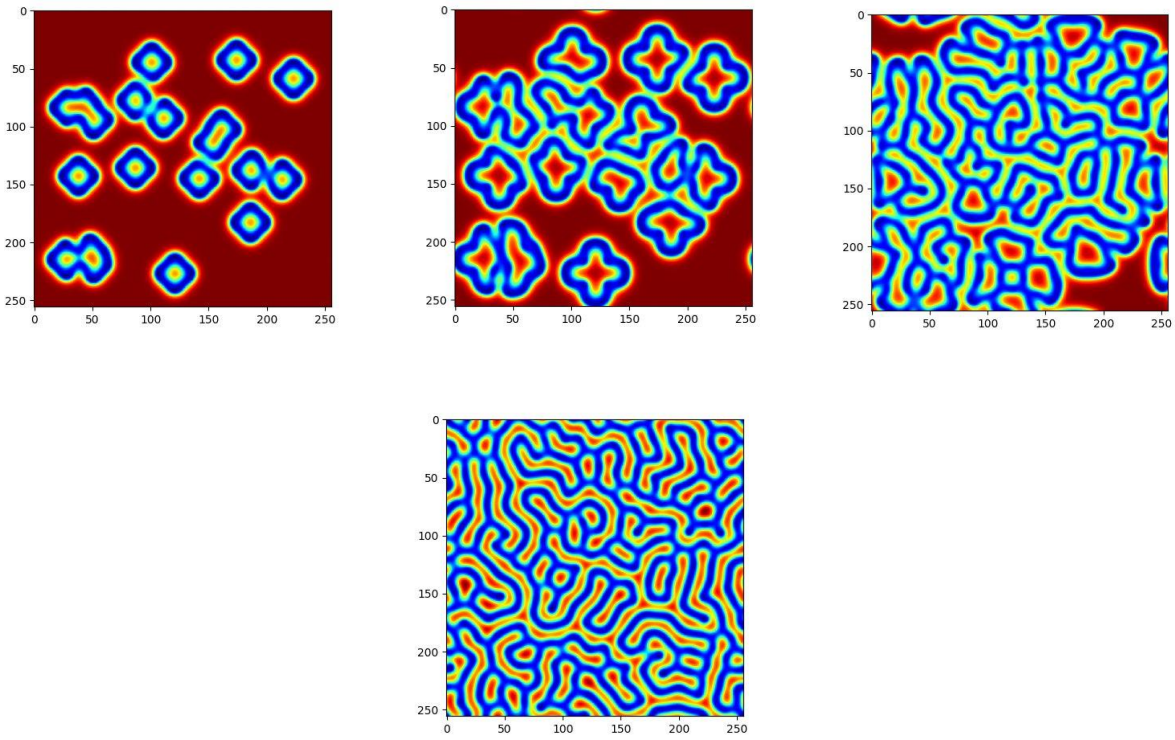


Figure 10: Stripes with f=0.04 | k = 0.06 | $D_u = 1$ | $dx = dt = 1$

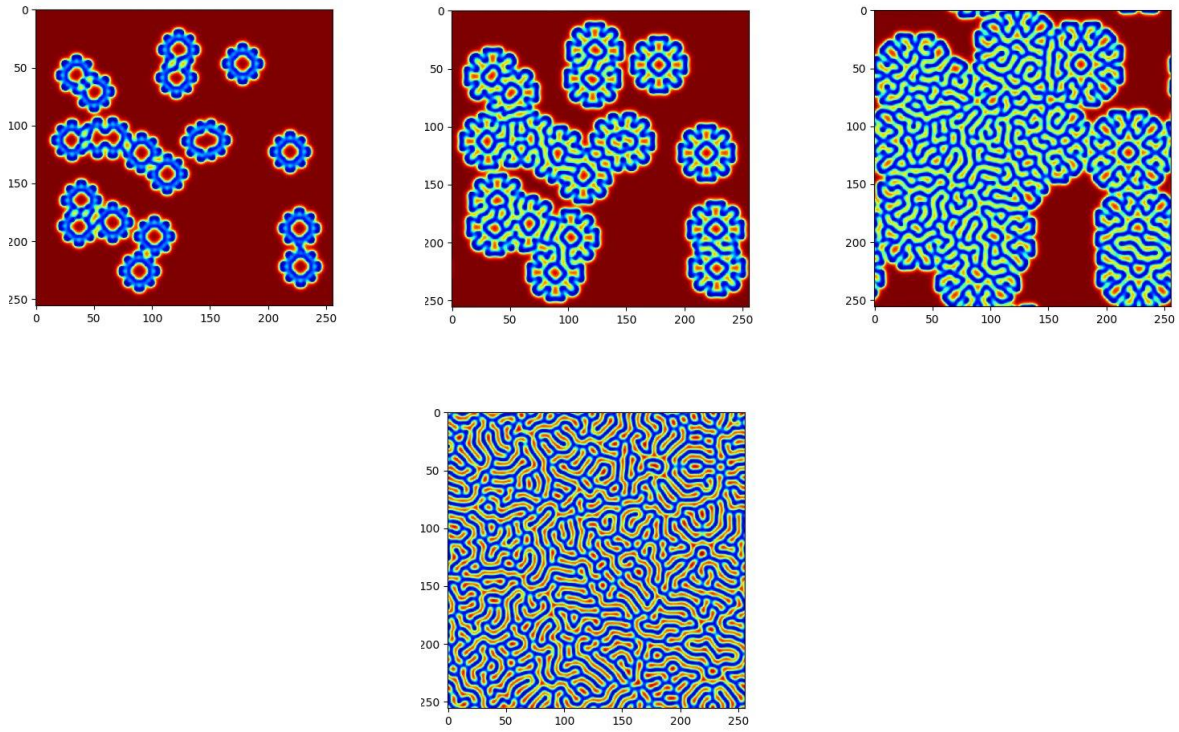
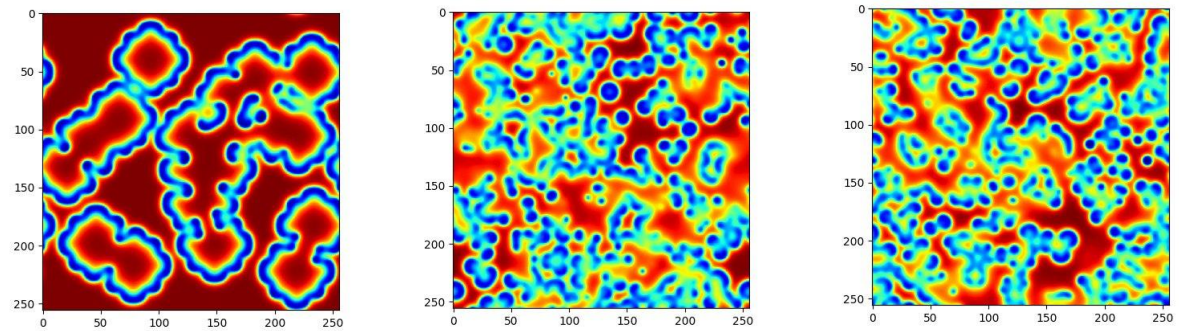


Figure 11: Stripes with same conditions than above except $dx = 2$

These are patterns that are stabilizing. But it is very easy to make chaos appear (which looks much better in GIF). Let's take for example

$$\text{f} = 0.018 \mid \text{k} = 0.051 \mid D_u = 2 \mid dx = 2 \mid dt = 1$$



Between that, another family of pattern which reminds us of the biology mentioned in the introduction:

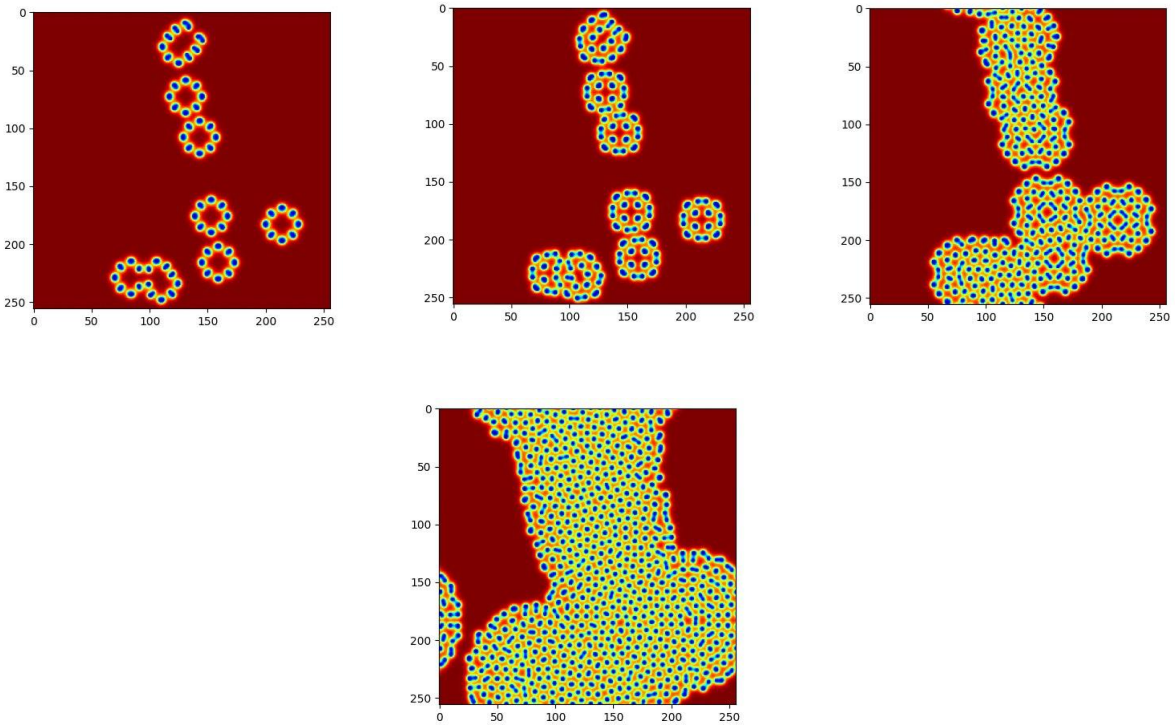
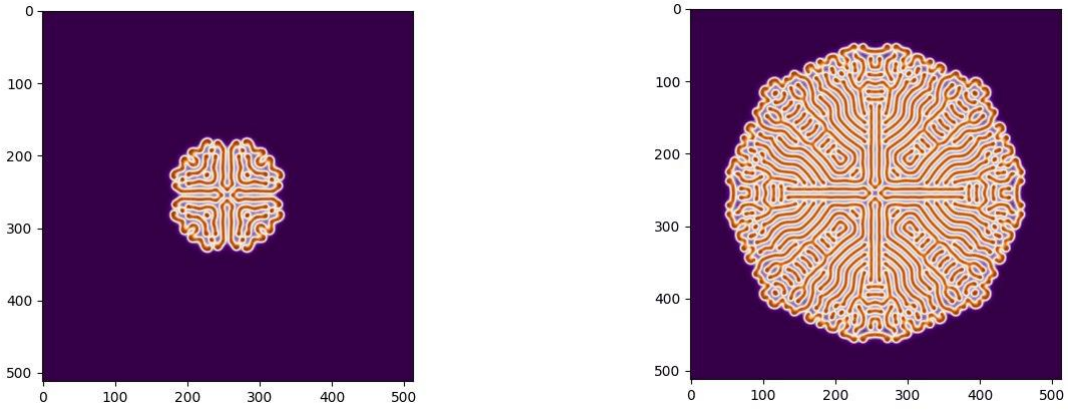
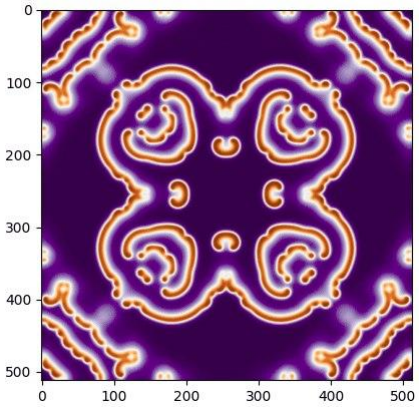
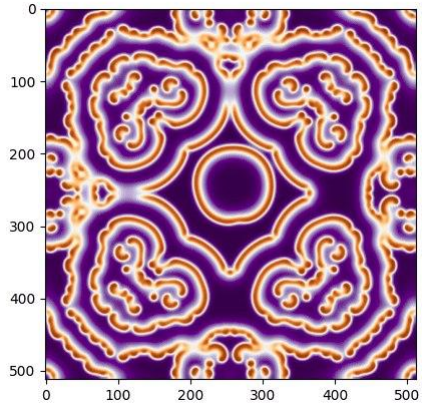


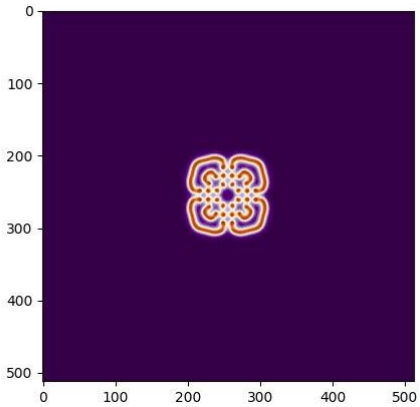
Figure 12: Cells that separate with f=0.03 | k = 0.062 | $D_u = 2$ | $dx = 2$ | $dt = 1$

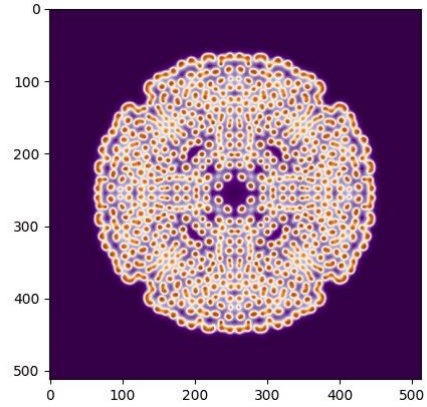
Now, if we stop randomizing the arrays and start with a reagent deposit in the center of the figure, what happens? Beautiful symmetrical figures appear. This is what the example of the three families (stripes, chaos and cells) looks like with those initial conditions on the arrays :











Part V
Ideas for improvement

6 Speed up

One of the major issue in this subject is the number of calculations and the size of the manipulated arrays. As explained before, Python was a good compromise. However, C remains way way much faster. It created at the beginning a little doubt on the choice of the language, since a test had been made under the same conditions between C and Python and the results were without appeal: less than **3 seconds** of execution time in C against **3 minutes** in Python.

Started but not finished due to lack of time:

- A program entirely in C including the calculation part (done) and the display part thanks to libraries like GTK 3 (in progress)
- Create a link between C (calculus) and Python (display) by either integrating C code into Python thanks to an API (in progress) or by stocking data generated in C and read them with Python in order to display them (done on one example, data were too big, not efficient)

7 Interactive mode

If we had the speed of calculation and a good optimization of the code, we could create a window with sliders for "kill" and "feed" to be able to change them live and observe the reactions. The ultimate goal would be this application: [1]
It would facilitate the mapping of values (kill, feed) to distinguish the families of pattern and perhaps reach a map as beautiful as this one created by someone working on the subject for many years : [2]

Part VI
Conclusion

Most of this problem was about finding the right initial conditions. This can be done through a qualitative or a mathematical analysis and that provides a great view of the problem. This proved to us that randomly changing the initial conditions was not the right way to find a solution.
We then found out that there are several families of patterns, depending on f and k carefully chosen as described before, some of them really reminding the mitosis.
Furthermore, another non-negligible factor is the initialization of the arrays : we found out that a random perturbation is way more efficient to create patterns than random arrays.
When everything would work out great, we could finally focus on optimizing the code, because of the power it would require. Actually, using auxiliary functions make the code more readable and more efficient.

Thank you for reading.

Part VII
References

[1] `https://pmneila.github.io/jsexp/grayscott/` : *"Reaction diffusion system (Gray-Scott model)"*. An stunningly fluid application by @pmneila which did help to find some families and is quite inspiring for a possible evolution of the project.

[2] `http://mrob.com/pub/comp/xmorphia/pearson-classes.html` : *Pearson's Classification (Extended) of Gray-Scott System Parameter Values*. His entire website is really interesting, considering the fact that he has been working on the subject for many years. Great reference to understand better the equations and the influence of feed and kill. Great cartography

[3] `https://matplotlib.org/stable/api/pyplot_summary.html` : The documentation of library used for displaying the images.

`https://github.com/neozhaoliang/pywonderland` A regroupment of insane creations. Way way bigger than our subject, fascinating tools.

[5] `https://github.com/nsbalbi/Gray-Scott-Reaction-Diffusion`. MATLAB code who did motivate us to try considering the diagonal neighbors in order to improve the code.

[6] `https://stackoverflow.com/` Mainly for secondary projects in C.