

La génération procédurale

La génération procédurale ou comment générer aléatoirement
un paysage réaliste homogène et cohérent

Paul PINEAU

June 11, 2018

Sommaire

- 1 Introduction
- 2 Générer la carte
- 3 Algorithme d'érosion
- 4 Conclusion
- 5 Bibliographie / Sitographie et annexes

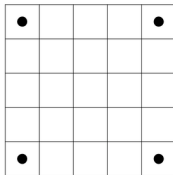
Introduction

Intérêts de la génération procédurale

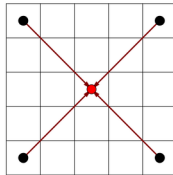
- 1 Variété des paysages créés
- 2 Optimisation de l'espace mémoire
- 3 Rapidité de création

Générer la carte

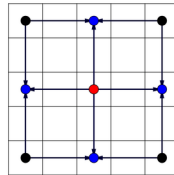
Algorithme du Diamant carré



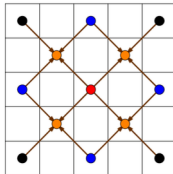
Initialisation des quatre coins



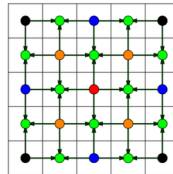
Phase du diamant avec un pas de 4



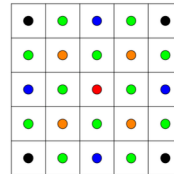
Phase du carré avec un pas de 4



Phase du diamant avec un pas de 2



Phase du carré avec un pas de 2



Matrice entièrement remplie

Figure: source : Wikipedia.fr

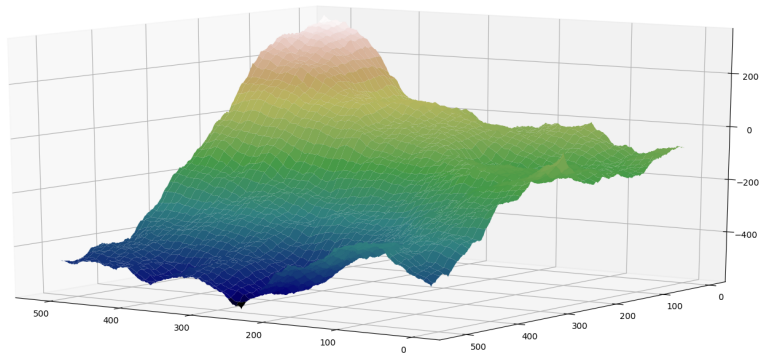


Figure: Génération par algorithme du Diamant Carré , carte 513×513

Génération par synthèse de Fourier

- 1 Génération d'une matrice de bruit
- 2 Passage au domaine fréquentiel
- 3 Atténuation des hautes fréquences
- 4 Retour dans le domaine réel



Figure: Paysage

Algorithme

1 Etape deux :
$$S(x, y) = \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} s(x, y) e^{-2i\pi(\frac{xk}{n} + \frac{yj}{n})}$$

2 Matrice amplitude $\rightarrow$ filtre en $\frac{1}{f^\beta}$

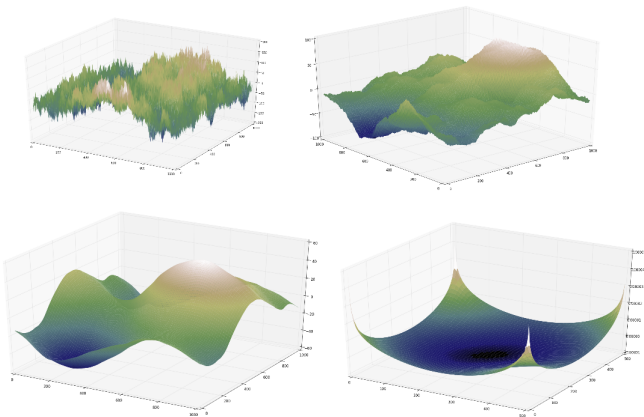
3 amplitude[x,y] $\leftarrow \sqrt{x^2 + y^2}^{-\beta}$

4 Transformation inverse :

$$s(x, y) = \frac{1}{n} \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} S(x, y) e^{2i\pi(\frac{xk}{n} + \frac{yj}{n})}$$

5 Complexité en $O(n^2)$

```
def Height_Map_FFT(f_hurst,n):  
    def regulateur(x, y):  
        if x == 0 and y == 0:  
            return 0.0  
        return f_hurst(np.sqrt(x**2 + y**2))  
  
    noise = np.fft.fft2(np.random.rand(n,n)) # $O(n \log(n))$   
    amplitude = np.zeros((n,n))  
  
    for i in range(n):  
        for j in range(n):  
            amplitude[i, j] = regulateur(i, j)  
  
    return np.fft.ifft2(noise * amplitude)
```



taille : 1000 , $\beta = 1.25; 1.9; 3$, matrice amplitude taille 500 $\beta = 2$

Algorithme d'érosion

Intérêt et principe (itération de t à $t + dt$)



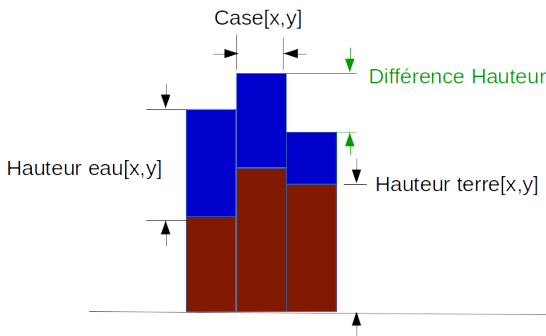
Source : <https://sciencing.com>

- 1 Pluie
- 2 Calcul des vecteurs flux
- 3 Déplacement des sédiments
- 4 Evaporation d'eau

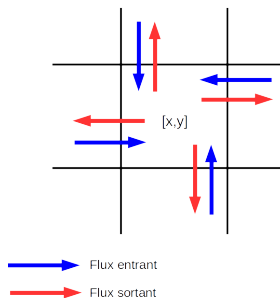
- 1 Eau : distribution aléatoire
- 2 Flux $F = (f^L, f^R, f^T, f^B) \text{ (m}^3.s^{-1}\text{)}$

$$\blacksquare f_{t+\Delta t}^i = K \cdot \max(0, f_t^i + \Delta t \cdot A \cdot \frac{g \cdot \Delta h^i[x, y]}{l}), i = L, R, T, B$$

$$\blacksquare K = \min(1, \frac{W2[x, y] \cdot l_x \cdot l_y}{(f^L + f^R + f^T + f^B) \cdot \Delta t})$$



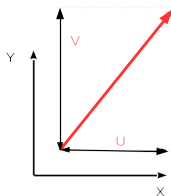
■ Volume d'eau à déplacer : $\Delta V = \Delta t.(\sum f_e - \sum f_s)$



■ Actualisation hauteur d'eau

■ Quantité moyenne d'eau selon x :

$$\Delta W_X = \frac{1}{2} \cdot (f^R(x-1, y) - f^L(x, y) + f^R(x, y) - f^L(x+1, y))$$

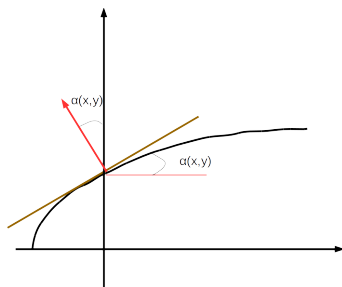


Vecteur vitesse / vélocité

■ Dédution de u : $u = \frac{2 \cdot \Delta W_X}{l_Y \cdot (W2[x,y] + W3[x,y])}$

■ Pareil sur Y

■ Capacité : $C[x, y] = K_C \cdot \sin(\alpha(x, y)) \cdot |\vec{v}(x, y)|$



Calcul de $\sin(\alpha(x, y))$:

```
dhx = (Height_map[x,y+1] - Height_map[x,y-1])/2
```

```
dhy = (Height_map[x-1,y] - Height_map[x+1,y])/2
```

```
angle[x,y] = np.sqrt(dhx ** 2 + dhy ** 2) / np.sqrt( 1 + dhx ** 2 + dhy ** 2 )
```

■ Actualisation carte de hauteur et carte de sédiment

■ $W[x, y] = W[x, y] \pm K_s(C[x, y] - s_t[x, y])$

■ $s1[x, y] = s_t[x, y] \mp K_s(C[x, y] - s_t[x, y])$

■ Déplacement de sédiments restants

$$\frac{\partial s}{\partial t} + (\vec{v} \cdot \nabla s) = 0$$

■ Résolution (discrétisation espace/temps) :

$$s_{t+\Delta t} = s_1(x - u \cdot \Delta t, y - v \cdot \Delta t)$$

■ Interpolation de $x - u \cdot \Delta t$ et $y - v \cdot \Delta t$

```
f = interpolate.interp2d(X,Y,s1,kind='linear')
```

```
Sediment_map[x,y] = f(x - u*dt, y - v*dt)[0]
```

Résultats

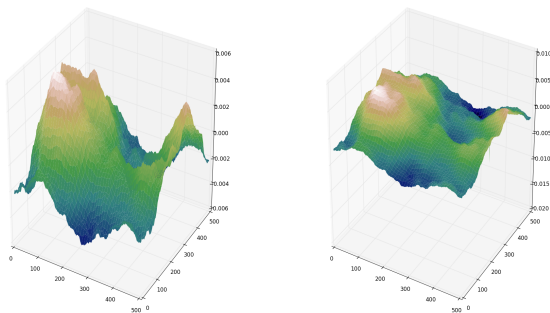


Figure: taille : 500 , $\beta = 1.75$, $n = 50$

Conclusion

- Diversité , Rapidité
- Contrôle limité (β)
- Algorithme d'érosion $\rightarrow$ coûteux
- Amélioration possible en ajoutant rivières, fleuves ou même végétations notamment par des L-sytèmes (système de Lindenmayer)

Bibliographie / Sitographie et annexes

Bibliographie

- 1 WIKIPEDIA, Algorithme Diamant-Carré, https://fr.wikipedia.org/wiki/Algorithme_Diamant-Carre, site consulté jusqu'à début décembre 2017
- 2 Keith Lantz, *Using Fourier synthesis to generate a fractional Brownian motion surface*, 19 Novembre 2011, <https://www.keithlantz.net/2011/11/using-fourier-synthesis-to-generate-a-fractional-brownian-motion-surface/>, site consulté régulièrement depuis janvier 2018
- 3 David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley *Texturing and Modeling , A Procedural Approach*, Chapitre 16, Morgan Kaufmann, 3 edition (16 Décembre 2002)
- 4 WIKIPEDIA , Pink Noise , https://en.wikipedia.org/wiki/Pink_noise, site consulté régulièrement depuis janvier 2018
- 5 Jacob Olsen *Realtime Procedural Terrain Generation*, 31 Octobre 2004, <http://web.mit.edu/cesium/Public/terrain.pdf>, site consulté régulièrement depuis décembre 2017
- 6 Xing Mei, Philippe Decaudin, Bao-Gang Hu, *Fast Hydraulic Erosion Simulation and Visualization on GPU*, 20 mars 2011

Logiciels utilisés

- 1 Idle3 (Python) , bibliothèque " matplotlib"
- 2 LibreOffice Draw , logiciel gratuit <https://fr.libreoffice.org/download/libreoffice-stable/>
- 3 Site Overleaf <https://www.overleaf.com/> pour $\text{\LaTeX}$

```
def diamant_carre(n):  
    m = 2 ** n + 1  
    matrice = np.zeros((m,m))  
    # valeur aléatoires pour les 4 coins  
    matrice[0,0] = rd.uniform(-m,m)  
    matrice[0,m-1] = rd.uniform(-m,m)  
    matrice[m-1,m-1] = rd.uniform(-m,m)  
    matrice[m-1,0] = rd.uniform(-m,m)  
  
    pas = m - 1  
    while pas > 1:  
        pas2 = pas // 2  
        # phase du diamant  
        for x in range(pas2 , m , pas ):  
            for y in range(pas2 , m , pas ):  
                #moy = (matrice[x - pas2 , y - pas2] + matrice[x - pas2 , y + pas2] +  
                #matrice[x + pas2 , y + pas2] + matrice[x + pas2 , y - pas2]) / 4  
                matrice[x,y] = moy + rd.uniform(-pas2,pas2)  
        #phase du carre  
        for x in range(0,m,pas2):  
            if x % pas == 0:  
                decalage = pas2  
            else:  
                decalage = 0  
  
        for y in range(decalage , m,pas):  
            somme = 0  
            i = 0  
            if x >= pas2:  
                somme += matrice[x - pas2, y]  
                i += 1
```

```
        if x + pas2 < m:
            somme += matrice[x + pas2, y]
            i +=1

        if y >= pas2 :
            somme += matrice[x , y - pas2]
            i +=1

        if y + pas2 < m:
            somme += matrice[x , y + pas2]
            i += 1

    matrice[x , y] = somme / i + rd.uniform(-pas2, pas2)

    pas = pas2
    return matrice

def FFT(n,hurst):
    plt.close()
    out = Height_Map_FFT(lambda k: k**(-hurst),n)
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    X = np.arange(1, n+1, 1)
    Y = np.arange(1, n+1, 1)
    X , Y = np.meshgrid( X , Y)
    Z = (out.real.reshape(X.shape))
    ax.plot_surface(X, Y, Z ,cmap = cm.gist_earth, linewidth = 0)
    plt.show()

def carte(taille,hurst):
```

```
'''renvoie une carte de hauteur generee par transformée de Fourier'''
Z = Height_Map_FFT(lambda k: k**(-hurst),taille)
Height_map = Z.real
return Height_map

def plot3d(carte_avant, carte_apres,taille):
    '''affiche la carte avant et apres modification par algorithme d'erosion'''
    X = np.arange(1, taille+1, 1)
    Y = np.arange(1, taille+1, 1)
    X , Y = np.meshgrid( X , Y)
    plt.close()
    fig = plt.figure()
    ax = fig.add_subplot(1,2,1, projection='3d')
    ax.plot_surface(X, Y, carte_avant ,cmap = cm.gist_earth, linewidth = 0)
    ax = fig.add_subplot(1,2,2, projection='3d')
    ax.plot_surface(X, Y, carte_apres ,cmap = cm.gist_earth, linewidth = 0)
    plt.show()

def Water_Map_func(n):
    ''' n : taille de la carte '''
    return np.zeros((n,n))

def Water_Map_func2(n,carte):
    ''' n : taille de la carte '''
    W = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            if carte[i,j] < 0 :
                W[i,j] = -carte[i,j]
    return W

def Sediment_Map_func(n):
```

```

    return np.zeros((n,n))

def Flux_Map_func(n):
    return np.array([[[[0,0,0,0]]*n]*n])

def pluie(Water_map,eau_max):
    carte_pluie = Water_map
    n = len(Water_map)
    for k in range(n):
        for j in range(n):
            carte_pluie[k,j] += (eau_max * np.random.random())
    return carte_pluie

def norme(vect):
    '''norme d'un vecteur x=(a,b)'''
    a,b = vect
    return np.sqrt(a ** 2 + b ** 2)

def Erosion_t_dt(taille,carte,A,l,lxy,eau,dt,Kc,Ks,Kd,Ke,n):#complexite O(n*taille*taille)
    '''algorithme d'erosion calculant la carte de hauteur apres erosion'''
    X = np.arange(1, taille+1, 1)
    Y = np.arange(1, taille+1, 1)
    #l,r,t,b
    Height_map = np.copy(carte)
    Water_map = Water_Map_func(taille)
    Water_map2 = Water_Map_func(taille)
    Water_map3 = Water_Map_func(taille)
    Sediment_map = Sediment_Map_func(taille)
    Flux_map = Flux_Map_func(taille)
    Velocity_field = np.array([[[[1.0,1.0]]*taille]*taille])
    angle = np.zeros((taille,taille))
    Capacite = np.zeros((taille,taille))
    s1 = Sediment_Map_func(taille)

    debut_barre()
    prog = 0 # pour la barre de progression

```

```

for k in range(n):

    Water_map2 = pluie(Water_map,eau)
    for x in range(taille):
        for y in range(taille):
            Flux = np.array([1.0,1.0,1.0,1.0])
            HW = Height_map[x,y] + Water_map2[x,y]

            #On calcule les vecteurs flux a partir des cases du haut, du bas , des cotes donc on differencie juste
            #les cas ou une des ces 4 cases n'existe pas

            if x == 0 :
                if y == 0 :
                    Flux[1] = max(0,Flux[1] + dt*A*(9.81*(HW - Height_map[0,1] - Water_map2[0,1]) / 1))
                    Flux[3] = max(0,Flux[3] + dt*A*(9.81*(HW - Height_map[1,0] - Water_map2[1,0]) / 1))

                elif y == taille - 1 :
                    Flux[0] = max(0,Flux[0] + dt*A*(9.81*(HW - Height_map[0,taille - 2] - Water_map2[0,taille - 2]) / 1))
                    Flux[3] = max(0,Flux[3] + dt*A*(9.81*(HW - Height_map[1,taille - 1] - Water_map2[1,taille - 1]) / 1))

                else :
                    Flux[0] = max(0,Flux[0] + dt*A*(9.81*(HW - Height_map[0,y-1] - Water_map2[0,y-1]) / 1))
                    Flux[1] = max(0,Flux[1] + dt*A*(9.81*(HW - Height_map[0,y+1] - Water_map2[0,y+1]) / 1))
                    Flux[3] = max(0,Flux[3] + dt*A*(9.81*(HW - Height_map[1,y] - Water_map2[1,y]) / 1))

            elif x == taille - 1 :
                if y == 0 :
                    Flux[1] = max(0,Flux[1] + dt*A*(9.81*(HW - Height_map[x,1] - Water_map2[x,1]) / 1))
                    Flux[2] = max(0,Flux[2] + dt*A*(9.81*(HW - Height_map[x-1,0] - Water_map2[x-1,0]) / 1))

                elif y == taille - 1 :
                    Flux[0] = max(0,Flux[0] + dt*A*(9.81*(HW - Height_map[x,y-1] - Water_map2[x,y-1]) / 1))
                    Flux[2] = max(0,Flux[2] + dt*A*(9.81*(HW - Height_map[x-1,y] - Water_map2[x-1,y]) / 1))

                else :
                    Flux[0] = max(0,Flux[0] + dt*A*(9.81*(HW - Height_map[x,y-1] - Water_map2[x,y-1]) / 1))
                    Flux[1] = max(0,Flux[1] + dt*A*(9.81*(HW - Height_map[x,y+1] - Water_map2[x,y+1]) / 1))
                    Flux[2] = max(0,Flux[2] + dt*A*(9.81*(HW - Height_map[x-1,y] - Water_map2[x-1,y]) / 1))

            elif y == 0 :
                Flux[1] = max(0,Flux[1] + dt*A*(9.81*(HW - Height_map[x,y+1] - Water_map2[x,y+1]) / 1))

```

```

Flux[2] = max(0,Flux[2] + dt*A*(9.81*(HW - Height_map[x-1,y] - Water_map2[x-1,y]) / 1))
Flux[3] = max(0,Flux[3] + dt*A*(9.81*(HW - Height_map[x+1,y] - Water_map2[x+1,y]) / 1))

elif y == taille - 1:
    Flux[0] = max(0,Flux[0] + dt*A*(9.81*(HW - Height_map[x,y-1] - Water_map2[x,y-1]) / 1))
    Flux[2] = max(0,Flux[2] + dt*A*(9.81*(HW - Height_map[x-1,y] - Water_map2[x-1,y]) / 1))
    Flux[3] = max(0,Flux[3] + dt*A*(9.81*(HW - Height_map[x+1,y] - Water_map2[x+1,y]) / 1))

else : #Cas general

    Flux[0] = max(0,Flux[0] + dt*A*(9.81*(HW - Height_map[x,y-1] - Water_map2[x,y-1]) / 1))
    Flux[1] = max(0,Flux[1] + dt*A*(9.81*(HW - Height_map[x,y+1] - Water_map2[x,y+1]) / 1))
    Flux[2] = max(0,Flux[2] + dt*A*(9.81*(HW - Height_map[x-1,y] - Water_map2[x-1,y]) / 1))
    Flux[3] = max(0,Flux[3] + dt*A*(9.81*(HW - Height_map[x+1,y] - Water_map2[x+1,y]) / 1))

    #Calcul de l'angle local d'inclinaison , utile apres

    dhx = (Height_map[x,y+1] - Height_map[x,y-1])/2
    dhy = (Height_map[x-1,y] - Height_map[x+1,y])/2
    angle[x,y] = np.sqrt(dhx ** 2 + dhy ** 2) / np.sqrt( 1 + dhx ** 2 + dhy ** 2 )

    #On ne peut pas enlever plus d'eau dans la case qu'il y en a
    if sum(Flux) != 0:
        K = min(1,(Water_map2[x,y] * lxy**2 )/(sum(Flux) * dt))
    else:
        K = 1
    Flux = [K*Flux[0],K*Flux[1],K*Flux[2],K*Flux[3]]

    Flux_map[x,y] = Flux

#Calcul nouveau niveau d'eau a partir des vecteurs flux
'''deltaV = dt*(somme flux_entrant - somme flux_sortant)'''
for x in range(taille):
    for y in range(taille):
        if x == 0 :
            if y == 0 :
                deltaV = dt*(Flux_map[x,y+1,0] + Flux_map[x+1,y,2] - sum(Flux_map[x,y]))
                deltaWX = Flux_map[x,y,1] - Flux_map[x,y+1,0]
                deltaWY = Flux_map[x,y,3] - Flux_map[x+1,y,2]
            elif y == taille - 1 :
                deltaV = dt*(Flux_map[x,y-1,1] + Flux_map[x+1,y,2] - sum(Flux_map[x,y]))
                deltaWX = Flux_map[x,y-1,1] - Flux_map[x,y,0]
                deltaWY = Flux_map[x,y,3] - Flux_map[x+1,y,2]

```

```

else :
    deltaV = dt*(Flux_map[x,y-1,1] + Flux_map[x+1,y,2] + Flux_map[x,y+1,0] - sum(Flux_map[x,y]))
    deltaWX = (Flux_map[x,y-1,1] - Flux_map[x,y,0] + Flux_map[x,y,1] - Flux_map[x,y+1,0]) / 2
    deltaWY = Flux_map[x,y,3] - Flux_map[x+1,y,2]

elif x == taille - 1 :
    if y == 0 :
        deltaV = dt*(Flux_map[x,y+1,0] + Flux_map[x-1,y,3] - sum(Flux_map[x,y]))
        deltaWX = Flux_map[x,y,1] - Flux_map[x,y+1,0]
        deltaWY = Flux_map[x-1,y,3] - Flux_map[x,y,2]

    elif y == taille - 1 :
        deltaV = dt*(Flux_map[x,y-1,1] + Flux_map[x-1,y,3] - sum(Flux_map[x,y]))
        deltaWX = Flux_map[x,y-1,1] - Flux_map[x,y,0]
        deltaWY = Flux_map[x-1,y,3] - Flux_map[x,y,2]

    else :
        deltaV = dt*(Flux_map[x,y-1,1] + Flux_map[x,y+1,0] + Flux_map[x-1,y,3] - sum(Flux_map[x,y]))
        deltaWX = (Flux_map[x,y-1,1] - Flux_map[x,y,0] + Flux_map[x,y,1] - Flux_map[x,y+1,0]) / 2
        deltaWY = Flux_map[x-1,y,3] - Flux_map[x,y,2]

elif y == 0 :
    deltaV = dt*(Flux_map[x,y+1,0] + Flux_map[x+1,y,2] + Flux_map[x-1,y,3] - sum(Flux_map[x,y]))
    deltaWX = Flux_map[x,y,1] - Flux_map[x,y+1,0]
    deltaWY = (Flux_map[x-1,y,3] - Flux_map[x,y,2] + Flux_map[x,y,3] - Flux_map[x+1,y,2])/2

elif y == taille - 1 :
    deltaV = dt * (Flux_map[x,y-1,1] + Flux_map[x+1,y,2] + Flux_map[x-1,y,3] - sum(Flux_map[x,y]))
    deltaWX = Flux_map[x,y-1,1] - Flux_map[x,y,0]
    deltaWY = (Flux_map[x-1,y,3] - Flux_map[x,y,2] + Flux_map[x,y,3] - Flux_map[x+1,y,2])/2

else :
    deltaV = dt * (Flux_map[x,y+1,0] + Flux_map[x,y-1,1] + Flux_map[x+1,y,2] + Flux_map[x-1,y,3]
    deltaV -= sum(Flux_map[x,y]))
    deltaWX = (Flux_map[x,y-1,1] - Flux_map[x,y,0] + Flux_map[x,y,1] - Flux_map[x,y+1,0]) / 2
    deltaWY = (Flux_map[x-1,y,3] - Flux_map[x,y,2] + Flux_map[x,y,3] - Flux_map[x+1,y,2])/2

Water_map3[x,y] = Water_map2[x,y] + deltaV/(lxy**2)

```

```

d = (Water_map[x,y] + Water_map2[x,y]) / 2

if (lxy*d) != 0:
    u = deltaWX / (lxy * d)
    v = deltaWY / (lxy * d)
else:
    u = 0
    v = 0

Velocity_field[x,y] = [u,v]
Capacite[x,y] = Kc * angle[x,y] * norme(Velocity_field[x,y])
#print(Capacite[x,y] - Sediment_map[x,y])
if Capacite[x,y] > Sediment_map[x,y]:
    Height_map[x,y] -= Ks * (Capacite[x,y] - Sediment_map[x,y])
    s1[x,y] = Sediment_map[x,y] + Ks * (Capacite[x,y] - Sediment_map[x,y])
    #Water_map3[x,y] = Water_map2[x,y] + dt * Ks * (Capacite[x,y] - Sediment_map[x,y])
else:
    Height_map[x,y] += Kd * (Sediment_map[x,y] - Capacite[x,y])
    s1[x,y] = Sediment_map[x,y] - Kd * (Sediment_map[x,y] - Capacite[x,y])
    #Water_map3[x,y] = Water_map2[x,y] - dt * Kd * (Sediment_map[x,y] - Capacite[x,y])

#Equation d'advection --> actualisation de la carte sediment

f = interpolate.interp2d(X,Y,s1,kind='linear')
for x in range(taille):
    for y in range(taille):
        u,v = Velocity_field[x,y][0] ,Velocity_field[x,y][1]
        a = x - u * dt
        b = y - v * dt

        Sediment_map[x,y] = f(a,b)[0]
Water_map[x,y] = Water_map3[x,y] * (1 - K0 * dt)

# Affichage de la barre de progression
if k > prog:
    avance_barre()
    prog += n / 50

fin_barre()
return Height_map

```