

# Les bruits pseudo-aléatoires

December 2, 2022

## Binome:

- PAUL Thomas
- RIBEIRO Zack

## Objectifs projet :

- Développer des bruits (2D et 3D) : Perlin(FBM), Worley(Cellular)
- Présentation de chacun des bruits avec possibilités / outils.
- Génération de shaders/textures/VFX avec les bruits (comme de l'eau avec Worley, Terrain avec Perlin ou via FBM(Perlin Fractal))

## Plan(Chapitres):

- Introduction - **Bruit de “Value”**
- Démonstration des possibilités avec le **bruit de Perlin 2D**
- Version **3D** du bruit de **Perlin**
- **Application** du bruit pour une **déformation de surface**
- **Application** du bruit de **Perlin** : Champ vec / **Art numérique et Optimisation**
- Utilisations des **transformés de Fourier** pour les bruits
- **Application** du bruit de **Perlin** : **Domain Warping**
- Tentative de simulation d'**érosion** (Non possible car optimisation impossible via GPU sur jupyter et taille trop faible pour avoir une bonne circulation de l'eau, voir : <https://hal.inria.fr/inria-00402079/document> pour + de détails).
- Bruit de **Worley**
- **Conclusion**

## Notes:

- Le PDF final est le résultat de plusieurs PDF mélangés en un (donc plusieurs notebooks : facilite le travail et l'organisation).
- Certaines images ont des problèmes de taille liés à un rendu via LaTeX pour être compatible avec l'export : “pdf via LaTeX” sur jupyter => Ne pas faire attention aux potentiels espaces entre images.
- La lecture en vue sur deux pages est conseillée. En effet le code et les images peuvent prendre beaucoup de place,i.e le nombre de pages du PDF peut sembler être conséquent mais ce n'est pas réellement le cas.

## 1 Introduction - Bruit de “Value”

Le bruit produit par les méthodes randoms sont non cohérentes dans l'espace, c'est à dire que pour  $n(x,y)$ , la valeur du bruit en  $(x,y)$ , il n'y devrait pas y avoir corrélation entre le choix de cette valeur est celles aux alentours. Ils sont totalement aléatoires.

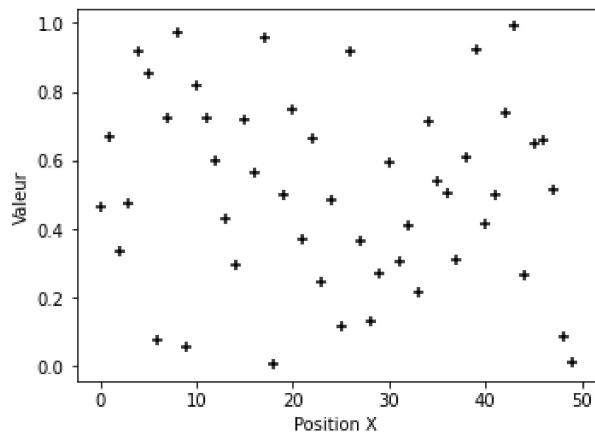
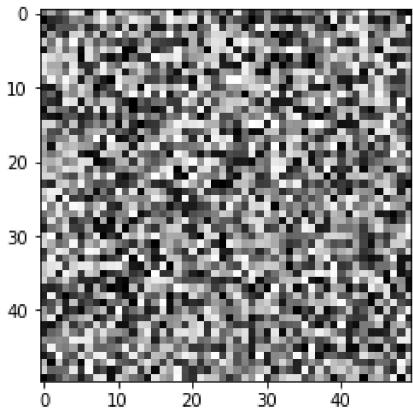
```
[1]: #On importe les outils dont on a besoin
import numpy as np
import matplotlib.pyplot as plt

#taille de la matrice donc de l'espace
shape = 50

#On génère la matrice aléatoire
random_numpy = np.random.rand(shape, shape)

#Rendu de la matrice via matplotlib
fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].imshow(random_numpy, cmap='gray')

#On rend une ligne/colonne de la matrice pour voir l'aleatoire
ax[1].scatter(range(0,shape),random_numpy[0], color="black", marker='+')
plt.xlabel("Position X")
plt.ylabel("Valeur")
plt.show()
```



On cherche donc un moyen de produire un bruit **cohérent (pseudo-aléatoire)** dans l'espace. Notre première solution est d'**interpoler cette matrice aléatoire** afin qu'un point de l'espace intéragisse avec un autre point proche et donc que la valeur aléatoire d'un point  $(x,y)$  soit influencée par les valeurs des points voisins et donc devient pseudo-aléatoire.

```
[2]: from scipy import interpolate

interpolate_noise = random_numpy

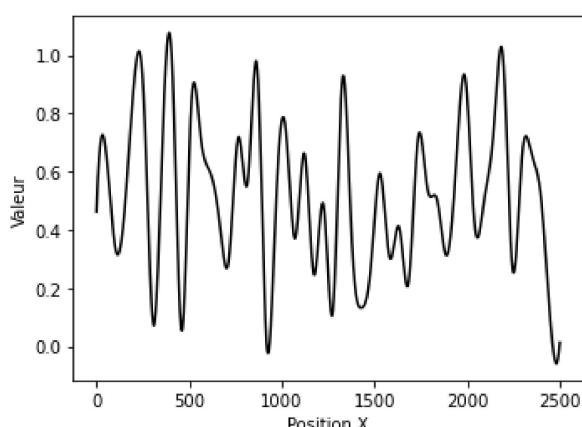
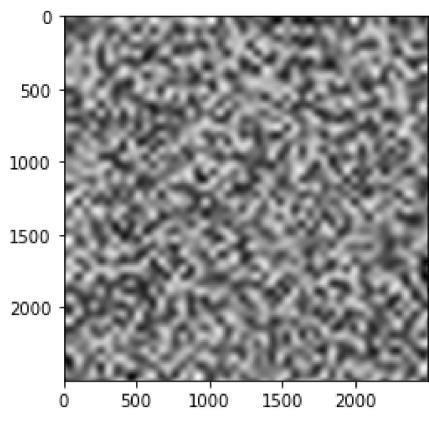
#On applique l'interpolation, on obtient donc une fonction qui approxime notre bruit aléatoire
f = interpolate.interp2d(np.linspace(0,shape), np.linspace(0,shape), interpolate_noise, kind='cubic')

#Le but de l'interpolation est de calculer les points entre x et x+1 donc entre deux voisins. Donc ici on veut 50 points entre 2 voisins
res_lerp = 50

#On applique la fonction d'interpolation pour obtenir la nouvelle matrice
interpolate_noise = f(np.linspace(0,shape,shape*res_lerp),np.linspace(0,shape,shape*res_lerp))

#Rendu de la matrice via matplotlib
fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].imshow(interpolate_noise, cmap='gray')

#On rend une ligne/colonne de la matrice pour voir l'aléatoire
ax[1].plot(interpolate_noise[0], color="black")
plt.xlabel("Position X")
plt.ylabel("Valeur")
plt.show()
```

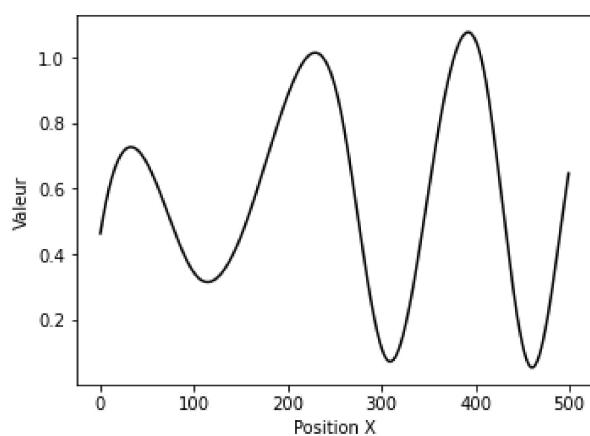
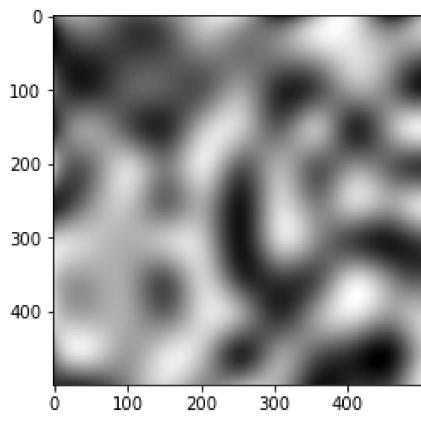


Le graphe de points qui monte la **relation entre la valeur et la position X** met en évidence l'**interpolation** entre plusieurs voisins et ainsi **les nouveaux points sont pseudo-aléatoires** car ils dépendent de la position et des valeurs voisines. Si on zoom donc sur le bruit on peut avoir un premier beau bruit pseudo-aléatoire.

```
[3]: #On zoom pour avoir {border_size} points de longueur et largeur
border_size = 500

#Rendu d'une partie de la matrice via matplotlib
fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].imshow(interpolate_noise[:border_size,:border_size], cmap='gray')

#On rend une ligne/colonne d'une partie de la matrice pour voir l'aléatoire
ax[1].plot(interpolate_noise[0][:border_size], color="black")
plt.xlabel("Position X")
plt.ylabel("Valeur")
plt.show()
```



Néanmoins ce **bruit a plusieurs problèmes** dont le principal qui est qu'on va **apercevoir des formes rectangulaires** (correspondant à nos valeurs de départ) assez facilement et que donc lors de l'utilisation on aura un pattern non voulu et **pas très esthétique**. Si on génère une montagne avec un bruit de valeurs on va obtenir un terrain très redondant/sans caractère qui ne fait pas naturel.

## 2 Démonstration des possibilités avec le bruit de Perlin 2D

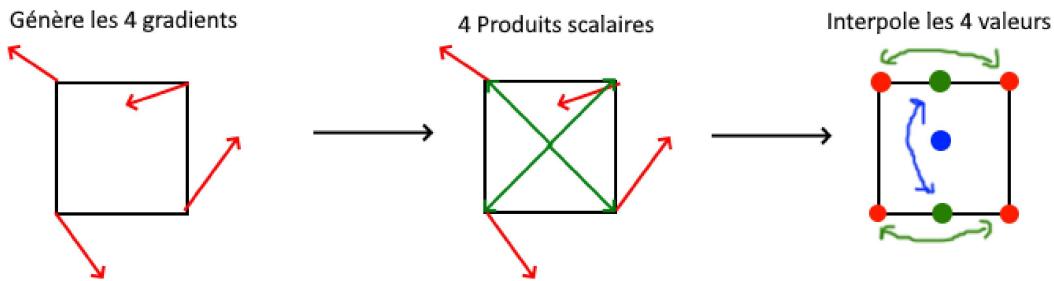
On cherche donc un moyen de produire un nouveau bruit cohérent (pseudo-aléatoire) dans l'espace qui n'a pas les défauts du bruit de valeurs. Une solution encore beaucoup utilisée (la version simplex(2001, aussi par Ken Perlin) avec moins d'artefacts (pour des dimensions supérieures : 4D, 5D) est le bruit de **Ken Perlin** (1985). Il est assez simple de le développer.

### 2.1 Génération de différents bruits en partant d'une même matrice aléatoire

Tous nos exemples ici seront issus d'une **même matrice aléatoire** (celle utilisée plus haut) afin de pouvoir comparer les différentes opérations dessus. A la fin de ce chapitre nous organiserons le bruit de Perlin dans un **module** afin d'être réutilisé dans d'autres chapitres avec un simple “import”.

Afin de générer un bruit de Perlin: - Commencer par créer une grille d'une certaine résolution(taille de chaque cellule) qui correspondra à notre fréquence finale. - Générer 1 gradient aléatoire pour chaque sommet de la grille (donc 1 cellule aura 4 gradients en 2D). - Faire le produit scalaire entre chaque gradient et de tous les vecteurs des pixels composant la cellule (donc 4 produits scalaires pour chaque pixel de la cellule). - Interpoler les valeurs obtenues : 3 interpolations pour le cas 2D (2 à 2 jusqu'à n'avoir plus qu'une seule valeur par pixel). - Sauvegarder la valeur obtenue dans une matrice (ou tenseur).

Pour une cellule, les opérations faites peuvent se résumer en : (exemple pour le pixel du centre)



```
[4]: import numpy as np

#based on https://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf ; https://fr.wikipedia.org/wiki/Bruit_de_Perlin

def lerp(t):
    return 6*t**5 - 15*t**4 + 10*t**3

def perlin2D(shape, res):
    delta = (res[0] / shape[0], res[1] / shape[1])
    d = (shape[0] // res[0], shape[1] // res[1])
    # Define grid
    grid = np.mgrid[0:res[0]:delta[0], 0:res[1]:delta[1]].transpose(1, 2, 0) % 1
    # Random Gradients
    #random_numpy = np.random.rand(res[0]+1, res[1]+1)
    angles = 2*np.pi*np.random.uniform(0:res[0]+1, 0:res[1]+1)
    gradients = np.dstack((np.cos(angles), np.sin(angles)))
    g00 = gradients[0:-1, 0:-1].repeat(d[0], 0).repeat(d[1], 1)
    g10 = gradients[1:, 0:-1].repeat(d[0], 0).repeat(d[1], 1)
    g01 = gradients[0:-1, 1:].repeat(d[0], 0).repeat(d[1], 1)
    g11 = gradients[1:, 1: ].repeat(d[0], 0).repeat(d[1], 1)
    # Ramps in simplex version(here) / Scalar products in first version
    n00 = np.sum(grid * g00, 2)
    n10 = np.sum(np.dstack((grid[:, :, 0]-1, grid[:, :, 1])) * g10, 2)
    n01 = np.sum(np.dstack((grid[:, :, 0], grid[:, :, 1]-1)) * g01, 2)
    n11 = np.sum(np.dstack((grid[:, :, 0]-1, grid[:, :, 1]-1)) * g11, 2)
    # Interpolation
```

```

t = lerp(grid)
n0 = n00*(1-t[:, :, 0]) + t[:, :, 0]*n10
n1 = n01*(1-t[:, :, 0]) + t[:, :, 0]*n11

#Normalization
return (np.sqrt(2)*((1-t[:, :, 1])*n0 + t[:, :, 1]*n1) + 1)/2

```

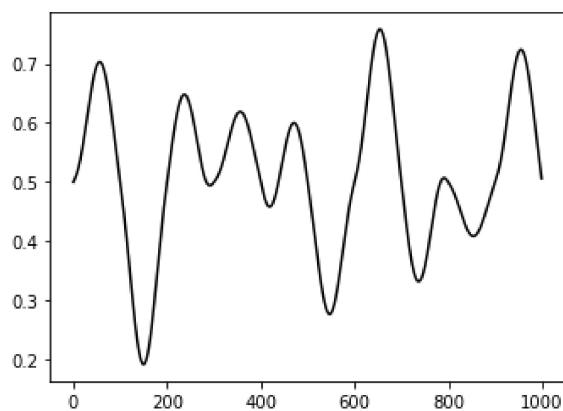
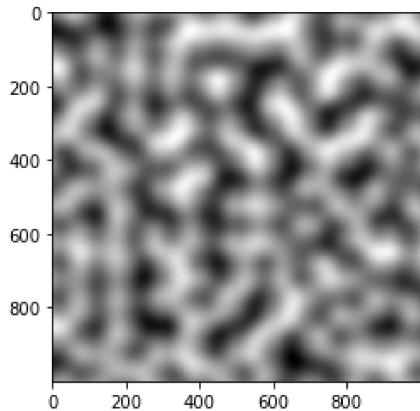
[5]: import matplotlib.pyplot as plt

```

#Rendu 2D
pic = perlin2D([1000,1000], [10,10])
fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].imshow(pic, cmap='gray')
ax[1].plot(pic[0], color="black")

```

[5]: [`<matplotlib.lines.Line2D at 0x7fe7aa2aeeb0>`]



On observe un **bruit continu** et qui n'a plus la forme "carré/rectangulaire" du bruit de "Value" ainsi que **moins d'artefacts/d'anomalies**. Ce bruit est ainsi plus utilisable et modulable.

Si on souhaite avoir un bruit plus détaillé on peut faire une **somme de bruits avec différentes fréquences et une amplitude décroissante**. On nomme ce bruit : "**fractal noise**".(ça sera expliqué à la fin du projet pourquoi ce nom).

```

[6]: def fractal2D(shape, res, octaves=1, persistance=0.5, exponentiation=1,
    ↪ lacunarity=1, norm=True, noise = []):
    #Si jamais on n'a pas mis de bruit en argument de la fonction, on crée une
    ↪ matrice
    if(len(noise) == 0):
        noise = np.zeros(shape)
    frequency = 1
    amplitude = 1
    #Tous les paramètres sont expliqués plus bas

```

```

for _ in range(octaves):
    noise += amplitude * perlin2D(shape, (frequency*res[0], frequency*res[1]))
    frequency *= lacunarity
    amplitude *= persistance
    noise = noise**exponentiation
#On normalise le bruit
return noise/noise.max() if(norm) else noise

```

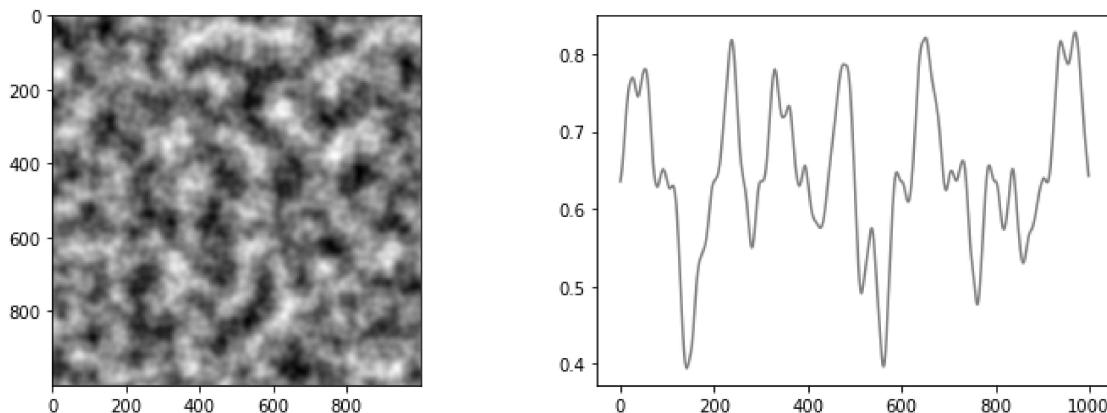
[7]: #Rendu d'un bruit fractal 2D

```

pic_fractal = fractal2D([1000,1000], [10,10], octaves=3, exponentiation=1, lacunarity=2)
fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].imshow(pic_fractal, cmap='gray')
ax[1].plot(pic_fractal[0], color="gray")

```

[7]: [<matplotlib.lines.Line2D at 0x7fe7b23e4b80>]



## 2.2 Effets des paramètres :

Faisons un rapide tour des paramètres présents dans la fonction “fractal”.

### 2.2.1 Fréquence

Les bruits visualisés jusqu’à maintenant et plus tard ont une fréquence qui est identique pour x,y et ensuite z. Néanmoins on peut donner une **fréquence différente** entre les 2(ou 3) axes. Cela mène à un “étirement” d’un des axes où la fréquence est plus faible.

### 2.2.2 Amplitude

L’**amplitude** est la **valeur maximale** du bruit.

### 2.2.3 Octave, Lacunarity & Persistance

Bruit à N octaves : Somme de N bruits s'exprimant comme : fractal =  $\sum_{i=0}^N A_i \times \text{perlin}(f_i)$  avec: - La fréquence du bruit i est  $f_i = (f_0) \times \text{lacunarity}^i$  - L'amplitude du bruit i est  $A_i = (A_0) \times \text{persistance}^i$

```
[8]: #On compare plusieurs bruits(de même matrice initiale/Le bruit de Perlin) de
      →nombre d'octaves différents en appliquant la fonction ci-dessus en changeant
      →seulement le nombre d'octaves

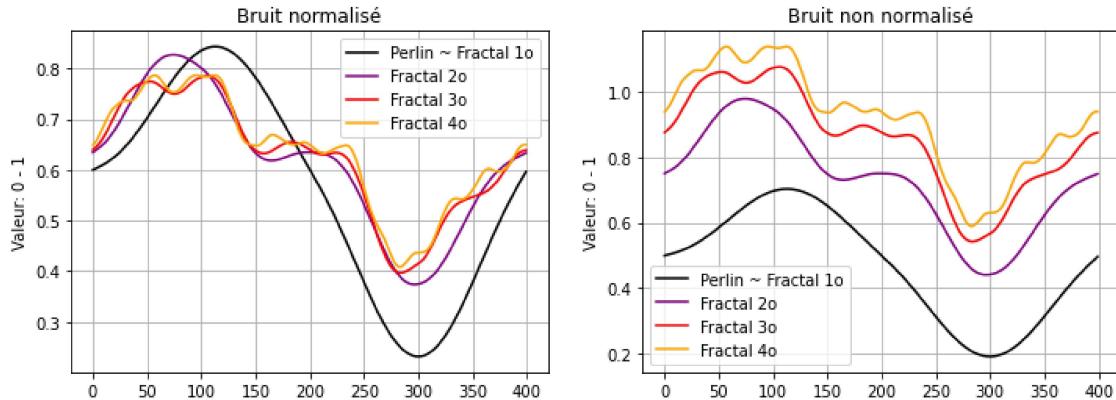
size_res = 2
size_shap = 400 * int(np.ceil(size_res/10))

fig, ax = plt.subplots(1,2, figsize=(12,4))
#Normalisés
ax[0].plot(fractal2D([size_shap,size_shap], [size_res,size_res], octaves=1, u
      →exponentiation=1, lacunarity=2, norm=True)[0], label='Perlin ~ Fractal 1o', u
      →color="black")
ax[0].plot(fractal2D([size_shap,size_shap], [size_res,size_res], octaves=2, u
      →exponentiation=1, lacunarity=2, norm=True)[0], label='Fractal 2o', u
      →color="purple")
ax[0].plot(fractal2D([size_shap,size_shap], [size_res,size_res], octaves=3, u
      →exponentiation=1, lacunarity=2, norm=True)[0], label='Fractal 3o', u
      →color="red")
ax[0].plot(fractal2D([size_shap,size_shap], [size_res,size_res], octaves=4, u
      →exponentiation=1, lacunarity=2, norm=True)[0], label='Fractal 4o', u
      →color="orange")
ax[0].set_ylabel('Valeur: 0 - 1')
ax[0].set_title('Bruit normalisé')
ax[0].grid()
ax[0].legend()

#Non normalisés
ax[1].plot(fractal2D([size_shap,size_shap], [size_res,size_res], octaves=1, u
      →exponentiation=1, lacunarity=2, norm=False)[0], label='Perlin ~ Fractal 1o', u
      →color="black")
ax[1].plot(fractal2D([size_shap,size_shap], [size_res,size_res], octaves=2, u
      →exponentiation=1, lacunarity=2, norm=False)[0], label='Fractal 2o', u
      →color="purple")
ax[1].plot(fractal2D([size_shap,size_shap], [size_res,size_res], octaves=3, u
      →exponentiation=1, lacunarity=2, norm=False)[0], label='Fractal 3o', u
      →color="red")
ax[1].plot(fractal2D([size_shap,size_shap], [size_res,size_res], octaves=4, u
      →exponentiation=1, lacunarity=2, norm=False)[0], label='Fractal 4o', u
      →color="orange")
ax[1].set_ylabel('Valeur: 0 - 1')
ax[1].set_title('Bruit non normalisé')
ax[1].grid()
```

```
ax[1].legend()
```

[8]: <matplotlib.legend.Legend at 0x7fe7a756ecd0>



Pour bien comprendre nous utiliserons le cas non normalisé(deuxième graphe) car il permet de mettre en évidence la somme de plusieurs bruits(exemple : rouge = noir + violet + un autre bruit plus faible). - Imaginons que notre première octave(noir) soit la forme grossière de notre terrain. - Et qu'ensuite nous rajoutons des gros rochers donc à plus petite échelle (c'est à dire plus haute fréquence) et moins haut que nos collines/montagnes(c'est à dire à amplitude plus faible) nous obtenons donc le bruit violet = Le terrain + Les rochers. - Maintenant on rajoute des arbres qui sont plus petites que des rochers ainsi le bruit rouge = Le terrain + Les rochers + Les arbres. - Et pour finir on met des cailloux qui sont donc plus petit que des arbres, etc etc. Ainsi le bruit jaune = Le terrain + Les rochers + Les arbres + Les cailloux. On a donc un terrain plus détaillé.

#### 2.2.4 Exponentiation

Permet d'augmenter ou de réduire le **contraste** : `result = fractalexponentiation`

### 2.3 Divers exemples d'opérations sur le bruit

Ce bruit qui est cohérent et continu dans l'espace est à la base de beaucoup de choses, il est très facile d'avoir des résultats très différents les un des autres et qui sont utiles à la **génération de texture, VFX...** (On n'utilisera plus que le fractal noise qui est donc la somme de plusieurs perlin = permet d'avoir des détails). En général un artiste **utilise plusieurs bruits différents sur lesquels il applique des opérations/ des transformations avant de les combiner** deux à deux (par exemple : un bruit pour une montagne qu'il va rajouter à un bruit de dunes...).

La comparaison est souvent faite entre un bruit et une série de sinus afin d'expérimenter de visualiser des opérations simples rapidement.

[9]: #On copie notre méthode pour faire un bruit fractal mais cette fois-ci on  
→ utilise un sinus plutôt que le bruit.  
`def sinus_somme(n, res, octaves=1, persistance=0.5, exponentiation=1,  
→ lacunarity=1, norm=True):`

```

x = np.linspace(0,n,n)
noise = np.linspace(0,0,n)
frequency = 1
amplitude = 1
for _ in range(octaves):
    noise += amplitude * np.sin(frequency*res*x / (n*(np.pi/5)))
    frequency *= lacunarity
    amplitude *= persistance
noise = noise**exponentiation
return noise/noise.max() if(norm) else noise

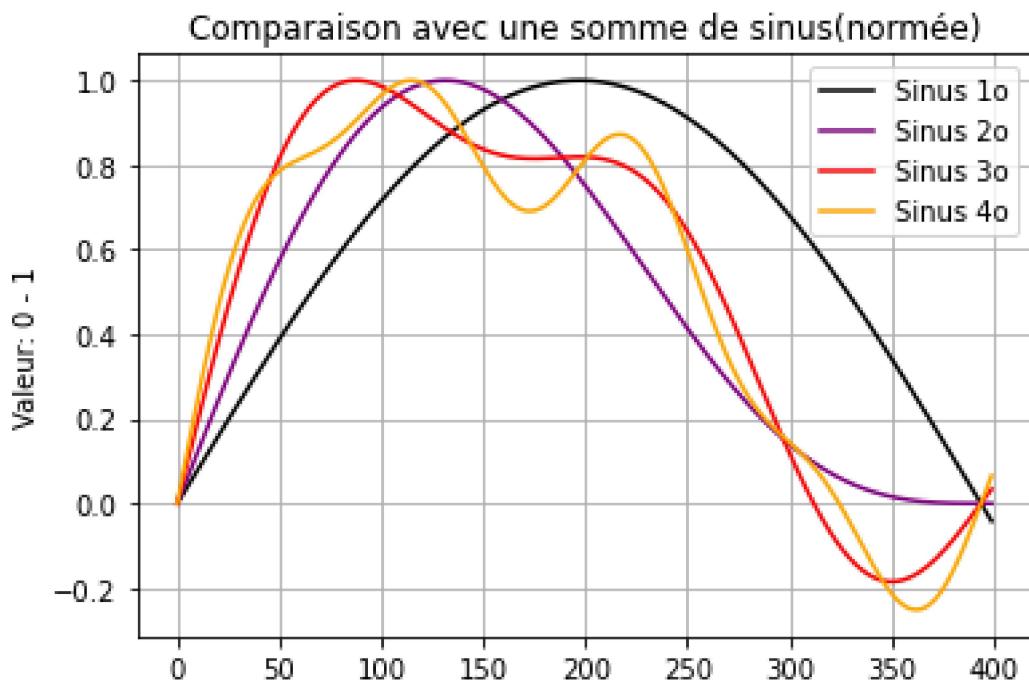
```

[10]: #On affiche plusieurs sommes de sinus pour vérifier le principe des octaves.

```

plt.plot(sinus_somme(size_shap, size_res, octaves=1, exponentiation=1,
                     →lacunarity=2, norm=True), label='Sinus 1o', color="black")
plt.plot(sinus_somme(size_shap, size_res, octaves=2, exponentiation=1,
                     →lacunarity=2, norm=True), label='Sinus 2o', color="purple")
plt.plot(sinus_somme(size_shap, size_res, octaves=3, exponentiation=1,
                     →lacunarity=2, norm=True), label='Sinus 3o', color="red")
plt.plot(sinus_somme(size_shap, size_res, octaves=4, exponentiation=1,
                     →lacunarity=2, norm=True), label='Sinus 4o', color="orange")
plt.ylabel('Valeur: 0 - 1')
plt.title('Comparaison avec une somme de sinus(normée)')
plt.grid()
plt.legend()
plt.show()

```

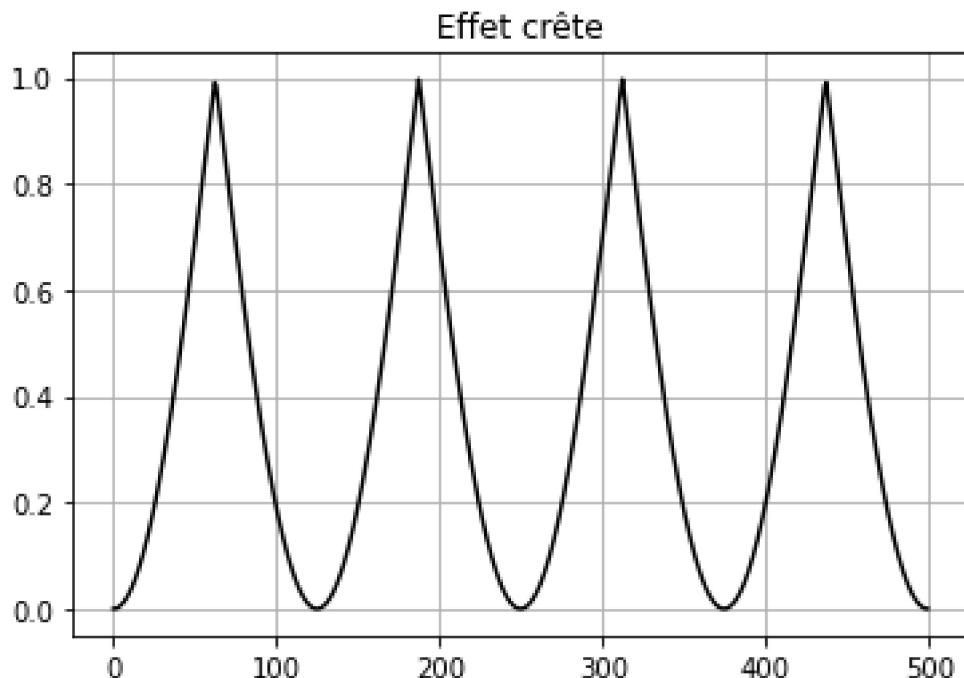


### 2.3.1 Ridge / Turbulent

Une opération très utilisée et simple est l'**effet “crête”(ridge) ou encore “turbulent”**. Elle est utile pour donner un look de montagnes à notre bruit de Perlin. Il suffit de faire l'opération suivante pour l'obtenir :  $n(x, y) = -1 \times |n_0(x, y)| + 1$  où  $n$  est notre bruit qui dépend de l'espace, ici en 2D. Le +1 est pour garder la normalisation du bruit donc qu'il soit entre 0 et 1 si  $n_0$  est déjà normalisée initialement. - “Turbulent” : Si on somme seulement les valeurs absolues. - “Ridge” : Si on fait l'inverse du bruit turbulent(\* -1).

[11]: *#Pour visualiser on applique l'opération sur un seul sinus/cosinus.*

```
x = np.linspace(0,np.pi*4,500)
plt.plot(-1 * np.abs(np.cos(x)) + 1, color="black")
plt.title('Effet crête')
plt.grid()
plt.show()
```



[12]: *#Notre bruit turbulent n'a de différence que dans la somme des octaves, donc le principe reste évidemment le même.*

```
def turbulent2D(shape, res, octaves=1, persistance=0.5, exponentiation=1, lacunarity=1, noise = []):
    if(len(noise) == 0):
        noise = np.zeros(shape)
    frequency = 1
```

```

amplitude = 1
for _ in range(octaves):
    #Notre bruit étant normalisé entre 0 et 1 sa valeur absolu ne donnera aucun résultat. On le place donc entre -0.5 et 0.5 afin que la valeur absolu donne bien un résultat différent que sans. (On peut très bien jouer sur le 0.5 et le changer)
    noise += np.abs(amplitude * perlin2D(shape, (frequency*res[0], frequency*res[1]))-0.5)
    frequency *= lacunarity
    amplitude *= persistance
    noise = noise**exponentiation
return noise / noise.max()

```

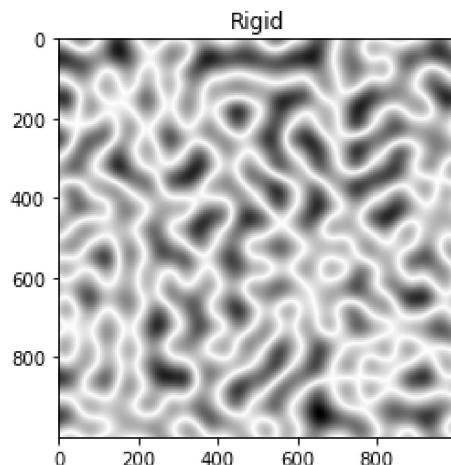
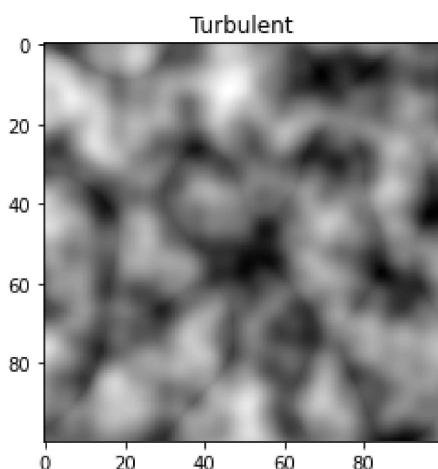
[13]: #On crée les deux bruits

```

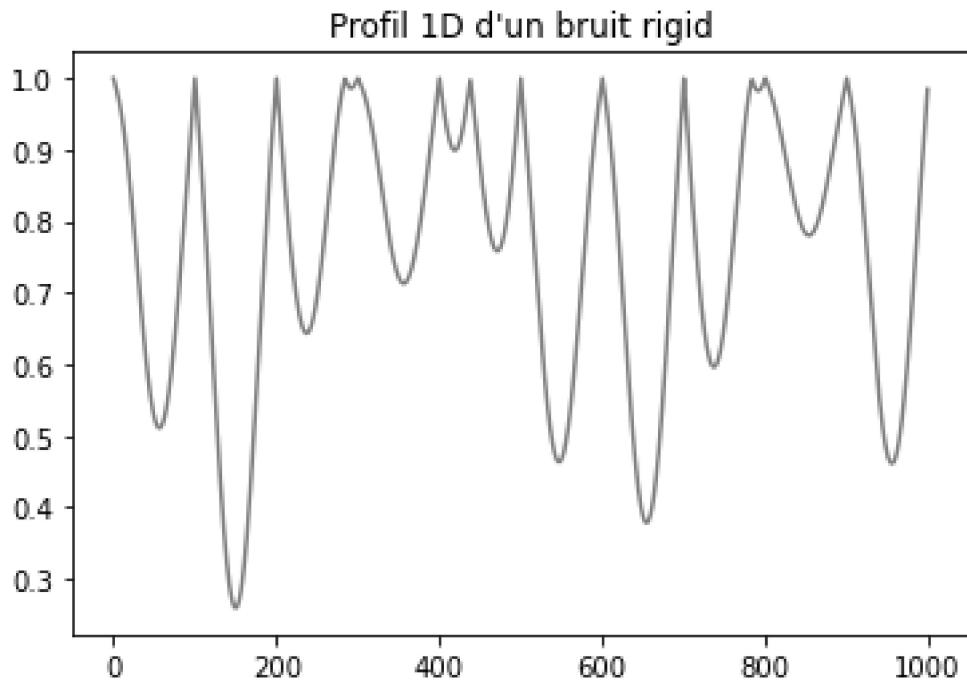
pic_turbulent = turbulent2D([320,320], [10,10], octaves=3, persistance= 0.5, exponentiation=0.5, lacunarity=2)
pic_rigid = -1*turbulent2D([1000,1000], [10,10], octaves=1, persistance= 0.5, exponentiation=1, lacunarity=2) +1

#On fait le rendu
fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].imshow(pic_turbulent[:100,:100], cmap='gray')
ax[0].set_title("Turbulent")
ax[1].imshow(pic_rigid, cmap='gray')
ax[1].set_title("Rigid")
plt.show()
plt.plot(pic_rigid[0], color="gray")
plt.title("Profil 1D d'un bruit rigid")

```



[13]: Text(0.5, 1.0, "Profil 1D d'un bruit rigid")



### 2.3.2 Max

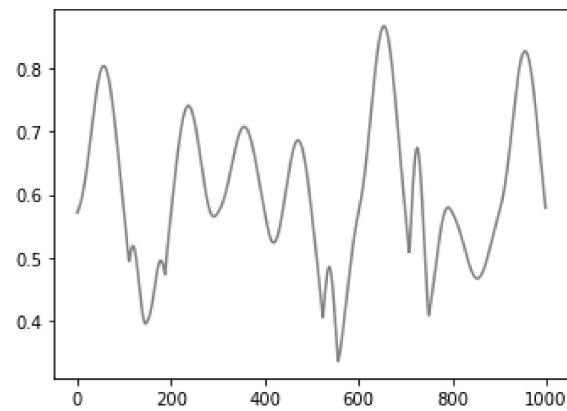
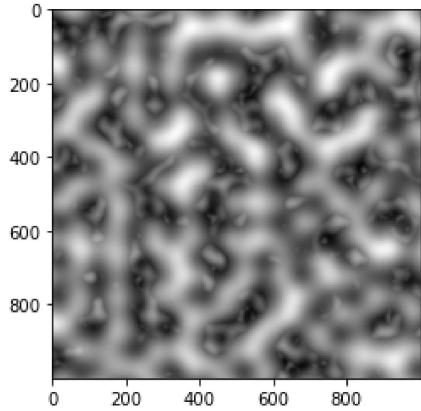
Si on veut maintenant produire un résultat très différent il nous suffit par exemple de **sommer le maximum** entre le bruit et chaque élément de la somme.

```
[14]: def maxperlin2D(shape, res, octaves=1, persistance=0.5, exponentiation=1, lacunarity=1, noise = []):
    if(len(noise) == 0):
        noise = np.zeros(shape)
    frequency = 1
    amplitude = 1
    for _ in range(octaves):
        #On fait le maximum avec numpy
        noise = np.maximum(noise,amplitude * perlin2D(shape, (frequency*res[0], frequency*res[1])))
        frequency *= lacunarity
        amplitude *= persistance
    noise = noise**exponentiation
    return noise / noise.max()
```

```
[15]: #Rendu
fig, ax = plt.subplots(1,2, figsize=(12,4))
pic_max = maxperlin2D([1000,1000], [10,10], octaves=3, persistance= 0.7, exponentiation=1, lacunarity=2)
```

```
ax[0].imshow(pic_max, cmap='gray')
ax[1].plot(pic_max[0], color="gray")
```

[15]: [<matplotlib.lines.Line2D at 0x7fe7a7225d90>]



### 2.3.3 Lerp

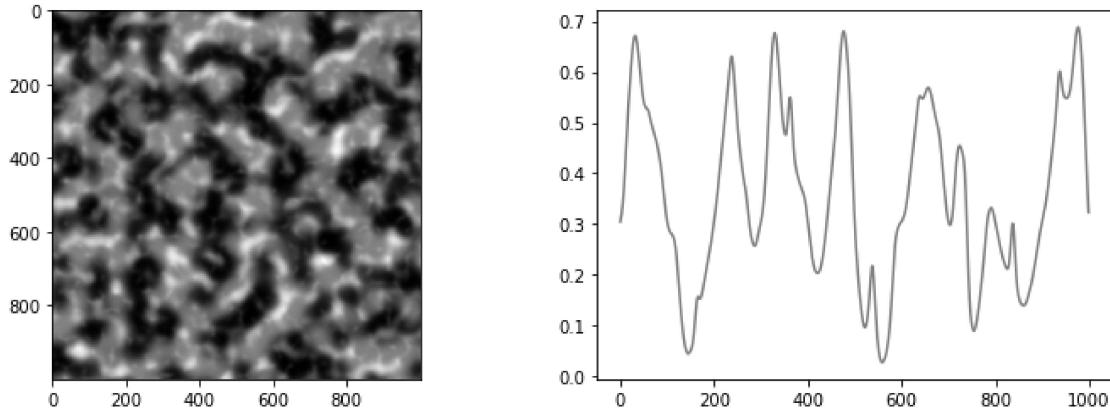
Peut être utilisé de divers manières pour smooth/adoucir le noise/bruit ou perdre un peu de détails.

```
[16]: def lerpperlin2D(shape, res, octaves=1, persistance=0.5, exponentiation=1, lacunarity=1, noise = []):
    if(len(noise) == 0):
        noise = np.zeros(shape)
    frequency = 1
    amplitude = 1
    for _ in range(octaves):
        #On applique le polynome d'interpolation de Perlin plusieurs fois.
        noise += lerp(lerp(amplitude * perlin2D(shape, (frequency*res[0], frequency*res[1]))))
        frequency *= lacunarity
        amplitude *= persistance
    noise = noise**exponentiation
    return noise / noise.max()
```

```
[17]: #Bruit
pic_lerp = lerpperlin2D([1000,1000], [10,10], octaves=3, persistance= 0.7, exponentiation=1, lacunarity=2)

#Rendu
fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].imshow(pic_lerp, cmap='gray')
ax[1].plot(pic_lerp[0], color="gray")
```

```
[17]: [<matplotlib.lines.Line2D at 0x7fe7a716d3d0>]
```



### 2.3.4 Round / Floor

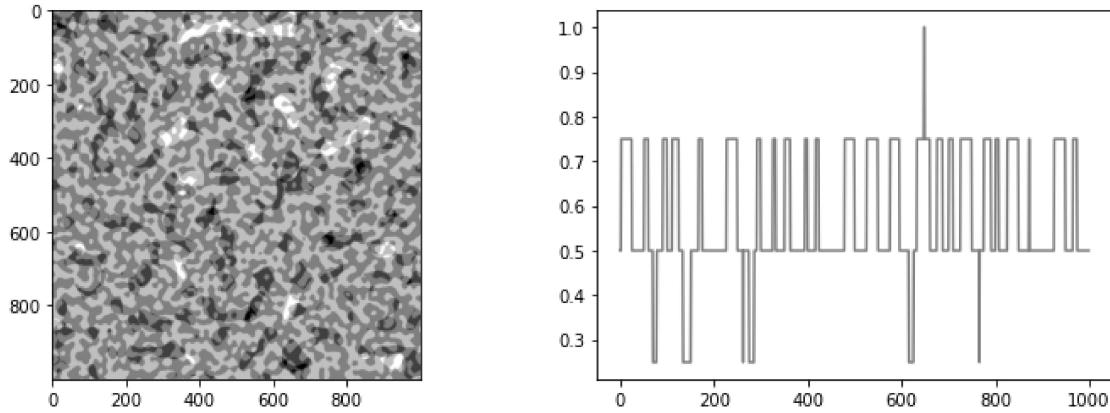
Permet de faire des plateaux, utile par exemple si on veut générer un canyon ou quelque chose de stratifiée.

```
[18]: #Nombre de plateau pour dec = 1 ~= plateau * 10^dec +- 2
def roundperlin2D(shape, res, octaves=1, persistance=0.5, exponentiation=1, u
→lacunarity=1, dec = 2, plateau = 1, noise = []):
    if(len(noise) == 0):
        noise = np.zeros(shape)
    frequency = 1
    amplitude = 1
    for _ in range(octaves):
        #On approxime à une certaine décimale chaque élément de la somme du
→bruit
        noise += np.round(amplitude * perlin2D(shape, (frequency*res[0], u
→frequency*res[1])) * plateau,dec) / plateau
        frequency *= lacunarity
        amplitude *= persistance
    noise = noise**exponentiation
    return noise / noise.max()
```

```
[19]: #Bruit
pic_round = roundperlin2D([1000,1000], [10,10], octaves=3, persistance= 0.7, u
→exponentiation=1, lacunarity=2, dec = 1, plateau = 0.2)

#Rendu
fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].imshow(pic_round, cmap='gray')
ax[1].plot(pic_round[0], color="gray")
```

[19]: [<matplotlib.lines.Line2D at 0x7fe7a73a9b20>]



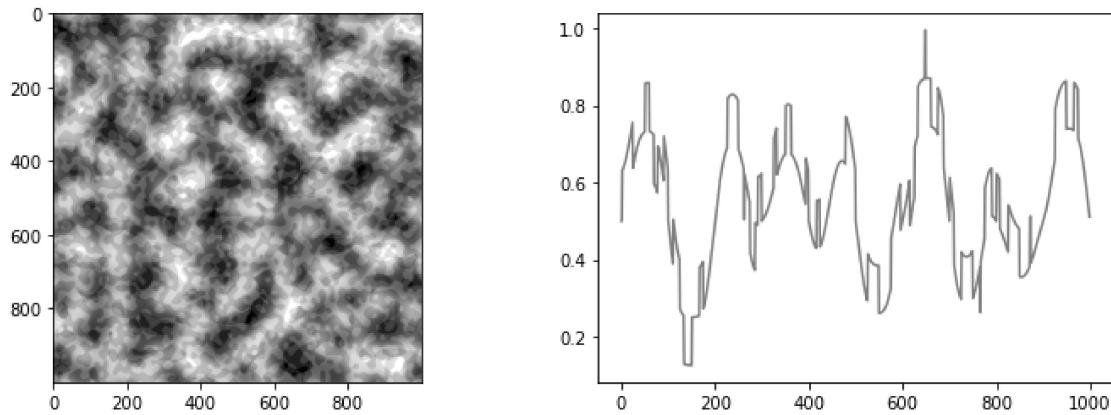
### 2.3.5 Combinaison

Le but est évidemment de combiner plusieurs opérations simples pour former un bruit plus complexe. - Lerp 1 octave du round au dessus : (On voit qu'en effet ça fait des transitions entre les plateaux + adoucies).

```
[20]: #Bruit
pic_combi = lerpperlin2D([1000,1000], [10,10], octaves=1, persistance= 0.7, ↪
    ↪exponentiation=1, lacunarity=2, noise =
        roundperlin2D([1000,1000], [10,10], octaves=3, persistance= 0. ↪
    ↪7, exponentiation=1, lacunarity=2, dec = 1, plateau = 0.2)
    )

#Rendu
fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].imshow(pic_combi, cmap='gray')
ax[1].plot(pic_combi[0], color="gray")
```

[20]: [<matplotlib.lines.Line2D at 0x7fe7a74b2790>]



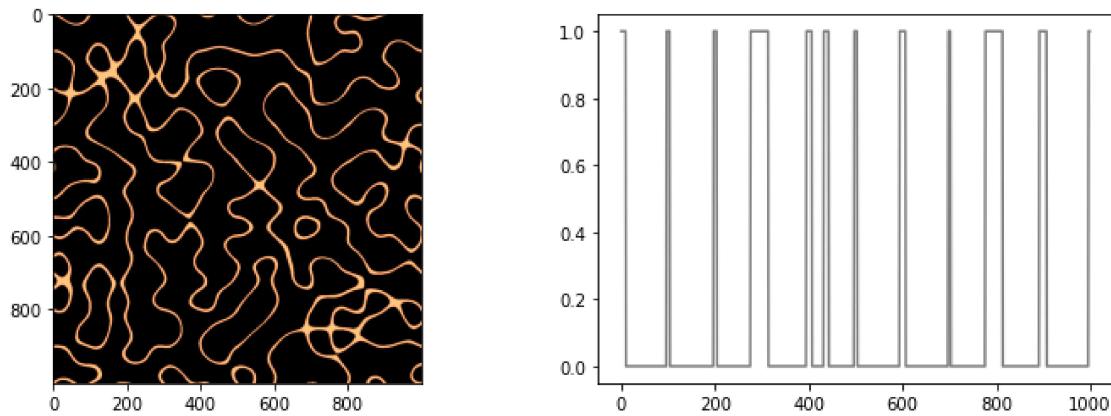
### 2.3.6 Strings

On peut appliquer une valeur minimum à notre bruit “turbulent” afin de ne garder qu’une matrice de 0 ou de 1, 1 si la valeur remplit la condition.

```
[21]: #Bruit avec la condition
pic_strings = ((-1 *turbulent2D([1000,1000], [10,10], octaves=1, persistance= 0.
    ↪7, exponentiation=1, lacunarity=2)+1) > 0.95)

#Rendu
fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].imshow(pic_strings, cmap='copper')
ax[1].plot(pic_strings[0], color="gray")
```

```
[21]: [<matplotlib.lines.Line2D at 0x7fe7a8e70f70>]
```



### 2.3.7 Curl

Il est possible d'aller encore plus loin en faisant la rotationnel(= curl) du bruit (ça donnera donc un champ vectoriel et non scalaire). Ainsi en 2D:  $\text{curl}(n(x, y)) = \left(\frac{\partial n}{\partial x}, -\frac{\partial n}{\partial y}\right)$ . Il permet de générer des écoulements : <https://www.cs.ubc.ca/~rbridson/docs/bridson-siggraph2007-curlnoise.pdf>

```
[47]: import perlin2D as perlin
import numpy as np
import matplotlib.pyplot as plt

shape = 2**9
res = 2**2
noise = perlin.Perlin2D([shape,shape],[res,res]).noise

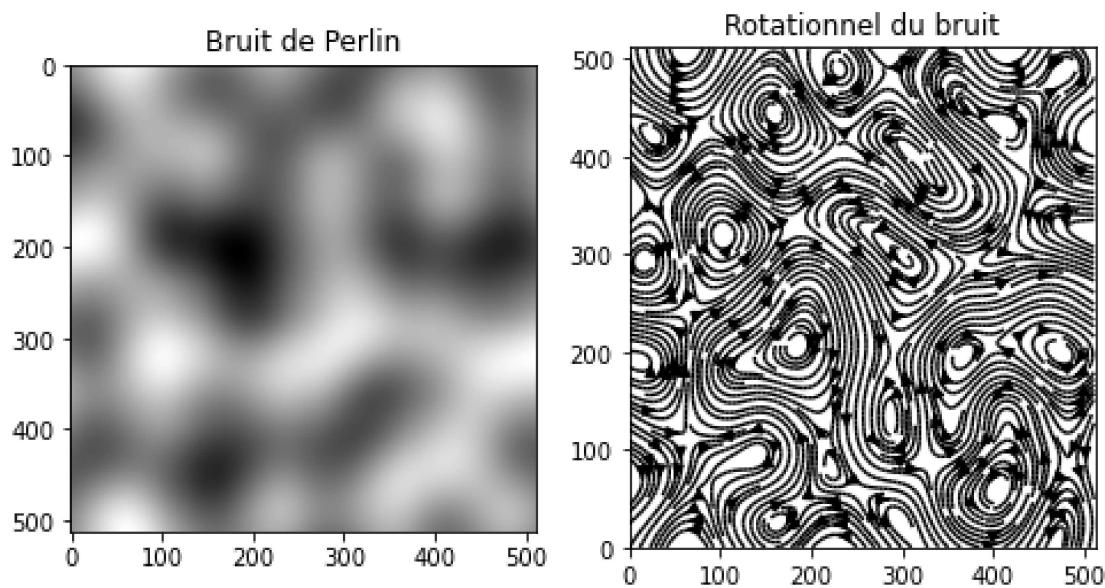
#Gradient du bruit
grad = np.gradient(noise)

#Construction du champ vectoriel
field = np.zeros([shape,shape,2])
for i in range(len(noise)):
    for j in range(len(noise[i])):
        #Def de la rotationnel en 2D
        field[i][j][0] = grad[0][i][j]
        field[i][j][1] = -grad[1][i][j]
field = np.asarray(field)

#Rendu
fig, ax = plt.subplots(1,2,figsize=(8,4))

ax[1].axis([0, shape, 0, shape])
ax[1].streamplot(np.arange(0,shape,1),np.arange(0,shape,1),field[:, :, 0],field[:, :, 1], density = 3, color="black")
ax[1].set_title("Rotationnel du bruit")

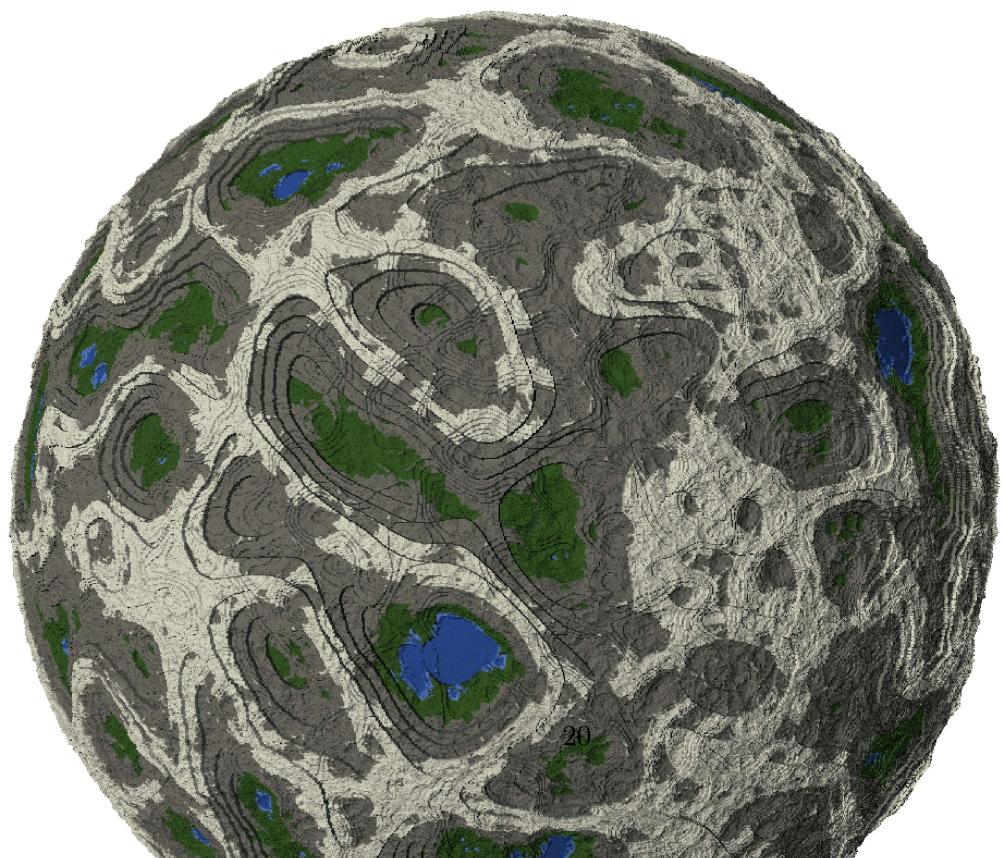
ax[0].imshow(noise, cmap="gray")
ax[0].set_title("Bruit de Perlin")
plt.show()
```

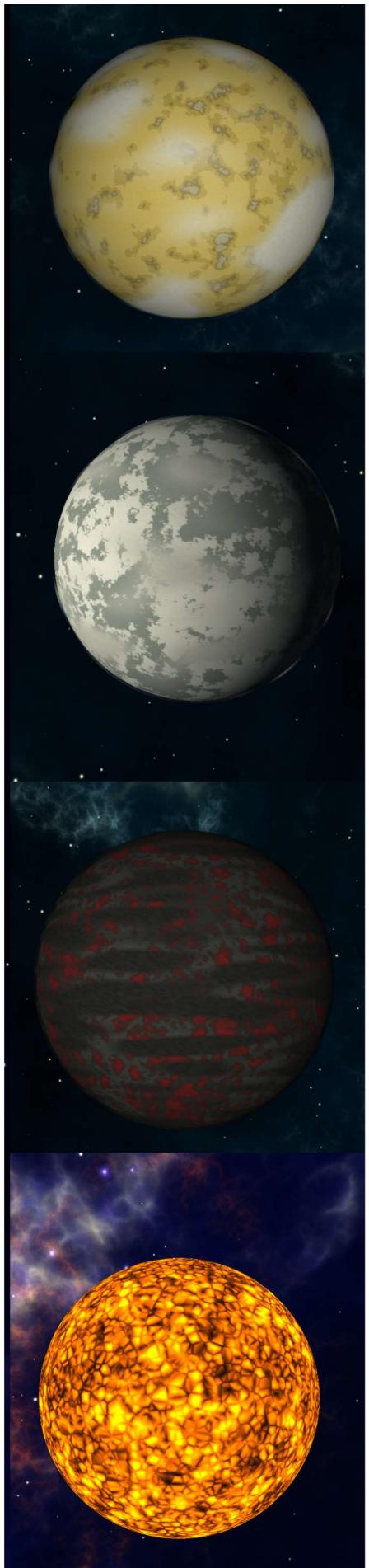


## 2.4 Exemples

### 2.4.1 Sur une sphère

**Dev en Javascript (Ridge - Rendu Threejs et Round):** - 15 Octaves, - exp  $\approx 0.5$  - persistance  $\approx 1.6$  - fréquence = 10 - lacunarity = 3





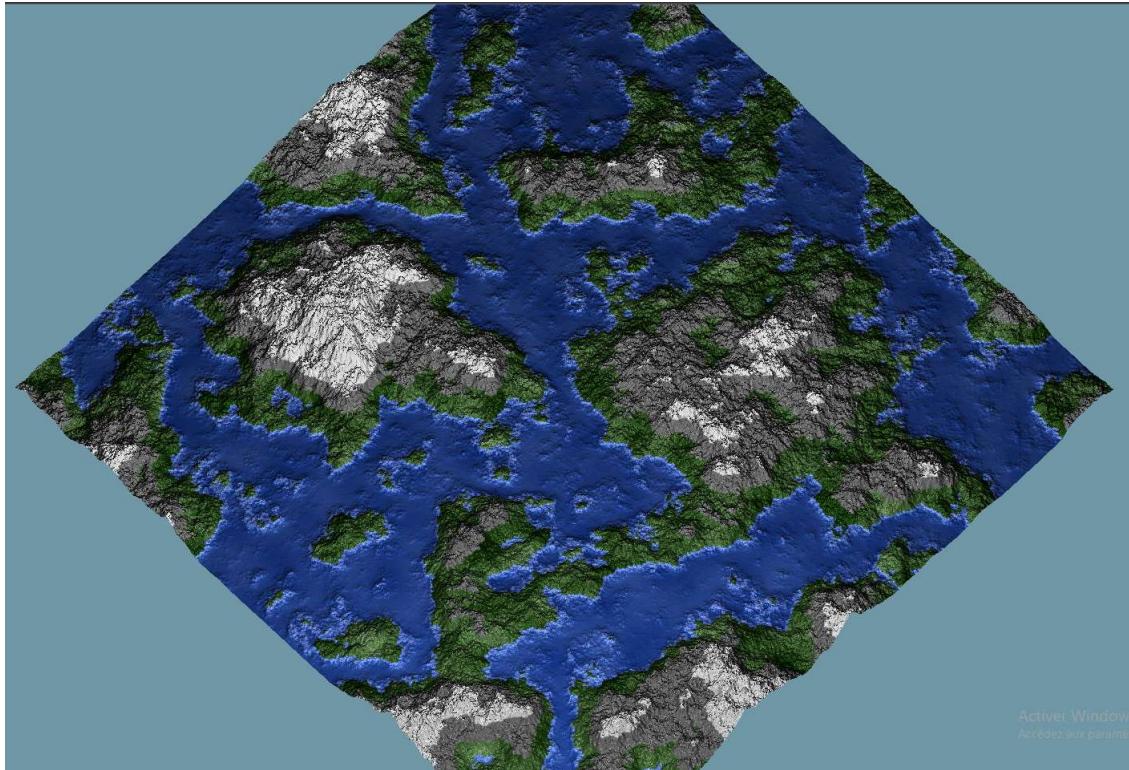
22

Anciens exemples sans relief en JS:

Le dernier est via le bruit de worley non perlin

#### 2.4.2 Sur un plan

Dev en Javascript (Rendu Threejs / Caméra isométrique(Angles conservés)):



### 3 Version 3D du bruit de Perlin

Une 3 ème dimension permet de soit: - Générer un espace (nul besoin pour les déformations de surface dans l'espace comme un plan ou une sphère comme ci-dessus) ; ex : Grottes en forme d'éponge. - Avoir 2 axes d'espace et 1 axe du temps afin ainsi d'animer la texture/déformation. Exemple : Eau (nous le verrons avec Worley qui est plus adapté pour de l'eau).

On peut aussi étendre en 4D afin d'avoir 3 axes d'espace et 1 axe du temps afin d'animer notre “éponge”.

Remarque : On diminue grandement la “résolution” du bruit afin d'animer la 3D “rapidement”. L'astuce pour détourner ce problème pour la 2D est de se servir de l'interpolation disponible par matplotlib.

#### 3.1 Génération de différents bruits

On revoit les algorithmes des différentes versions du bruit de Perlin dans l'ordre.

### 3.1.1 Base

Commençons par adapter donc notre algorithme de génération du bruit de Perlin **2D à la 3D**. Il n'est pas difficile de l'adapter si on connaît bien l'algorithme de la 2D. En effet à la place d'avoir **4 gradients aléatoires** pour une cellule de la grille, il y en aura **8 = les 8 sommets** d'un cube. Donc on applique les **produits scalaires et l'interpolation à ces 8 gradients** plutôt qu'à 4. Le polynome d'**interpolation ne change pas** entre la 2D et la 3D.

```
[22]: def perlin3D(shape, res, tileable=(False, False, False), lerp=lerp):
    delta = (res[0] / shape[0], res[1] / shape[1], res[2] / shape[2])
    d = (shape[0] // res[0], shape[1] // res[1], shape[2] // res[2])
    grid = np.mgrid[0:res[0]:delta[0], 0:res[1]:delta[1], 0:res[2]:delta[2]]
    grid = np.mgrid[0:res[0]:delta[0], 0:res[1]:delta[1], 0:res[2]:delta[2]]
    grid = grid.transpose(1, 2, 3, 0) % 1
    # Gradients
    theta = 2 * np.pi * np.random.rand(res[0] + 1, res[1] + 1, res[2] + 1)
    phi = 2 * np.pi * np.random.rand(res[0] + 1, res[1] + 1, res[2] + 1)
    gradients = np.stack((np.sin(phi) * np.cos(theta), np.sin(phi) * np.sin(theta),
    ↪np.cos(phi)), axis=3)
    if tileable[0]:
        gradients[-1, :, :] = gradients[0, :, :]
    if tileable[1]:
        gradients[:, -1, :] = gradients[:, 0, :]
    if tileable[2]:
        gradients[:, :, -1] = gradients[:, :, 0]
    gradients = gradients.repeat(d[0], 0).repeat(d[1], 1).repeat(d[2], 2)
    g000 = gradients[:, :-d[0], :, :-d[1], :, :-d[2]]
    g100 = gradients[d[0]:, :, :-d[1], :, :-d[2]]
    g010 = gradients[:, :-d[0], d[1]:, :, :-d[2]]
    g110 = gradients[d[0]:, :, d[1]:, :, :-d[2]]
    g001 = gradients[:, :-d[0], :, :-d[1], d[2]:]
    g101 = gradients[d[0]:, :, :-d[1], d[2]:]
    g011 = gradients[:, :-d[0], d[1]:, :, d[2]:]
    g111 = gradients[d[0]:, d[1]:, :, d[2]:]
    # Ramps
    n000 = np.sum(np.stack((grid[:, :, :, 0], grid[:, :, :, 1], grid[:, :, :, 2]), ↪axis=3) * g000, 3)
    n100 = np.sum(np.stack((grid[:, :, :, 0]-1, grid[:, :, :, 1], grid[:, :, :, 2]), ↪axis=3) * g100, 3)
    n010 = np.sum(np.stack((grid[:, :, :, 0], grid[:, :, :, 1]-1, grid[:, :, :, 2]), ↪axis=3) * g010, 3)
    n110 = np.sum(np.stack((grid[:, :, :, 0]-1, grid[:, :, :, 1]-1, grid[:, :, :, 2]), ↪axis=3) * g110, 3)
    n001 = np.sum(np.stack((grid[:, :, :, 0], grid[:, :, :, 1], grid[:, :, :, 2]-1), ↪axis=3) * g001, 3)
    n101 = np.sum(np.stack((grid[:, :, :, 0]-1, grid[:, :, :, 1], grid[:, :, :, 2]-1), ↪axis=3) * g101, 3)
```

```

n011 = np.sum(np.stack((grid[:, :, :, 0] , grid[:, :, :, 1]-1, grid[:, :, :, 2]-1),axis=3) * g011, 3)
n111 = np.sum(np.stack((grid[:, :, :, 0]-1, grid[:, :, :, 1]-1, grid[:, :, :, 2]-1),axis=3) * g111, 3)
# Interpolation
t = lerp(grid)
n00 = n000*(1-t[:, :, :, 0]) + t[:, :, :, 0]*n100
n10 = n010*(1-t[:, :, :, 0]) + t[:, :, :, 0]*n110
n01 = n001*(1-t[:, :, :, 0]) + t[:, :, :, 0]*n101
n11 = n011*(1-t[:, :, :, 0]) + t[:, :, :, 0]*n111
n0 = (1-t[:, :, :, 1])*n00 + t[:, :, :, 1]*n10
n1 = (1-t[:, :, :, 1])*n01 + t[:, :, :, 1]*n11
return (((1-t[:, :, :, 2])*n0 + t[:, :, :, 2]*n1) + 1) / 2

```

Pour rendre en 3D via matplotlib on va utiliser la méthode “scatter3D”. Néanmoins les paramètres (X,Y,Z) vont être des listes 1D pas des matrices. Donc il faut rearranger nos anciens paramètres à nos nouveaux. On le fait via une fonction simple :

```
[23]: """pic" = notre bruit;
"""val" et "infe" permettent d'avoir que des points avec des valeurs
→ supérieurs(si "infe" = False)/inférieurs(si "infe" = True) à "val".
def rearrange_list(pic, val, infe):
    #Nos 3 nouvelles listes
    X = []
    Y = []
    Z = []
    #On garde une liste supplémentaire pour la valeur du bruit au point de
→ coordonnée : (Xn, Yn, Zn)
    values = []
    #On itère donc sur nos 3 listes
    for k in range(len(pic)):
        for j in range(len(pic[k])):
            for i in range(len(pic[k][j])):
                value = pic[k][j][i]
                #On vérifie la condition énoncée dans le premier commentaire,
→ et on rajoute des nos points et valeurs.
                if((infe and (value < val)) or (not(infe) and (value > val))):
                    values.append(value)
                    X.append(i / len(pic[k][j]))
                    Y.append(j / len(pic[k]))
                    Z.append(k / len(pic))
    return (X, Y, Z, values)
```

On rajoute une autre fonction si jamais on souhaite baisser la résolution du bruit:

```
[24]: #"l" = une liste, "red" = nombre de points
def reduce_l(l, red):
```

```

res = []
for i in range(0,len(l),int(len(l)/red)):
    res.append(l[i])
return res

```

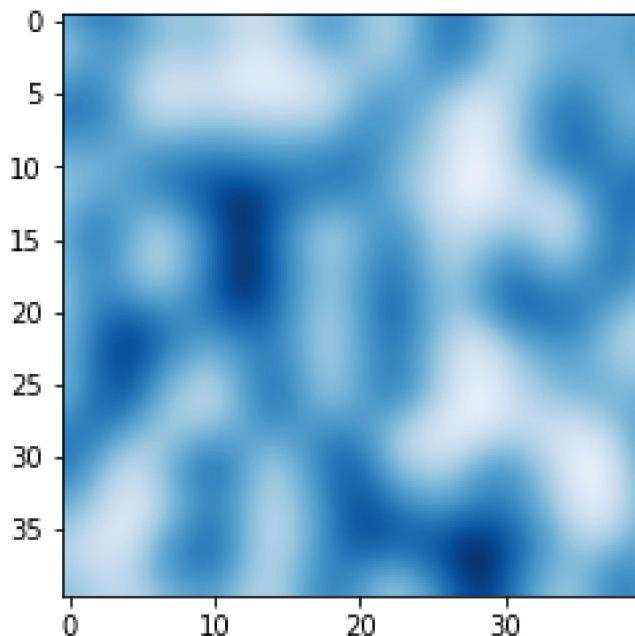
Essayons donc d'animer un plan de Perlin. C'est à dire qu'on va parcourir à chaque temps t un plan du bruit de Perlin 3D:

```
[25]: import matplotlib.pyplot as plt
from matplotlib import animation, rc
import matplotlib.cm as cm
rc('animation', html='jshtml')

pic = perlin3D([40,40,40], [5,5,5])

fig1 = plt.figure()
l = plt.imshow(pic[0], cmap="Blues", interpolation='quadric', animated=True)
animation.FuncAnimation(fig1, lambda p: l.set_data(pic[p]), frames=40, interval=100)
```

[25]: <matplotlib.animation.FuncAnimation at 0x7fe7a74534f0>



Rendons maintenant le bruit de Perlin 3D en entier avec une valeur minimale de 0.4 (sur 1 vu que le bruit de Perlin est normalisé):

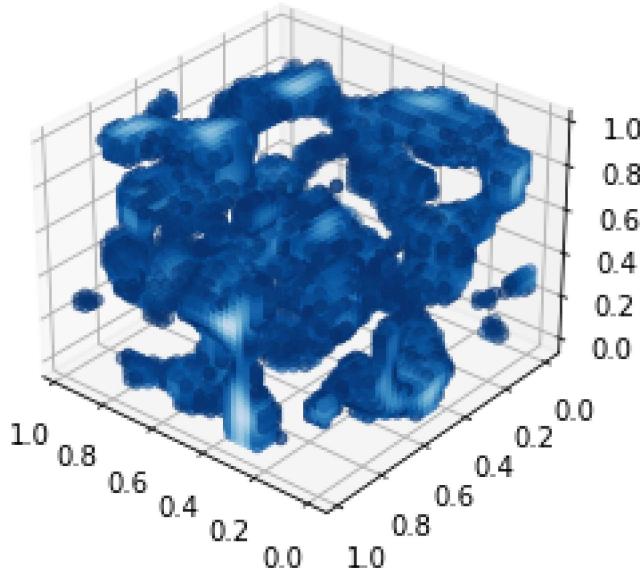
```
[26]: #On obtient nos paramètres avec la fonction décrite ci-dessus
X,Y,Z,values = rearrange_list(pic,0.4, True)

#Si on souhaite réduire nos points à 1000 par exemple.
reduce = 1000
#X = reduce_l(X, reduce)
#Y = reduce_l(Y, reduce)
#Z = reduce_l(Z, reduce)
#values = reduce_l(values, reduce)
values = np.array(values)

#Rendu avec matplotlib
fig = plt.figure(figsize=(4,4))
ax = fig.add_subplot(111, projection='3d')

#On rend et on anime
ax.scatter3D(X, Y, Z, c=values, cmap='Blues');
animation.FuncAnimation(fig, lambda phi: ax.view_init(30, phi*10), frames=50, interval=100)
```

[26]: <matplotlib.animation.FuncAnimation at 0x7fe7a6ec59a0>



### 3.1.2 Fractal perlin 3D

De la même manière que pour la 2D, on peut définir une version fractal:

```
[27]: def fractal3D(shape, res, octaves=1, persistance=0.5, exponentiation=1, lacunarity=1, norm=True, noise = []):
    #Si jamais on part d'un noise
    if(len(noise) == 0):
        noise = np.zeros(shape)
    frequency = 1
    amplitude = 1
    #On fait la somme de différents bruits de Perlin3D
    for _ in range(octaves):
        noise += amplitude * perlin3D(shape, (frequency*res[0], frequency*res[1], frequency*res[2]))
        frequency *= lacunarity
        amplitude *= persistance
    noise = noise**exponentiation
    #On normalise le résultat
    return noise/noise.max()
```

```
[28]: def reduqsdqsdce_noise(noise, val, infe):
    data = []
    values = []
    for k in range(len(pic)):
        data.append([])
        for j in range(len(pic[k])):
            data[k].append([0])
            for i in range (len(pic[k][j])):
                value = pic[k][j][i]
                if((infe and (value < val)) or (not(infe) and (value > val))):
                    data[k][j].append(value)
    return data
```

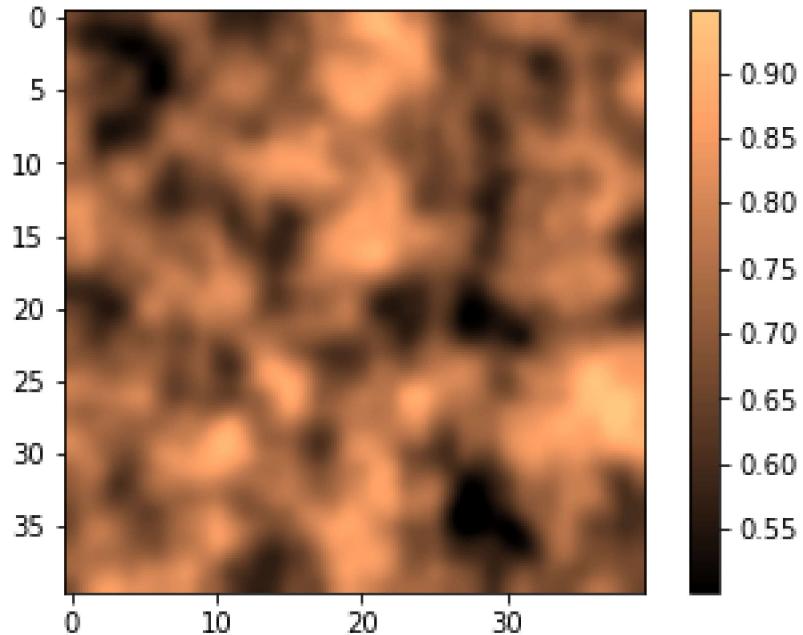
On calcule 2 résultats : une animation 2D (donc 2 axes position, 1 axe temps) ainsi qu'une animation 3D statique(3 axes position): (*Le code n'est plus commenté car il est exactement le même qu'au dessus, je change juste quelques paramètres par ci par là pour obtenir un meilleur rendu*).

```
[29]: import matplotlib.pyplot as plt
from matplotlib import animation, rc
import matplotlib.cm as cm
rc('animation', html='jshtml')

pic_fractal = fractal3D([40,40,40], [5,5,5], 3, 0.7, 1, 2, pic)

fig1 = plt.figure()
l = plt.imshow(pic_fractal[0], cmap="copper", interpolation='quadric', animated=True)
plt.colorbar()
animation.FuncAnimation(fig1, lambda p: l.set_data(pic_fractal[p]), frames=40, interval=100)
```

[29]: <matplotlib.animation.FuncAnimation at 0x7fe7a6e7b550>



```
[30]: import matplotlib.pyplot as plt
from matplotlib import animation, rc
import matplotlib.cm as cm
rc('animation', html='jshtml')

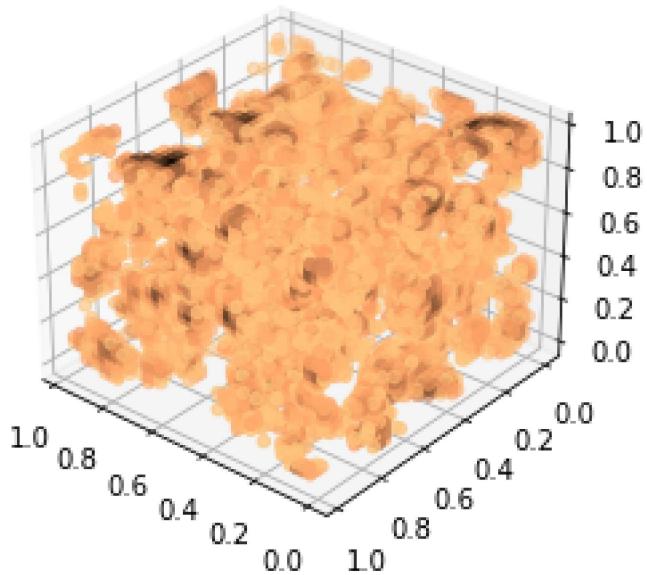
X,Y,Z,values = rearrange_list(pic_fractal,0.65, True)

reduce = 1000
#X = reduce_l(X, reduce)
#Y = reduce_l(Y, reduce)
#Z = reduce_l(Z, reduce)
#values = reduce_l(values, reduce)
values = np.array(values)

fig = plt.figure(figsize=(4,4))
ax = fig.add_subplot(111, projection='3d')

ax.scatter3D(X, Y, Z, c=values, cmap='copper');
animation.FuncAnimation(fig, lambda phi: ax.view_init(30, phi*10), frames=50, interval=100)
```

[30]: <matplotlib.animation.FuncAnimation at 0x7fe7a6c7b430>



### 3.1.3 Max

Ici aussi la méthode ne change pas:

```
[31]: def max_fractal3D(shape, res, octaves=1, persistance=0.5, exponentiation=1,  
→ lacunarity=1, noise = []):  
    if(len(noise) == 0):  
        noise = np.zeros(shape)  
    frequency = 1  
    amplitude = 1  
    for _ in range(octaves):  
        #On prend bien le maximum  
        noise = np.maximum(noise,amplitude * perlin3D(shape, (frequency*res[0],  
→frequency*res[1], frequency*res[2])))  
        frequency *= lacunarity  
        amplitude **= persistance  
    noise = noise**exponentiation  
    return noise / noise.max()
```

On rend les résultats:

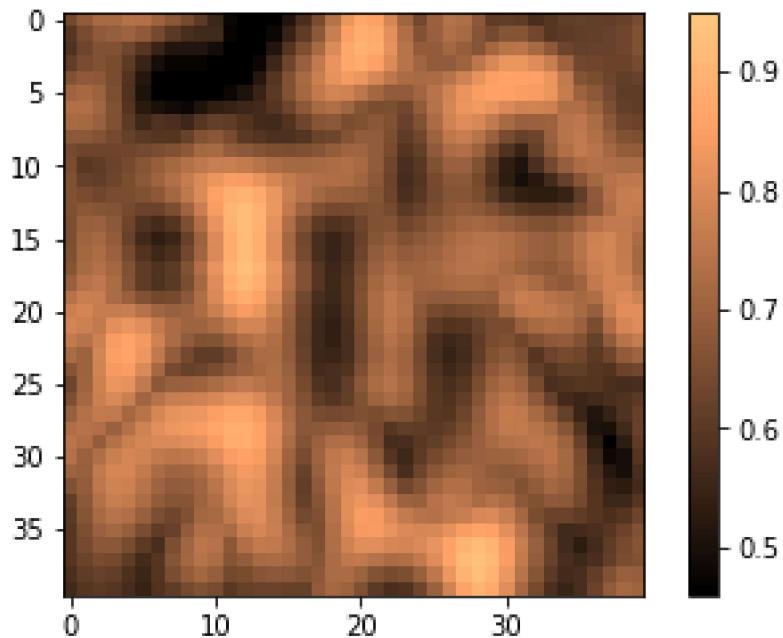
```
[32]: import matplotlib.pyplot as plt  
from matplotlib import animation, rc  
import matplotlib.cm as cm  
rc('animation', html='jshtml')  
  
pic_max = max_fractal3D([40,40,40], [5,5,5], 3, 0.7, 1, 2, pic)
```

```

fig1 = plt.figure()
l = plt.imshow(pic_max[0], cmap="copper", interpolation='None', animated=True)
plt.colorbar()
animation.FuncAnimation(fig1, lambda p: l.set_data(pic_max[p]), frames=40, interval=100)

```

[32]: <matplotlib.animation.FuncAnimation at 0x7fe7a6c05250>



```

[33]: import matplotlib.pyplot as plt
from matplotlib import animation, rc
import matplotlib.cm as cm
rc('animation', html='jshtml')

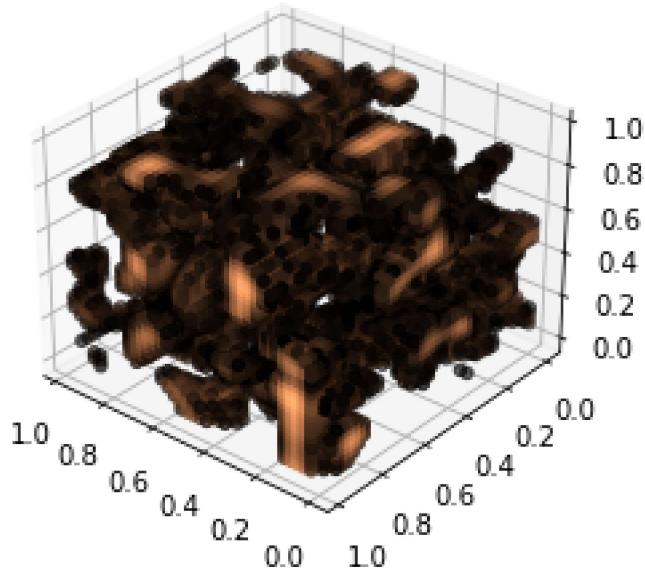
X,Y,Z,values = rearrange_list(pic_max,0.75, False)
values = np.array(values)

fig = plt.figure(figsize=(4,4))
ax = fig.add_subplot(111, projection='3d')

ax.scatter3D(X, Y, Z, c=values, cmap='copper');
animation.FuncAnimation(fig, lambda phi: ax.view_init(30, phi*10), frames=50, interval=100)

```

```
[33]: <matplotlib.animation.FuncAnimation at 0x7fe7a6a6eb80>
```



### 3.1.4 Strings

Pour finir voici le résultat si on l'applique avec le “string noise”:

```
[34]: def turbulent_fractal3D(shape, res, octaves=1, persistance=0.5, ↴exponentiation=1, lacunarity=1, noise = []):
    if(len(noise) == 0):
        noise = np.zeros(shape)
    frequency = 1
    amplitude = 1
    for _ in range(octaves):
        noise += np.abs(amplitude * perlin3D(shape, (frequency*res[0], ↴frequency*res[1], frequency*res[2]))- 0.5)
        frequency *= lacunarity
        amplitude *= persistance
    noise = noise**exponentiation
    return noise / noise.max()
```

```
[35]: import matplotlib.pyplot as plt
from matplotlib import animation, rc
import matplotlib.cm as cm
rc('animation', html='jshtml')

pic_strings = -1 * turbulent_fractal3D([100,100,100], [5,5,5], 1, 0.7, 1, 2) +1
```

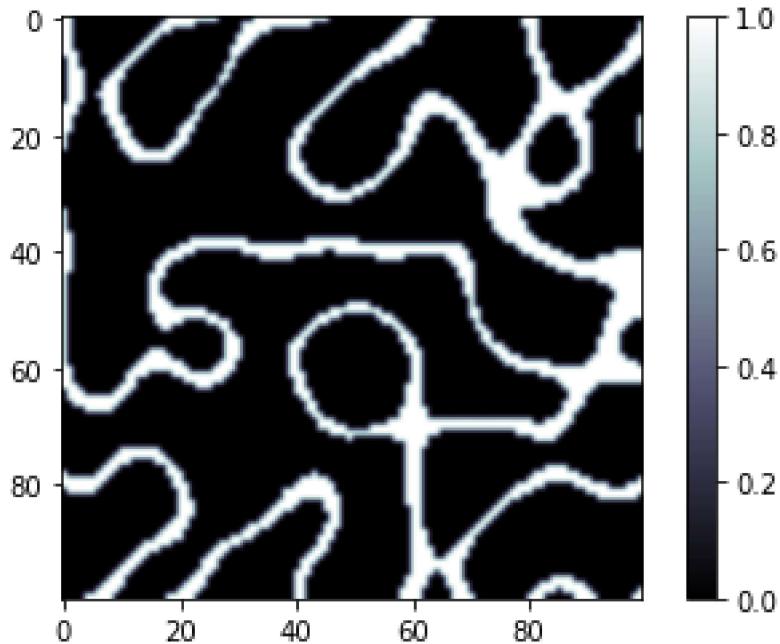
```

pic_strings = pic_strings > (pic_strings.max() * 0.9)

fig1 = plt.figure()
l = plt.imshow(pic_strings[0], cmap="bone", interpolation='quadric',□
→animated=True)
plt.colorbar()
animation.FuncAnimation(fig1, lambda p: l.set_data(pic_strings[p]), frames=40,□
→interval=100)

```

[35]: <matplotlib.animation.FuncAnimation at 0x7fe7a6c27e50>

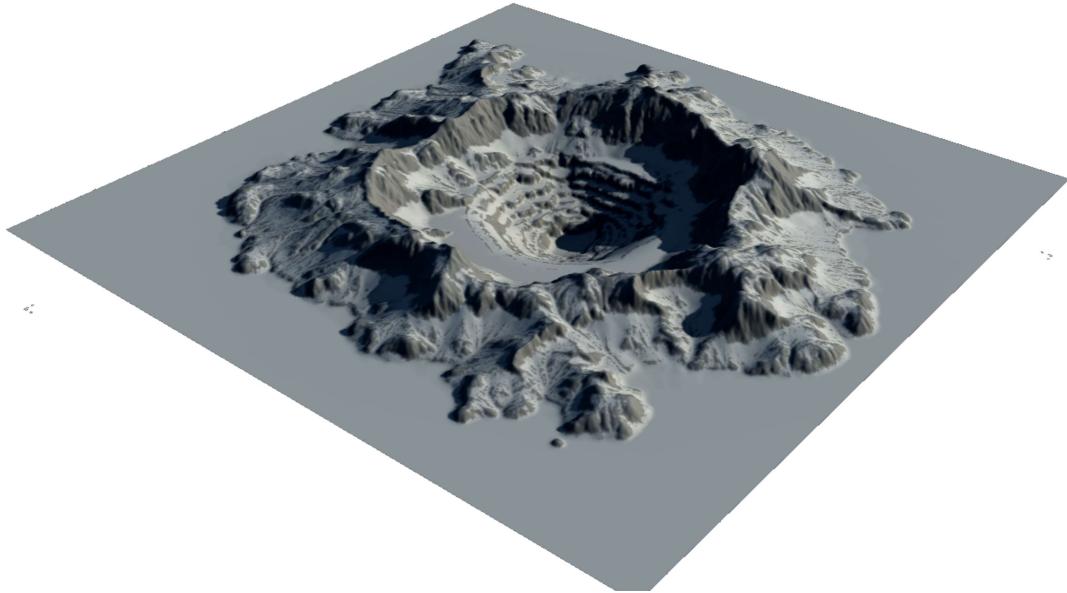


## 4 Application du bruit pour une déformation de surface

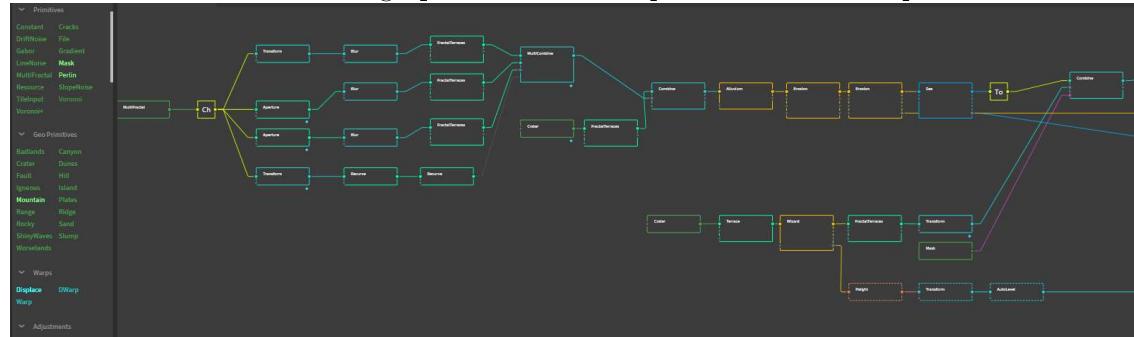
Jusqu'ici nous avons fait que des **textures**, c'est à dire nous **n'avons pas déformé** quelque chose d'existant  $\sim$  une géométrie 3D. Un exemple très important et où les bruits pseudo-aléatoires prennent sens est la **génération de terrain**. En effet que ça soit pour les décors de films, les jeux-vidéos ou encore des fonds pour des rendus la génération de terrain est essentiel. Et les logiciels générant du terrain se base essentiellement sur : 1. Les **bruits pseudo-aléatoires en première couche** et en ajout plus faible par dessus (opérations entre divers bruits). 2. Des **opérations mathématiques simples**: comme par exemple le plus utilisé “combine” = somme/multiplication/division... avec un “mask”(filtre spatial) pouvant être appliqué... 3. De la **simulation physique** : Principalement de l'érosion pour donner tout le détail au terrain. (Voir le chapitre dédié à la tentative de simulation de l'érosion sur un bruit de Perlin : échec lié à la résolution, à l'optimisation(ne pas pouvoir utiliser le GPU) et à beaucoup d'approximations faites mais le model physique est simple à comprendre).

Pour réaliser tous ça l'artiste n'a pas besoin de savoir coder. Les logiciels ont très souvent un système de **blueprints**(= cellules que l'on relie entre elles)(J'ai réalisé en 2021 une tentative de coder un système de blueprint en JavaScript, code disponible sur Github si besoin).

Un bon exemple, un des logiciels utilisés aujourd'hui: **Gaea**:  
Exemple de terrain créé sur Gaea(exemple réalisé fin 2021):



- Un morceau du graphe des blueprints utilisés pour le terrain:



On va voir dans cette partie que ce n'est pas difficile de déformer une forme avec un bruit.

#### 4.1 Pour un Plan

Afin de déformer un plan par une matrice 2D contenant notre bruit il faut **déplacer chaque "vertex" du plan sur l'axe z** en fonction de la valeur associée dans la matrice. C'est à dire que si notre plan est une fonction  $z(x,y) = \text{constante}$  : on la change en la fonction  $z(x,y) = \text{constante} + \text{bruit}(x,y)$ .

```
[36]: from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt

shape = 18*10
```

```

#On utilise le bruit fractal avec 3 octaves + avec une fréquence différente en ↴
fonction de l'axe.
pic_fractal = fractal3D([shape,shape,shape], [3,3,1], 3, 0.7, 1, 2)

#Sur MatPlotLib on doit réfléchir en fonctions et non en vertex/géométrie 3D.
def f(x, y, i):
    return np.zeros([shape,shape]) + pic_fractal[i]

#Notre grille
x = np.linspace(0, 1, shape)
y = np.linspace(0, 1, shape)

X, Y = np.meshgrid(x, y)
Z = f(X, Y, 0)

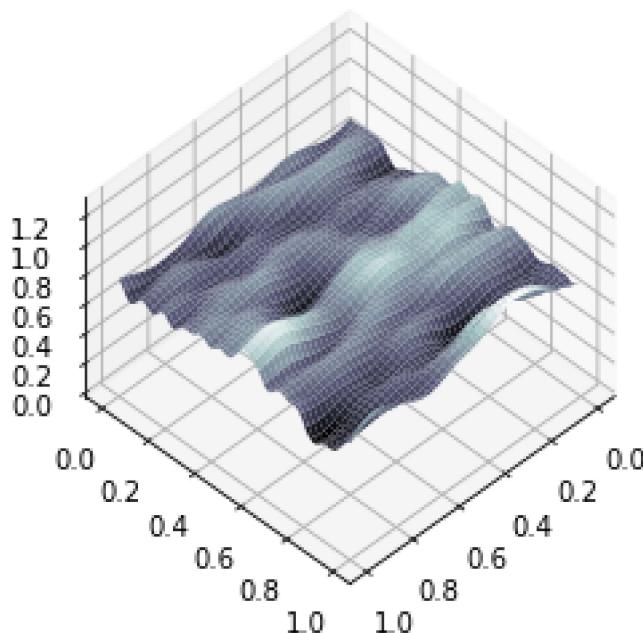
#Rendu sur matplotlib
fig = plt.figure(figsize = (4,4))
ax = plt.axes(projection ='3d')
ax.set_proj_type('ortho')
ax.view_init(45, 45)
ax.set_zlim3d(0,1.3)
ax.plot_surface(X, Y, Z, cmap=cm.terrain, linewidth=2, antialiased=True)

#On réalise une animation
def update(p):
    ax.clear()
    ax.set_zlim3d(0,1.3)
    ax.plot_surface(X, Y, f(X,Y,p), cmap="bone", linewidth=0, antialiased=True)

animation.FuncAnimation(fig, update, frames=int(shape/3), interval=50)

```

[36]: <matplotlib.animation.FuncAnimation at 0x7fe7a6d71f70>



## 4.2 Pour une Sphère

Déformer une sphère est **plus difficile à réaliser sur matplotlib** car il faut réfléchir en fonctions et non en géométrie 3D (donc non pas en coordonnées dans l'espace que l'on déplace avec des vecteurs). C'est pour cela que je générerai une sphère sur Python mais je donnerai le résultat final via ThreeJs sur Javascript en expliquant schématiquement ce que j'ai fait. À savoir qu'en général nous n'utilisons pas une sphère mais un cube déformé ("gonflé") pour éviter d'avoir + de points quand on se rapproche des pôles = hétérogénéité de la résolution sur la sphère.

```
[37]: from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt
import scipy.ndimage

shape = 9*10

#Coo de la sphère
u = np.linspace(0, 2 * np.pi, shape)
v = np.linspace(0, np.pi, shape)

#Maths pour faire la sphère
x = (np.outer(np.cos(u), np.sin(v)) +1 )/2
y = (np.outer(np.sin(u), np.sin(v)) +1 )/2
z = (np.outer(np.ones(np.size(u)), np.cos(v)) +1 )/2
```

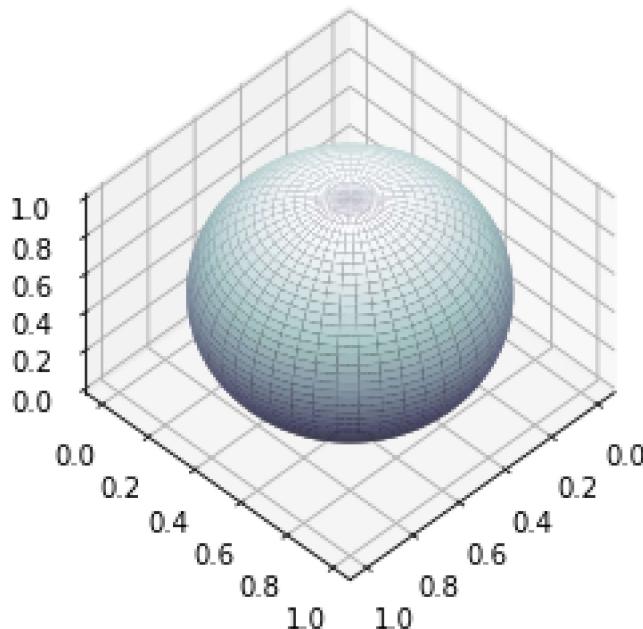
```

#Scipy pour adapter notre sphère à notre rendu
X = scipy.ndimage.zoom(x, 3)
Y = scipy.ndimage.zoom(y, 3)
Z = scipy.ndimage.zoom(z, 3)

#Rendu sur matplotlib
fig = plt.figure(figsize = (4,4))
ax = plt.axes(projection ='3d')
ax.set_proj_type('ortho')
ax.view_init(45, 45)
ax.plot_surface(X, Y, Z, cmap="bone", linewidth=2, antialiased=True)

```

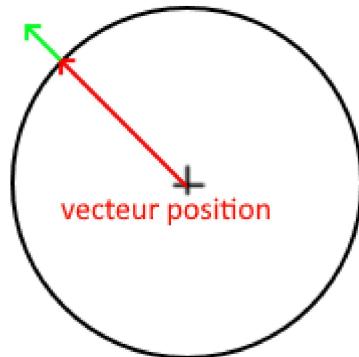
[37]: <mpl\_toolkits.mplot3d.art3d.Poly3DCollection at 0x7fe7a64f61f0>



Si chaque point du maillage est défini par un vecteur position avec le repère centré ou non sur le centre de la sphère (ci ce n'est pas le cas il suffit de rajouter un offset) alors on lui rajoute à ce vecteur sa copie normalisé(donc vecteur radial unitaire) puis multiplié par la valeur du bruit en ce point de l'espace. Le plus simple étant d'utiliser un bruit de Perlin

Nouveau vecteur  
position = somme des 2

vecteur radial unitaire \* bruit(x,y,z)



3D et sa valeur au point  $(x,y,z)$ . -  
uniquement opération avec un bruit de Perlin (Fractal : Ridge) on obtient le résultat suivant

En faisant cette



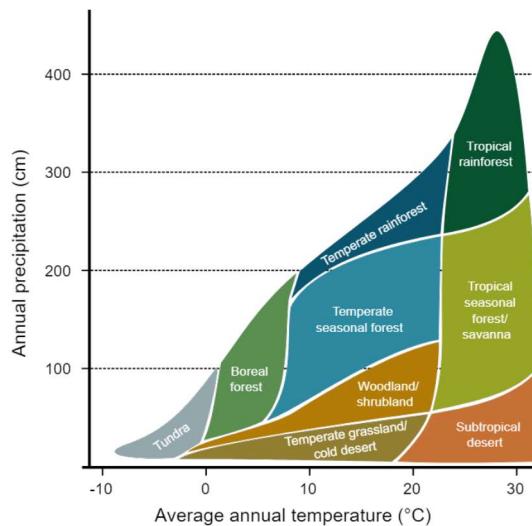
: - Ceux  
sont les deux géométries les plus utilisées (au petit détail près cité au début de la sous partie pour  
la sphère).

### 4.3 Application : Terrain complexe

Comme évoqué plusieurs fois avant une grande utilité du bruit de Perlin est la génération de Terrain, couplé avec une simulation réaliste de l'érosion on obtient des résultats avec un réalisme convaincant.

Ici nous allons réaliser comme exemple un océan avec des îles très montagneuses (en pic). Pour faire cela on va définir un bruit de base puis en fonction de ce bruit appliquer d'autres opérations. C'est à dire que l'intensité d'une opération : par exemple montagne est multiplié par la valeur du premier bruit (avec une puissance ou une translation...)

En général pour réaliser un terrain avec par exemple trois biomes : montagnes, plaines et canyons, on va générer un bruit de Perlin : "Température" et qu'en fonction de sa valeur à un point (x,y) les 3 biomes seront générés + ou - fortement. Dans un cas encore + réel on génère 2 bruits : "Précipitation" ET "Température" car par exemple un désert avec une forte humidité devient une forêt tropicale. C'est un moyen simple de donner de la diversité à la génération. On peut aussi changer la température manuellement en rajoutant un facteur variant en fonction de la latitude si on génère sur une planète : par exemple une gaussienne pour que l'équateur soit + chaud que les pôles.



```
[38]: import perlin2D as perlin
from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

shape = 2**11

temperature = perlin.Perlin2D([shape,shape],[2**2,2**2]).noise

mountains = -1 * np.abs(perlin.Perlin2D([shape,shape],[2**4,2**4]).fractal2D(3,
→0.5, 1, 2) - 0.8) + 1
plains = perlin.Perlin2D([shape,shape],[2**5,2**5]).noise

mountains = mountains**2
mountains *= 0.7 * temperature**3
mountains = np.clip(mountains,mountains.min()+mountains.max()/5,mountains.max())

plains *= (0.5-np.abs(0.5-temperature)) * 0.2
plains = plains **1.2

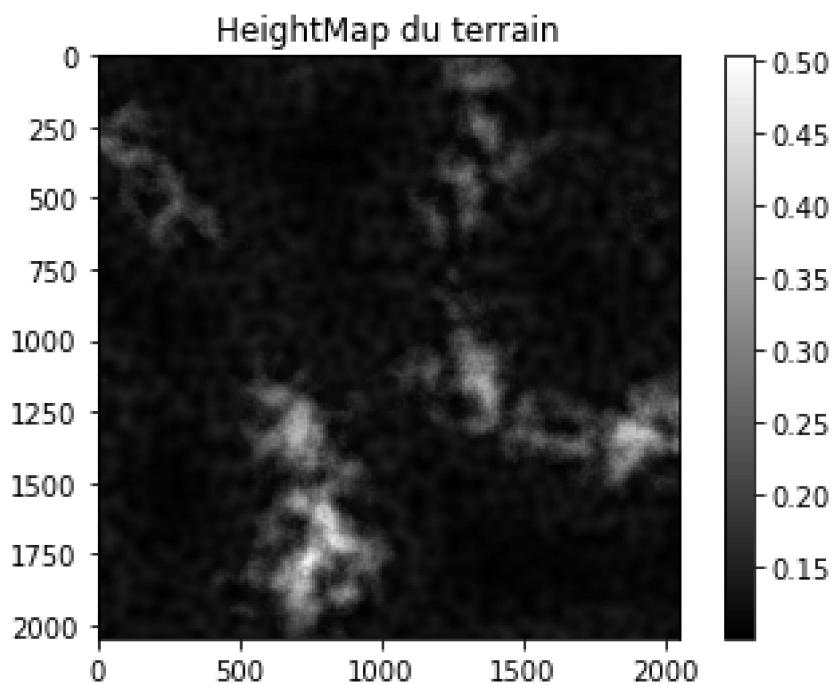
final_noise = mountains + plains
plt.imshow(final_noise, cmap=cm.gray)
plt.title("HeightMap du terrain")
plt.colorbar()

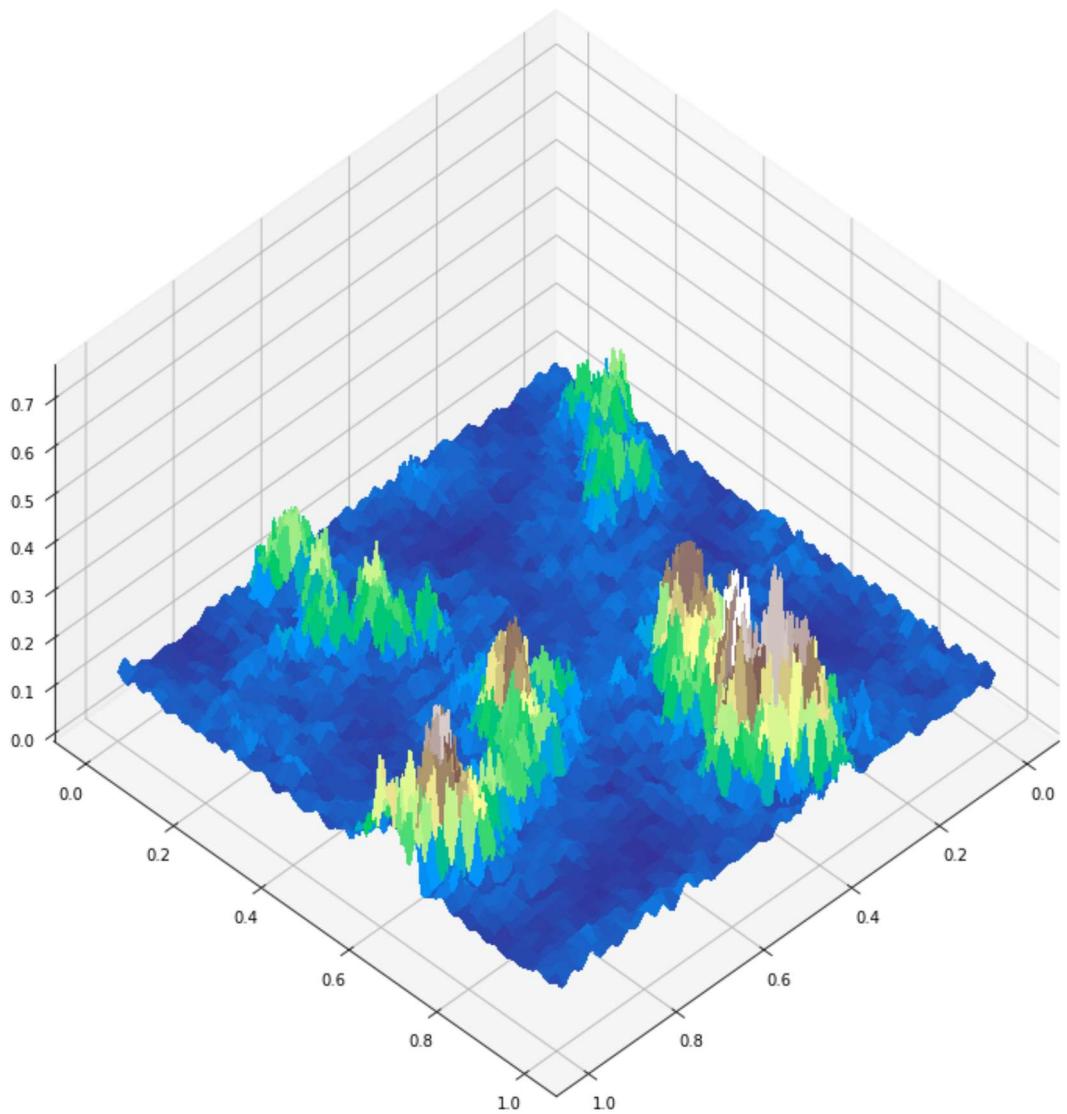
#Notre grille
x = np.linspace(0, 1, shape)
y = np.linspace(0, 1, shape)

X, Y = np.meshgrid(x, y)

#Rendu sur matplotlib
fig = plt.figure(figsize = (13,13))
ax = plt.axes(projection ='3d')
ax.set_proj_type('ortho')
ax.view_init(45, 45)
ax.set_zlim3d(0,final_noise.max()*1.5)
ax.plot_surface(X, Y, final_noise, cmap=cm.terrain, linewidth=0,
→antialiased=False)
```

[38]: <mpl\_toolkits.mplot3d.art3d.Poly3DCollection at 0x7fe7a6964970>





# Perlin - ChampVec - Rendu

December 2, 2022

## 1 Application du bruit de Perlin : Champ vec / Art numérique et Optimisation

**Objectif:** Générer un bruit plus complexe et original en partant d'un bruit de Perlin.

**Idée:** On utilise les valeurs du bruit de **Perlin** comme **angles**(donc des valeurs compris dans  $(0,2\pi)$ ) dans un champ vectoriel donc où un **vecteur de l'espace** sera :  $\vec{u}(x,y) = (\cos(\theta(x,y)), \sin(\theta(x,y)))$  où  $\theta$  est simplement notre bruit de Perlin rapporté sur  $0:2\pi$ . Ensuite on dessine la **trajectoire de particules** influencées par le champ.

**Plan:**

1. Prérequis
  - Rappel du bruit de **Perlin**
  - Bar de chargement
2. Premier visuel / **Idée** générale
3. **Génération** du bruit
4. **Manipulation** du bruit
  - 2D
  - 3D
5. **Optimisations** possibles pour gagner en temps de rendu
  - MultiProcessing (CPU)
  - QuadTree
  - Utilisation du GPU
6. Conclusion

### 1.1 Prérequis / Rappel : Génération du bruit de Perlin

On génère notre bruit de Perlin, pour le rappel et éviter de regarder entre plusieurs fichiers voici le code:

```
[1]: %%writefile perlin2D.py
import numpy as np

class Perlin2D():

    #Shape : [sx,sy] où sx et sy sont la taille de la matrice
    #Res : [rx,ry] où sx est multiple de rx, et rx = la résolution/fréquence
    #primaire sur x,
    #c'est à dire si rx augmente alors on calcule une plus grande zone
    #de notre bruit (même idée sur y)
    def __init__(self,shape,res):
        self.shape = shape
        self.res = res
        #Matrice aléatoire qui donne le côté aléatoire au bruit vu qu'elle sera
        #différente pour tous les instances
        #de l'objet perlin2D.
        self.random_numpy = np.random.rand(res[0]+1, res[1]+1)

        self.noise = self.perlin2D()

    def lerp(self, t):
        return 6*t**5 - 15*t**4 + 10*t**3

    def regenRdmNoise(self):
        self.random_numpy = np.random.rand(int(self.res[0])+1, int(self.
        res[1])+1)

    #Gen le bruit de Perlin en 2D
    def perlin2D(self):
        shape = self.shape
        res = self.res
        lerp = self.lerp

        #basé sur https://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf
        #; https://fr.wikipedia.org/wiki/Bruit_de_Perlin
        delta = (res[0] / shape[0], res[1] / shape[1])
        d = (shape[0] // res[0], shape[1] // res[1])

        #On définit la grille sur laquelle on va travailler, en effet res<shape
        #donc on créer les coordonnées dans les
        #dans les intervalles entre les points de la matrice aléatoire(de
        #taille res).
        grid = np.mgrid[0:res[0]:delta[0],0:res[1]:delta[1]].transpose(1, 2, 0)
        #%
```

```

#Le bruit de Perlin est basé sur l'utilisation des gradients et tout
→son intérêt provient de cette opération
    #appliquée sur la matrice aléatoire
    random_numpy = self.random_numpy
    angles = 2*np.pi*random_numpy[0:res[0]+1, 0:res[1]+1]
    gradients = np.dstack((np.cos(angles), np.sin(angles)))
    g00 = gradients[0:-1,0:-1].repeat(d[0], 0).repeat(d[1], 1)
    g10 = gradients[1:,0:-1].repeat(d[0], 0).repeat(d[1], 1)
    g01 = gradients[0:-1,1:].repeat(d[0], 0).repeat(d[1], 1)
    g11 = gradients[1:,1:].repeat(d[0], 0).repeat(d[1], 1)

    #Par la suite on l'applique à la grille en faisant des produits
→scalaires/rampes.
    n00 = np.sum(grid * g00, 2)
    n10 = np.sum(np.dstack((grid[:, :, 0]-1, grid[:, :, 1])) * g10, 2)
    n01 = np.sum(np.dstack((grid[:, :, 0], grid[:, :, 1]-1)) * g01, 2)
    n11 = np.sum(np.dstack((grid[:, :, 0]-1, grid[:, :, 1]-1)) * g11, 2)

    #On applique un procédé d'interpolation sur la grille (procédé défini
→dans le doc de Perlin)
    t = lerp(grid)
    n0 = n00*(1-t[:, :, 0]) + t[:, :, 0]*n10
    n1 = n01*(1-t[:, :, 0]) + t[:, :, 0]*n11

    #On normalise le bruit pour qu'il soit entre 0 et 1(pour faciliter la
→manipulation)
    return (np.sqrt(2)*((1-t[:, :, 1])*n0 + t[:, :, 1]*n1) + 1)/2

#Gen le bruit fractal
def fractal2D(self, octaves=1, persistance=0.5, exponentiation=1,
→lacunarity=1, norm=True):
    shape = self.shape
    res = self.res
    noise = np.zeros([int(shape[0]), int(shape[1])])

    frequency = 1
    amplitude = 1

    #Comparable à une série de Fourier, on ajoute plusieurs octaves/bruits
→avec une fréquence et amplitude
        #multipliées pour chaque itération de la série. = On rajoute du
→"détail" au bruit
    for _ in range(octaves):
        res = [frequency * res[0], frequency * res[1]]
        self.res = res
        self.regenRdmNoise()

```

```

        noise += amplitude * self.perlin2D()
        frequency *= lacunarity
        amplitude *= persistance

        noise = noise**exponentiation

#On normalise pour être certain qu'il soit entre 0 et 1
    return noise/noise.max() if(norm) else noise

```

### Writing perlin2D.py

En supplément, on développe une fonction pour montrer la progression du calcul dans la console afin d'avoir une estimation du temps que ça va prendre (vu que ça peut facilement être plusieurs heures).

```
[3]: def printProgressBar (iteration, total, prefix = '', suffix = '', decimals = 1,
→length = 100, fill = ' ', printEnd = "\r"):
    percent = ("{0:." + str(decimals) + "f}").format(100 * (iteration /
→float(total)))
    filledLength = int(length * iteration // total)
    bar = fill * filledLength + '-' * (length - filledLength)
    print(f'\r{prefix} |{bar}| {percent}% {suffix}', end = printEnd)
    if iteration == total:
        print()
```

## 1.2 Premier visuel

On développe deux objets : - Un pour une particule qui sera simulée dans notre espace, influencée par le champ vectoriel généré via le bruit de Perlin. - Un autre pour avoir les informations de la simulation dans sa globalité.

On peut déjà faire quelques essais (temps rendu long).

```
[29]: from matplotlib import animation, rc
rc('animation', html='jshtml')
import random
import matplotlib.pyplot as plt

#Grille de {shape} points de longueur avec {shape/delta} vecteurs sur une ligne.
→ Exemples générés donc avec 30*30 vecteurs.
shape = 300.0
delta = 10.0

#On génère le bruit
noise = Perlin2D([shape/10,shape/10],[5,5]).noise

#On génère le champ via le bruit de Perlin, une méthode est d'utiliser la
→valeur du bruit de Perlin comme un angle.
```

```

field = []
for i in range(len(noise)):
    for j in range(len(noise[i])):
        field.append((np.cos(noise[i][j] * (2*np.pi)),np.sin(noise[i][j] *_
→(2*np.pi))))
field = np.asarray(field)

#On transforme tout en des variables utilisables par matplotlib
X,Y = np.mgrid[delta:shape+delta:delta, delta:shape+delta:delta]
U=np.array([field[i][0] for i in range(len(field))]).reshape((int(shape/delta),_
→int(shape/delta)))
V=np.array([field[i][1] for i in range(len(field))]).reshape((int(shape/delta),_
→int(shape/delta)))

fig, ax = plt.subplots(figsize=(8,8))

#On crée notre objet particule
class Particle2D():
    #plot : [X,Y,U,V]
    def __init__(self,ax, mass, pos, plot, friction=0.5, color="red"):

        #A chaque itération du temps on va dessiner un cercle sur le plot
        self.c = plt.Circle((pos[0],pos[1]), 2, facecolor=color)
        self.iteration = 0
        self.color = color
        self.pos = pos
        self.mass = mass
        self.plot = plot
        #La particule perd un pourcentage de sa vitesse par itération du temps
        self.friction = friction
        #vitesse de la particule
        self.v = np.asarray([0,0])

        #pour dessiner le cercle
        ax.add_patch(self.c)

    #On update les propriétés de la particule à chaque dt
    def update(self,dt,t,time):
        #On redessine un cercle
        ax.add_patch(plt.Circle((self.pos[0],self.pos[1]), 3, facecolor=self.-
→color))
        #On update la position
        self.pos += self.v * dt

        #Si la particule est sorti de l'espace: on termine son calcul de_
→trajectoire

```

```

        if(self.pos[0] > shape or self.pos[0] < 0 or self.pos[1] > shape or
→self.pos[1] < 0):
            return True

        self.c.center = [self.pos[0],self.pos[1]]

#La particule perd de la vitesse car frottement
self.v = np.asarray((self.v[0] *self.friction,self.v[1] *self.friction))
#On récupère le vecteur de Perlin le plus proche
vec = self.getClosestVector()
#Le vecteur est une force nous donnant une accélération
self.v = np.asarray([(1/self.mass) * vec[0],(1/self.mass) * vec[1]])
return False

#Méthode pour avoir le plus proche vecteur d'une particule
def getClosestVector(self):
    X,Y,U,V = self.plot
    mini = [X.max(),Y.max()]
    vec = (0,0)
    #double boucle for classique(approche naïve/couteuse) pour itérer sur
→un espace
    for i in range(len(X)):
        for j in range(len(Y)):
            x = X[i][0]
            y = Y[0][j]
            if(np.abs(x-self.pos[0]) <= mini[0] and np.abs(y-self.pos[1])≤
→≤ mini[1]):
                mini[0] = np.abs(x-self.pos[0])
                mini[1] = np.abs(y-self.pos[1])
                vec = (U[j][i],V[j][i])
    return vec
#Si jamais on veut vérifier que le vecteur est bien le bon
def getAbsoluteVecPos(self,vec,U,V):
    for i in range(len(U)):
        for j in range(len(V)):
            if(U[j][i] == vec[0] and V[j][i] == vec[1]):
                return (i*delta,j*delta)

#On crée l'objet global qui contiendra les particules
class SimulationPlot():
    def __init__(self,number,ax,shape,X,Y,U,V):
        self.particles = []
        for i in range(number):
            #On ajoute autant de particules demandées avec position aléatoire
            self.particles.append(Particle2D(ax,1,np.array([random.
→random()*shape,random.random()*shape]),(X,Y,U,V),color=(0,0,0,0.5)))

```

```

#On update la simulation tt les dt
def update(self,dt,t,time):
    #Bar de progression pour suivre le rendu
    printProgressBar(t, time/dt, prefix = 'Progress:', suffix = 'Complete', □
→length = 50)
    temp_part = []
    #Si la particule n'est pas sorti on continue de l'update sinon on la □
→supprime de la liste des particules à itérer.
    for p in self.particles:
        if(not(p.update(dt,t,time))):
            temp_part.append(p)
    self.particles = temp_part

#Pour lancer la simulation
def simulate(self, ax, time, dt):
    for t in range(int(time/dt)):
        self.update(dt,t,time)

#On crée notre objet avec les paramètres
s = SimulationPlot(200,ax,shape,X,Y,U,V)

ax.axis([0, shape, 0, shape])

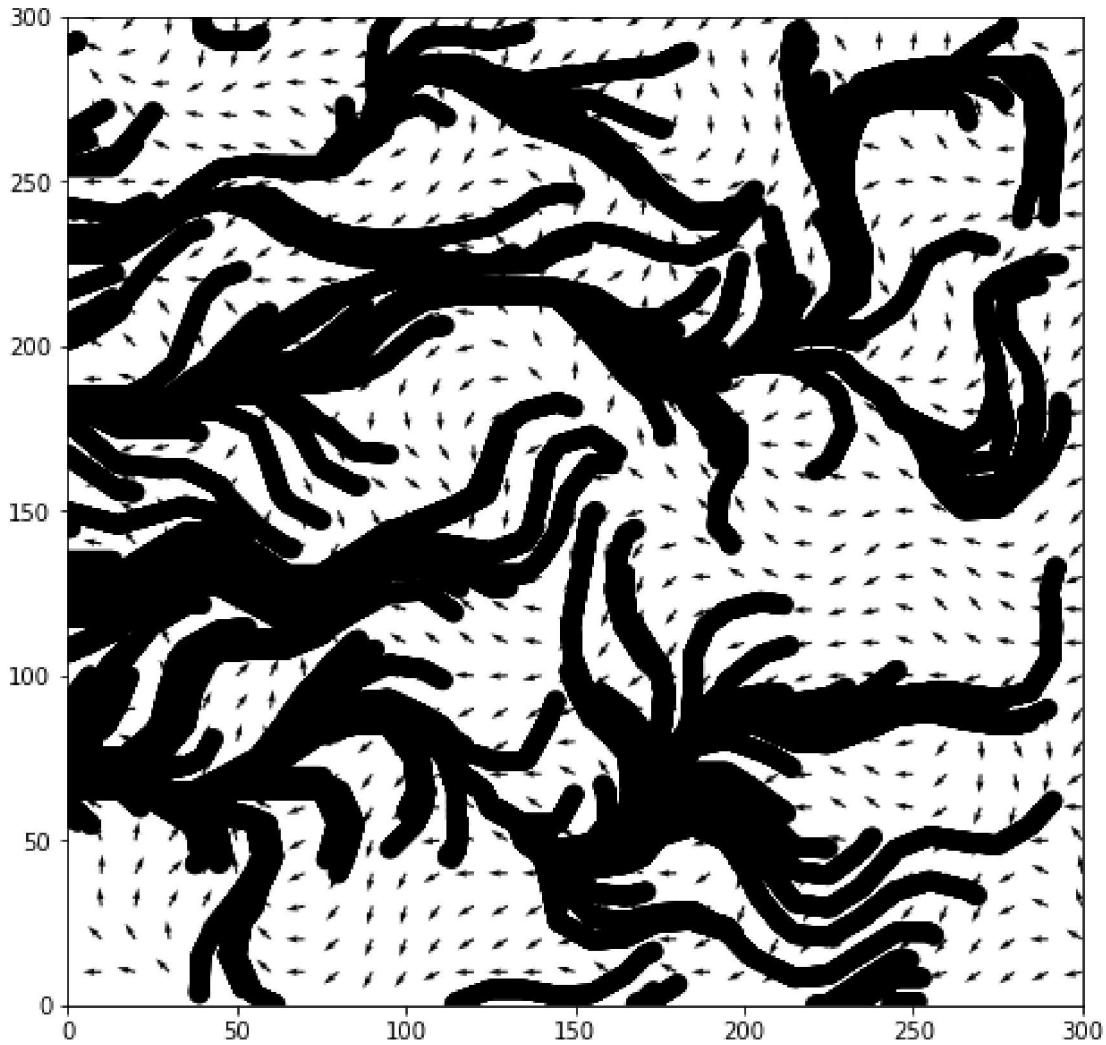
#On dessine le champ vectoriel pour avoir un aperçu
q = ax.quiver(X, Y, U, V)

#On lance la simulation
s.simulate(ax,500,0.1)

#On rend le résultat
plt.show()
#animation.FuncAnimation(fig, lambda p: s.update(.5,p,200), frames=200, □
→interval=50)

```

Progress: | -| 100.0% Complete



Le résultat a l'air d'être quelques choses de très visuel / de très "fluide", parfait... Maintenant on veut générer ça sur tout l'espace et pas d'une seule couleur mais sur une color map (discret à "continu").

### 1.3 Génération de notre nouveau bruit

Pour cela on va séparer l'espace en grille défini via une variable résolution puis dans chaque cellule de la grille on va simuler la trajectoire d'une particule partant du centre de celle-ci. Or à la place de dessiner directement la trajectoire on va augmenter une variable (de 0 à 1) compris dans une matrice correspondant à l'espace. C'est à dire que chaque position dans l'espace aura une valeur associée compris entre 0 et 1 à la fin. On dessinera ensuite cette matrice via matplotlib. (Cette matrice possède donc les mêmes propriétés d'un noise et est donc un "nouveau" bruit qu'on peut utiliser pour d'autres applications...).

[4]: #Même chose que précédemment mais on crée une particule par cellule de la grille défini par notre {shape} et {delta2}.

#Et cette fois-ci à la place de dessiner un cercle on augmente la valeur d'un élément d'une matrice qui correspond à notre espace

```

import numpy as np
import matplotlib.pyplot as plt

shape = 300.0
delta = 10.0

noise = Perlin2D([shape/10,shape/10],[5,5]).noise
field = []
for i in range(len(noise)):
    for j in range(len(noise[i])):
        field.append((np.cos(noise[i][j] * (2*np.pi)),np.sin(noise[i][j] * (2*np.pi))))
field = np.asarray(field)

X,Y = np.mgrid[delta:shape+delta:delta, delta:shape+delta:delta]
U=np.array([field[i][0] for i in range(len(field))]).reshape((int(shape/delta),int(shape/delta)))
V=np.array([field[i][1] for i in range(len(field))]).reshape((int(shape/delta),int(shape/delta)))

fig, ax = plt.subplots(figsize=(4,4))

ax.axis([0, shape, 0, shape])

q = ax.quiver(X, Y, U, V)
plt.show()

class Particle2D():
    #plot : [X,Y,U,V]
    def __init__(self, res, mass, pos, plot, friction=0.5):

        self.iteration = 0
        self.pos = pos
        self.mass = mass
        self.plot = plot
        self.friction = friction
        self.v = np.asarray([0,0])

        #On crée la matrice de l'espace pour cette particule (Inutile ici: pour du single thread)
        self.M = np.zeros((int(shape/res),int(shape/res)))

    def simulate(self, time, dt, incr):

```

```

        for t in range(int(time/dt)):
            flag, self.M = self.update(self.M,incr,dt,t,time)

def update(self,M,incr,dt,t,time):
    #On incrémente la variable correspond à la position de la particule
    →dans la matrice globale
    M[int(self.pos[1] * len(M) / shape)][int(self.pos[0] * len(M[0]) /
    →shape)] += incr

    self.pos += self.v * dt

    if(self.pos[0] > shape or self.pos[0] < 0 or self.pos[1] > shape or
    →self.pos[1] < 0):
        return (True, M)

    self.v = np.asarray((self.v[0] *self.friction,self.v[1] *self.friction))
    vec = self.getClosestVector()
    self.v = np.asarray([(1/self.mass) * vec[0],(1/self.mass) * vec[1]])
    #On retourne en + la matrice
    return (False, M)

def getClosestVector(self):
    X,Y,U,V,X2,Y2 = self.plot
    mini = [X.max(),Y.max()]
    vec = (0,0)
    for i in range(len(X)):
        for j in range(len(Y)):
            x = X[i][0]
            y = Y[0][j]
            if(np.abs(x-self.pos[0]) <= mini[0] and np.abs(y-self.pos[1]) <=
    →<= mini[1]):
                mini[0] = np.abs(x-self.pos[0])
                mini[1] = np.abs(y-self.pos[1])
                vec = (U[j][i],V[j][i])
    return vec
def getAbsoluteVecPos(self,vec,U,V):
    for i in range(len(U)):
        for j in range(len(V)):
            if(U[j][i] == vec[0] and V[j][i] == vec[1]):
                return (i*delta,j*delta)

class SimulationPlot():
    def __init__(self,res,shape,X,Y,U,V):

```

```

#Matrice de l'espace
self.M = np.zeros((int(shape/res),int(shape/res)))

self.particles = []
X2,Y2 = np.mgrid[res:shape+res:res, res:shape+res:res]

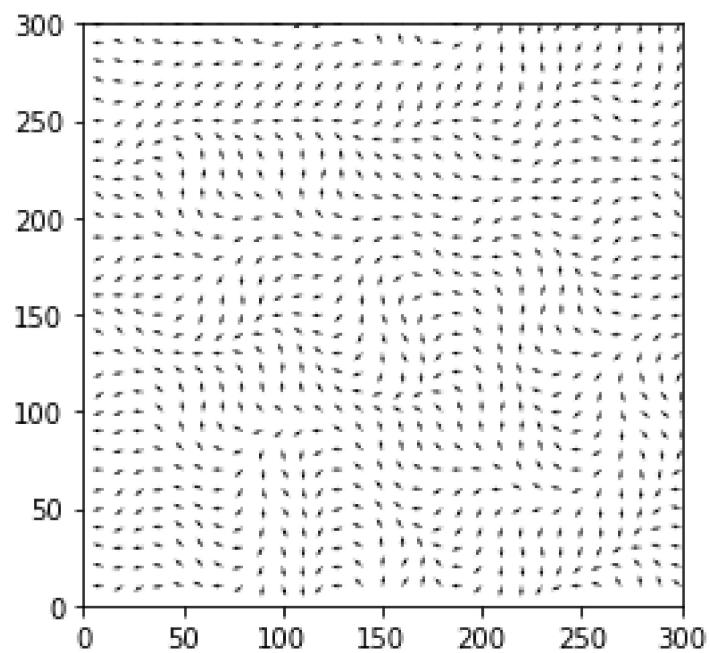
for i in range(int(shape/res)):
    for j in range(int(shape/res)):
        self.particles.append(Particle2D(res,1,np.
→array([i*res,j*res]),(X,Y,U,V,X2,Y2)))

def update(self,incr,dt,t,time):
    printProgressBar(t, time/dt, prefix = 'Progress:', suffix = 'Complete',_
→length = 50)
    temp_part = []
    for p in self.particles:
        #On récupère la matrice modifiée
        flag, self.M = p.update(self.M,incr,dt,t,time)
        if(not(flag)):
            temp_part.append(p)
    self.particles = temp_part
def simulate(self, border, time, dt, incr):
    for t in range(int(time/dt)):
        self.update(incr,dt,t,time)

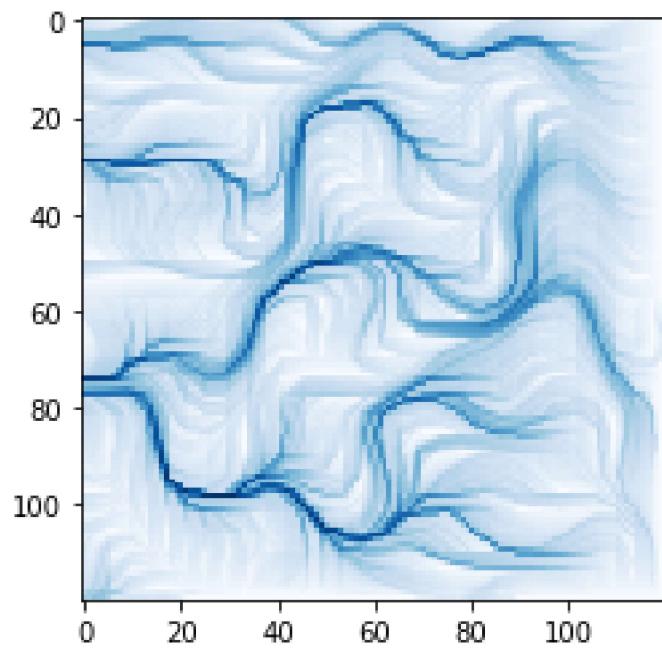
    #On return la matrice finale normalisée et avec un log ainsi qu'une_
→limite supérieur afin de faire ressortir
    #les valeurs basses (car sinon les hautes valeurs "cachent" les plus_
→faibles)
    return np.array(list(map(lambda i: list(map(lambda j: j if j <= border_
→else border ,i)),np.log(self.M+1)/self.M.max())))

delta = 2.5
s = SimulationPlot(delta,shape,X,Y,U,V)
result = s.simulate(0.1,100,0.5, 0.025)
#On peut appliquer de l'interpolation via matplotlib, par exemple : quadric,_
→pour, quelques fois, augmenter la qualité.
plt.imshow(result, cmap="Blues", interpolation='None')
plt.show()

```



Progress: | -| 99.5% Complete



```
[8]: #Si on veut enregistrer le rendu dans un fichier
```

```
#import numpy as np

#with open("Perlin_Vec1.npy", 'wb') as f:
#    np.save(f, result)

#NE PAS REUTILISER / FICHIER QUI COMPREND UN RENDU DE 1H30 (10k particules / ↗
→sans multiprocessing(avec multiprocessing sur jupyter uga:15mins))
```

## 1.4 Manipulation du bruit

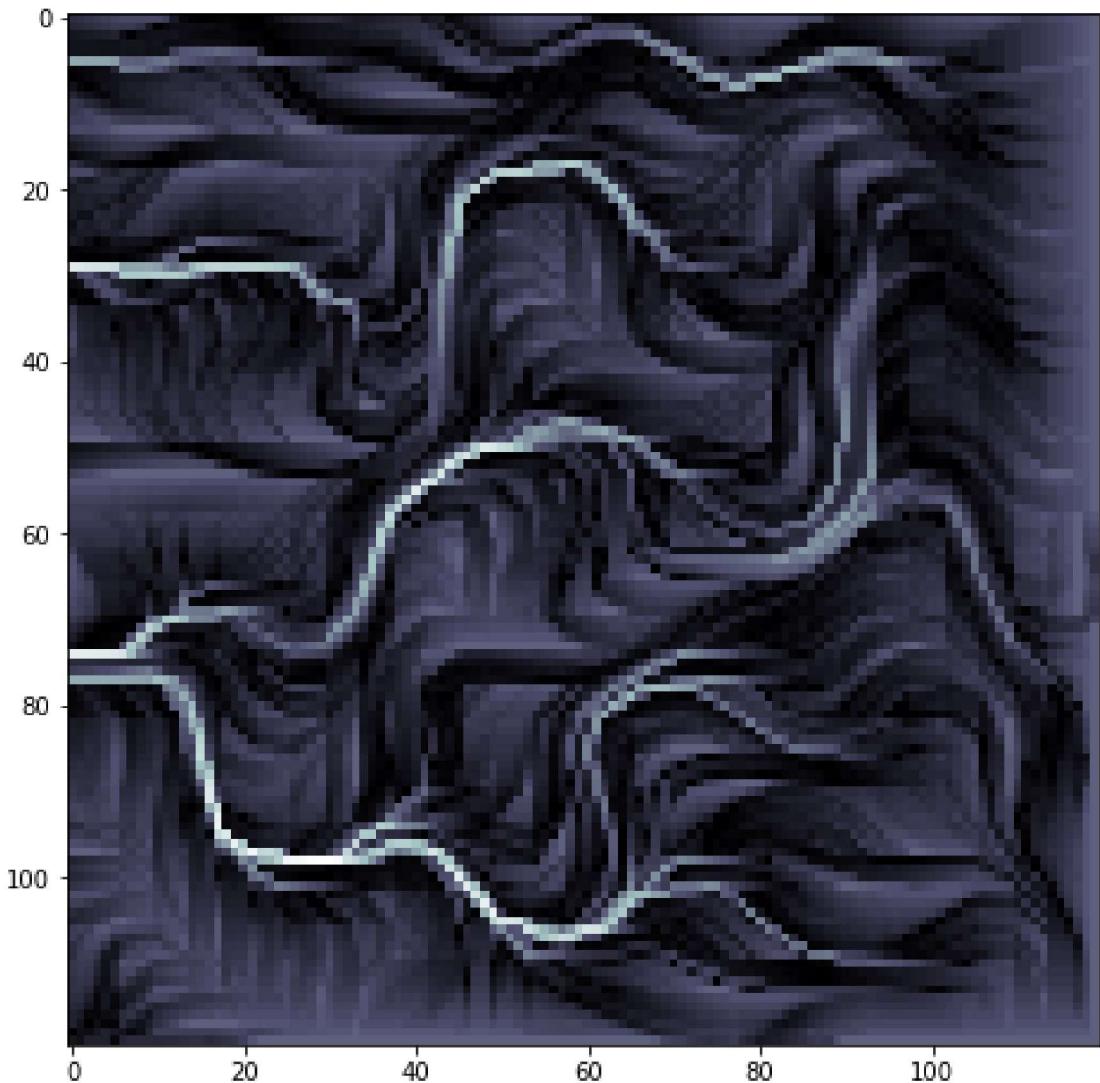
Maintenant qu'on a notre résultat, on peut lui appliquer quelques opérations pour le rendre plus visuel (dans notre cas) ou alors plus utilisable en lui appliquant une interpolation par exemple. Dans notre cas on va lui appliquer un “ridge” c'est à dire:  $-1 * |n(x, y)| + 1$  afin de marquer le relief. On lui a déjà appliqué à la fin du rendu un  $\log()$  afin de faire ressortir les “courants” du fond.

```
[66]: #On charge le rendu pré-généré lors d'une session précédente
with open('Perlin_Vec1.npy', 'rb') as f:
    result = np.load(f)

#Traitement de l'image

fig, ax = plt.subplots(figsize=(8,8))
#Même si, normalement, le bruit est déjà normalisé rien n'empêche de ↗
→renormaliser pour être certain de ce qu'on manipule.
result = result / result.max()
#On applique un ridge car on explore les possibilités visuels du bruit
result = -1 * np.abs(result-0.3) + 1
#On applique une palette de couleur
ax.imshow(result, cmap="bone_r", interpolation='None')
```

[66]: <matplotlib.image.AxesImage at 0x7f6a449130d0>



```
[128]: #Pourquoi on ne le rendrait-il pas en 3D ?
```

```
#Si jamais on ne lance que cette cellule de code, on le recharge
with open('Perlin_Vec1.npy', 'rb') as f:
    result = np.load(f)

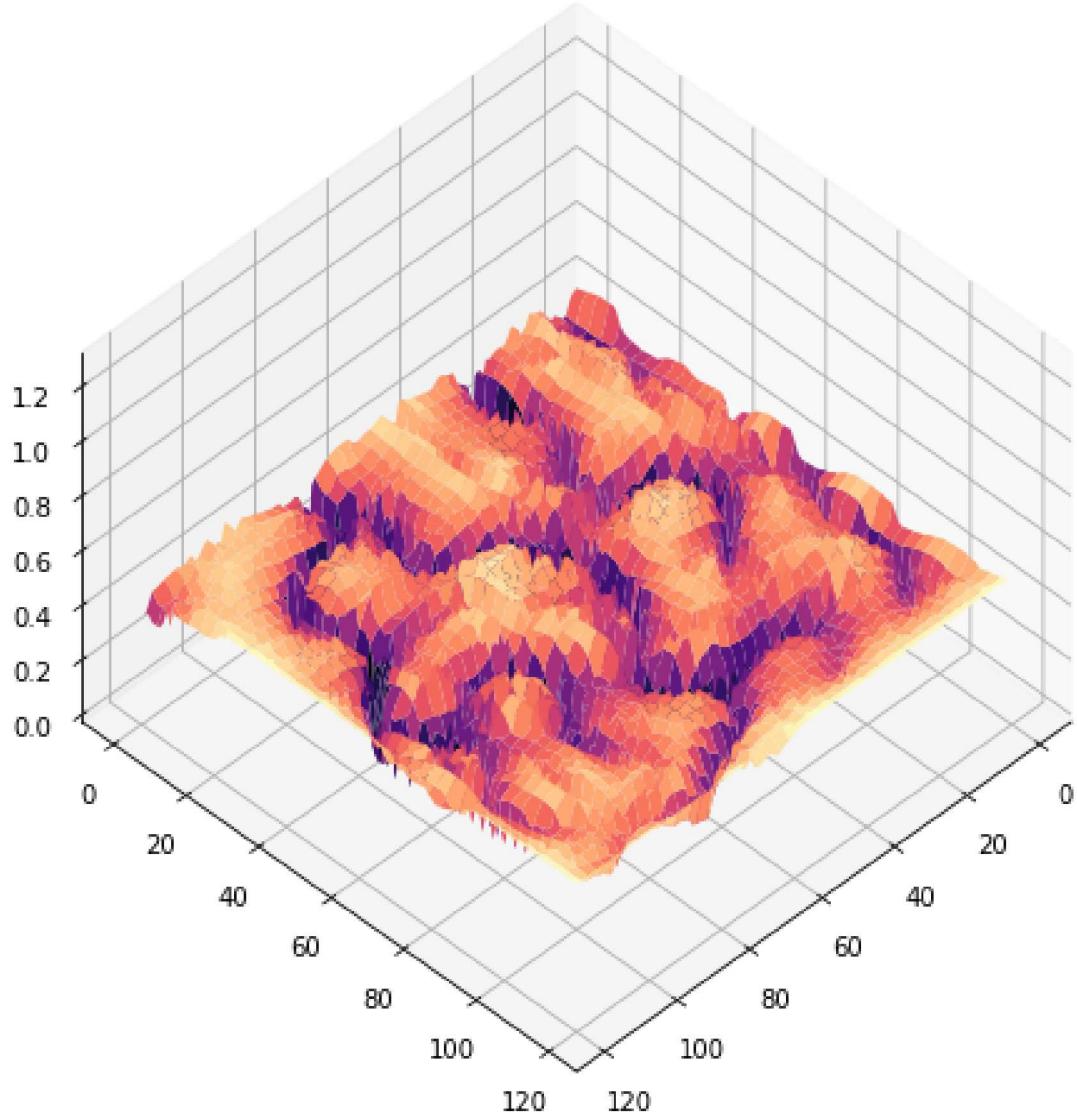
#Modifications empiriques
def f(x, y):
    return np.zeros([int(120), int(120)]) + (-result*10+0.5)

X, Y = np.mgrid[0:120, 0:120]

fig = plt.figure(figsize = (6,6))
```

```
ax = plt.axes(projection ='3d')
ax.set_proj_type('ortho')
ax.view_init(45, 45)
ax.set_zlim3d(0,1.3)
ax.plot_surface(X, Y, f(X, Y), cmap="magma", linewidth=0, antialiased=True)
```

[128]: <mpl\_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f6a3a8ab4f0>



## 1.5 Optimisation du temps de rendu/de calcul pour utilisations concrètes

### 1.5.1 - Introduction au MultiProcessing

Le rendu plus haut a pris **1h30** à rendre en faisant les instructions linéairements (comme une seule grande tâche). Or sur Python (et sur la majorité des autres langages de programmations) il est possible de “découper” cette grande tâche et calculer simultanément plusieurs petites tâches en même temps. Il ne faut pas confondre le **MultiThreading** et le **MultiProcessing**. En effet imaginons que nous voulons calculer deux tâches T1 et T2 avec des threads : le CPU va calculer un bout de la tâche T1 puis un bout de la tâche T2 et ainsi de suite. Ce n'est donc pas du simultané. C'est utile pour des interactions I/O mais pas pour du calcul pur et simple. C'est pour cela que nous passons par du MultiProcessing, à la place de créer des Threads sur un même core du CPU (donc une même unité de calcul) nous divisons la tâche principale en plusieurs petites tâches que nous allons attribuer à chaque core du CPU. (En général 4-8 cores pour un PC “normal”, 16-32 pour des VPS (virtual private server - hébergeur (comme sur jupyter) or ces cores ont une puissance de calcul divisé par le nombre de sessions/nombre d'utilisateurs sur le même VPS). Ainsi chaque core va faire une tâche (ici la trajectoire d'une seule particule à la place de toutes les particules) et dès qu'il a fini sa tâche on lui réattribue une autre particule et ainsi de suite jusqu'à avoir toutes calculées. Pour rassembler les résultats, il suffit d'additionner les matrices des trajectoires pour en obtenir une seule. On passe donc de **1h30 de rendu à 10 minutes** de rendu en utilisant **10 cores**, pour une même résolution.

(PS: On pourrait aussi utiliser le GPU qui est fait pour faire des tâches répétitives plus rapidement (des milliers de cores de + faible puissance). En gros GPU » CPU quand le nombre de données augmente)

Néanmoins on est limité par le nombre de particules donc notre résolution mais ce n'est pas un soucis de temps de calcul (la taille de la liste des particules est trop grande).

(Voir le fichier associé à celui-ci/ dédié au calcul)

**Ce qu'on a rajouté (ce qui est important) dans le code:**

```
[1]: #Pour savoir le nombre de cores disponibles sur le serveur
      import os
      print(os.cpu_count())
```

32

```
[ ]: import multiprocessing

#On crée notre manager
manager = multiprocessing.Manager()

#Le nombre de particules > au nombre de cores du CPU donc on divise en
#plusieurs étapes
for i in range(math.ceil(len(self.particles) / NUM_PROC)):
```

```

#Afin de partager la matrice entre les "threads" il faut qu'on
→utilise une variable instanciée qui ne change pas en fonction d'où on y
→accède, faut juste faire attention à ne pas utiliser la même variable pour 2
→process qui s'exécutent en même temps si ils sont synchrones. Donc DATA =
→dico de tt les matrices de pos de chaque particule qui sont exécutées en
→même temps.

DATA = manager.dict()

#jobs = les threads en cours
jobs = []
for k in range(NUM_PROC):
    if(i * NUM_PROC + k < len(self.particles)):
        #On crée le process
        process = multiprocessing.Process(
            target=self.particles[i*NUM_PROC+k].simulate,
            args=(i * NUM_PROC + k,DATA,time,dt,incr)
        )
        jobs.append(process)
#On les lance
for j in jobs:
    j.start()

#join() = on attend la fin du process pour revenir au main thread
→et donc continuer après = on attend tous les process pour continuer.
#Une optimisation serait de ne pas attendre tous les process pour
→en relance mais dès qu'il y a une place on relance directement.
for j in jobs:
    j.join()

#Bar de progression
printProgressBar(i, math.ceil(len(self.particles) / NUM_PROC),_
→prefix = 'Progress:', suffix = 'Complete', length = 50)

#On additionne les matrices de positions obtenus avec celle globale.
for m in DATA.values():
    self.M = self.M + m

```

### 1.5.2 - QuadTree / Subdivision par une grille (ou plusieurs grilles (grille contenant une grille))

(Optimisation possible pour le calcul d'un bruit de Worley; Passer d'une simulation de gravitation de 1k particules à 100k-1M(en temps réel); fluide etc etc)

Exemple pour **simulateur de gravitation**: on subdivise notre plan en une grille, on calcule les forces entre les particules d'une seule cellule, on fait une moyenne empire/ou gradient et on applique cette interaction moyenne aux autres cellules. Donc on peut subdiviser autant qu'on veut avec des grilles de taille différente. (PS le nombre d'opérations nécessaires pour une approche naïve d'un

simulateur de ce type est de  $n^2$  où  $n$  est le nombre de particules (car double boucle for (pour toutes les particules on calcule la force de toutes les particules sur celle de la première boucle) ). Avec ça on peut par exemple simuler la formation d'un système planétaire(avec collisions entre proto-planètes...), formation/collision de galaxies, expansion de l'Univers après le big bang et observer la forme d'"éponge" de l'Univers !. Ce n'est pas le projet ici mais c'est un très bon projet à faire pour débuter et très visuel.

Pour ce cas précis le mouvement de nos particules ne dépendent pas les une des autres donc ce n'est pas nécessaire d'utiliser QuadTree.

### 1.5.3 - Via le GPU

Le GPU contient bien plus de cores qu'un CPU mais à plus faible fréquence = moins performant. En effet le GPU est fait pour calculer des petites tâches répétitives en très grande quantité (d'où son utilisation dans le rendu en général). Pour faire du MultiProcessing via le GPU il faudrait un module dont je n'ai pas accès sur jupyter.

## 1.6 Conclusion

Avec un peu d'imagination on a obtenu un résultat complètement différent, si on veut on peut aussi rajouter des interactions entre les particules par exemple, afin d'obtenir un résultat bien différent encore.

Des artistes utilisent cette infinité de possibilités afin de produire de l'art numérique et donc des résultats qui ne sont que purement visuels (c'est à dire pas à but d'être utilisé dans un shader(=effet visuel d'un jeu par exemple)).

Si on voudrait utiliser le résultat qu'on a obtenu pour du terrain par exemple il serait fort intéressant de lerp/interpoler le résultat pour l'adoucir ou d'appliquer d'autres opérations en sortie.

# Perlin - Fourier

December 2, 2022

## 1 Utilisations des transformés de Fourier pour les bruits

Quand on parle de bruit on parle souvent aussi de **fréquences**. En effet un bruit même si il est purement aléatoire peut être considérée comme une **suite de fonctions à divers fréquences** qui peut aller jusqu'à une infinité de fonctions pour un bruit totalement aléatoire (mais **fini sur un domaine borné et dénombrable**).

Ainsi on peut en tirer deux questions : Que se passerait-il si on **filtrer les fréquences d'un bruit de Perlin** ? Ou encore pour un bruit **purement aléatoire**  $\Leftrightarrow$  Est-il possible de **produire un nouveau bruit pseudo-aléatoire** avec la transformé de Fourier ?

Plan : 1. **Filtres fréquentiels** pour le bruit de **Perlin**

- Génération du bruit de **Perlin** - Domaine fréquentiel du bruit ( $TF[]$ ) - **Filtre** et retour au domaine spatial ( $TF^{-1}[]$ )

2. Génération d'un **nouveau bruit pseudo-aléatoire**

- Génération du bruit **Random** - Domaine fréquentiel du bruit - Filtre et retour au domaine spatial

### 1.1 Filtres fréquentiels

On souhaite avoir qu'un échantillon des fréquences d'un bruit de Perlin "complet" afin de vérifier ce que nous avons dit plus haut et voir si on peut avoir un bruit de Perlin à faible fréquence si on utilise un filtre passe-bas sur le domaine fréquentiel.

#### 1.1.1 Génération du bruit de Perlin

On commence donc par générer notre bruit de Perlin, via le module que nous avons développé précédemment, en lui mettant une très forte fréquence spatiale ainsi qu'une bonne résolution.

```
[488]: import numpy as np
import perlin2D as perlin
import matplotlib.pyplot as plt

#La TF est calculée plus vite si on est en puissance de 2(car méthode
#qu'utilise numpy)
shape = 2**8
noise = perlin.Perlin2D([shape,shape],[2**6,2**6]).noise

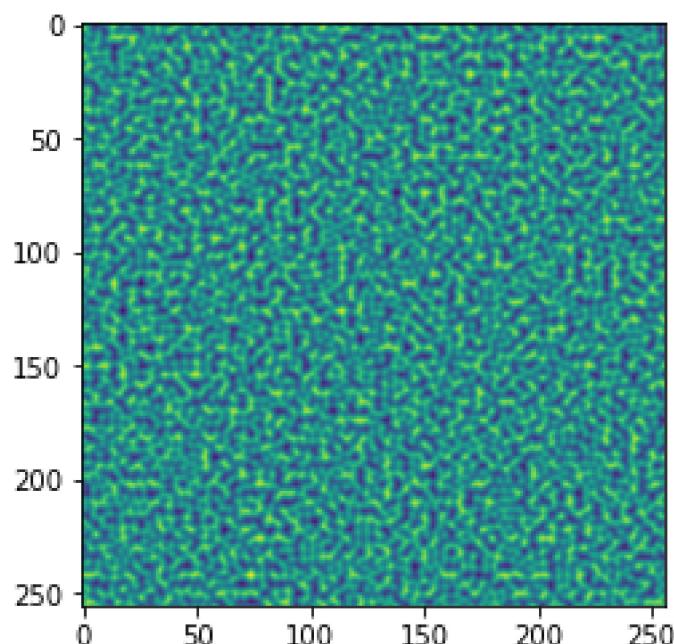
#On génère une grille/système de coo pour pyplot
X, Y = np.mgrid[0:shape, 0:shape]
```

```

#On rend le bruit en 2D
fig = plt.figure(figsize = (4,4))
ax = plt.axes()
ax.imshow(noise)
plt.show()

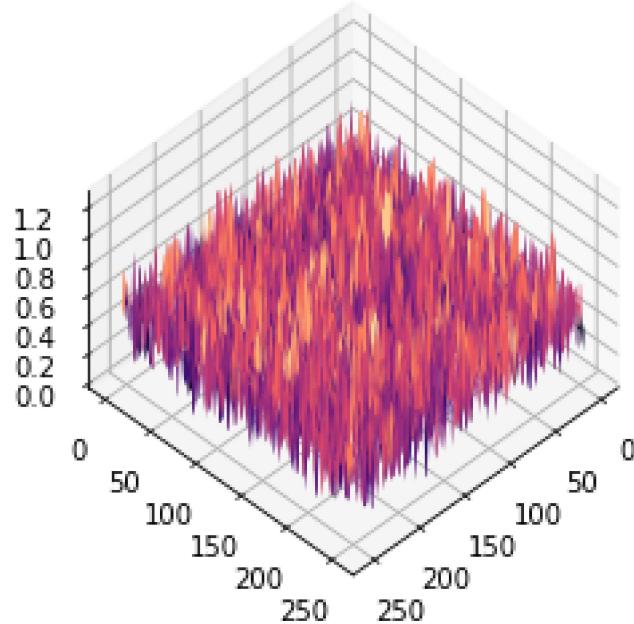
#On rend le bruit en 3D
fig = plt.figure(figsize = (4,4))
ax = plt.axes(projection ='3d')
ax.set_proj_type('ortho')
ax.view_init(45, 45)
ax.set_zlim3d(0,1.3)
plt.title('Bruit de Perlin')
ax.plot_surface(X, Y, noise, cmap="magma", linewidth=0, antialiased=True)

```



[488]: <mpl\_toolkits.mplot3d.art3d.Poly3DCollection at 0x7efc1fd95f70>

## Bruit de Perlin



### 1.1.2 Domaine fréquentiel du bruit ( [] )

Appliquons la transformé de Fourier pour ce bruit, nous utiliserons une unité arbitraire pour la fréquence qui est de 1 à {shape} (ici 512). Les basses fréquences sont celles proches de 0.

```
[469]: #On applique la tf 2D à l'aide de numpy
fft = np.fft.fft2(noise)

#On centre la TF, seulement pour le côté visuel d'où une variable à côté.
dfft = np.fft.fftshift(fft/2)

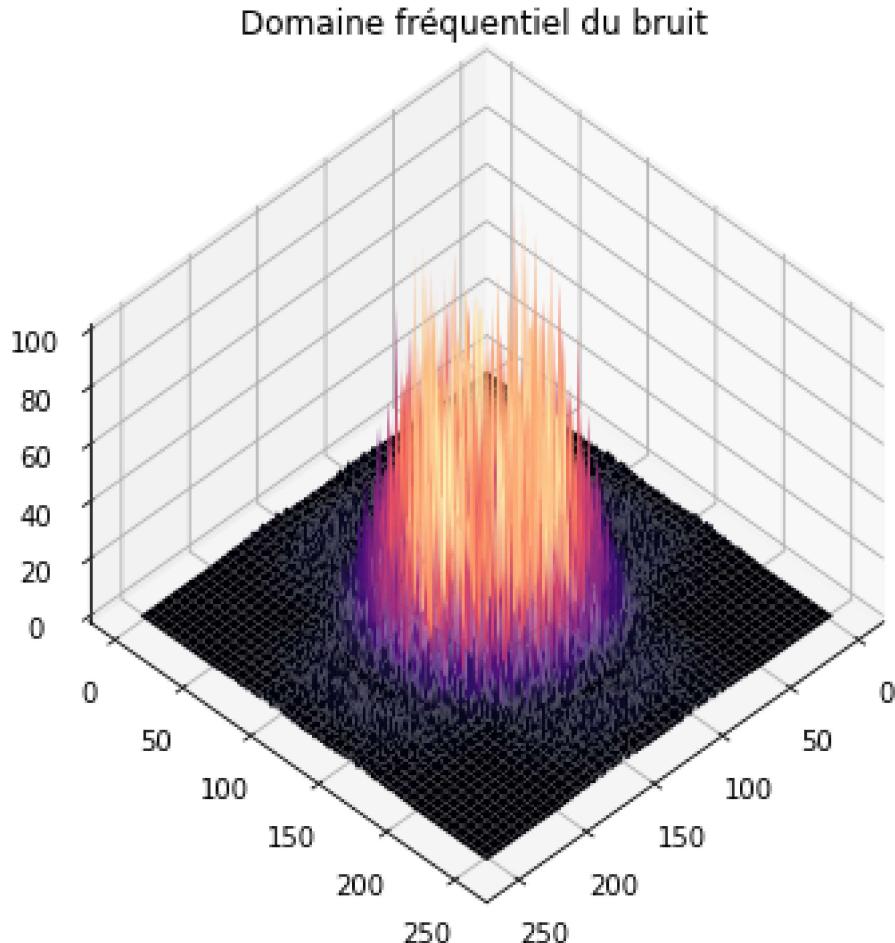
#paramètres du rendu
fig = plt.figure(figsize = (4,4))
ax = plt.axes(projection ='3d')
ax.set_proj_type('ortho')
ax.view_init(45, 45)
ax.set_zlim3d(0,100)
plt.title("Domaine fréquentiel du bruit")

#coordonnées
x, y = np.mgrid[:shape,:shape]

#On rend les valeurs absolues des valeurs de la matrice du bruit car ça permet
#d'avoir une belle répartition de couleur.
```

```
ax.plot_surface(x, y, np.abs(dfft[:len(x), :len(y)]), cmap="magma", linewidth=0,  
    antialiased=True)
```

[469]: <mpl\_toolkits.mplot3d.art3d.Poly3DCollection at 0x7efc26557670>



On observe bien que le nombre de fréquences est élevé vu la complexité apparente de la fonction 2D.

### 1.1.3 Filtre et retour au domaine spatial ( -1[] )

On programme une simple fonction de filtre qui si la fréquence est trop haute ou basse on réduit le bruit à 0. Puis on applique la transformée de Fourier inverse.

```
[487]: #Bornes du filtre  
limit_f = [0,5]  
fX = X  
fY = Y
```

```

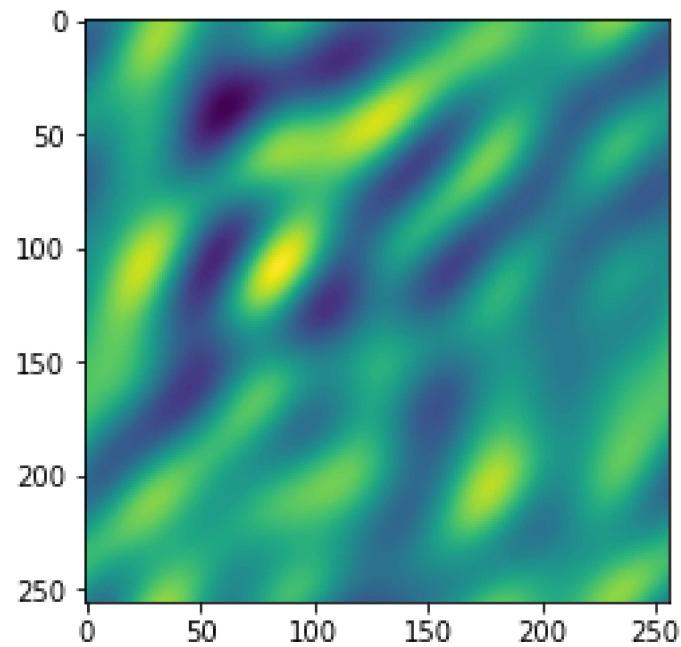
#On vérifie si on est bien dans les bornes, sinon on réduit le bruit à 0
l_fft = np.copy(fft[:len(fX),:len(fY)])
for i in range(len(fX)):
    for j in range(len(fY)):
        if(fX[i][0] > limit_f[1] or fY[0][j] > limit_f[1] or fX[i][0] < limit_f[0] or fY[0][j] < limit_f[0]):
            l_fft[i][j] = 0

#On retourne dans le domaine spatiale avec la TF inverse
ifft = np.fft.ifft2(l_fft)
#On prend les valeurs réels, celles qui ont un réel sens physique.
low_noise = np.real(ifft)

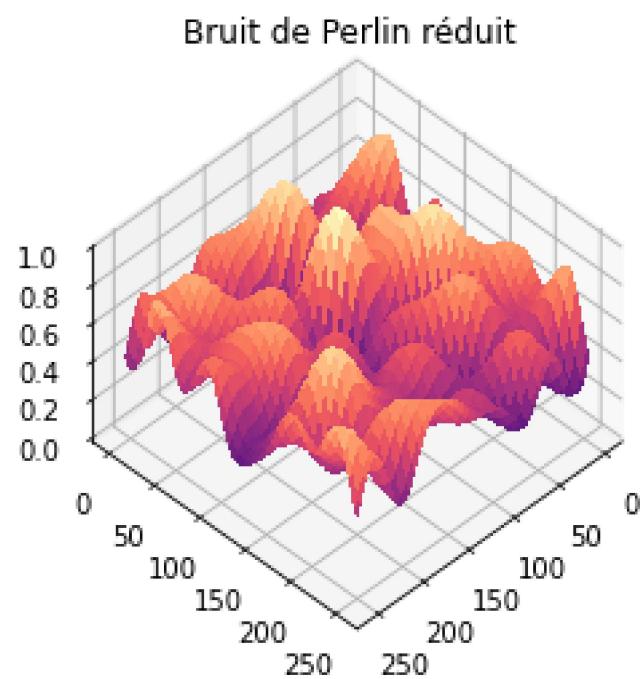
#Rendu 2D du bruit
fig = plt.figure(figsize = (4,4))
ax = plt.axes()
ax.imshow(low_noise)
plt.show()

#rendu et normalisation du bruit pour la 3D
X, Y = np.mgrid[0:len(low_noise), 0:len(low_noise)]
low_noise = low_noise - low_noise.min()
low_noise = low_noise / low_noise.max()
fig = plt.figure(figsize = (4,4))
ax = plt.axes(projection ='3d')
ax.set_proj_type('ortho')
ax.view_init(45, 45)
plt.title('Bruit de Perlin réduit')
ax.plot_surface(X, Y, low_noise, cmap="magma", linewidth=0, antialiased=False)

```

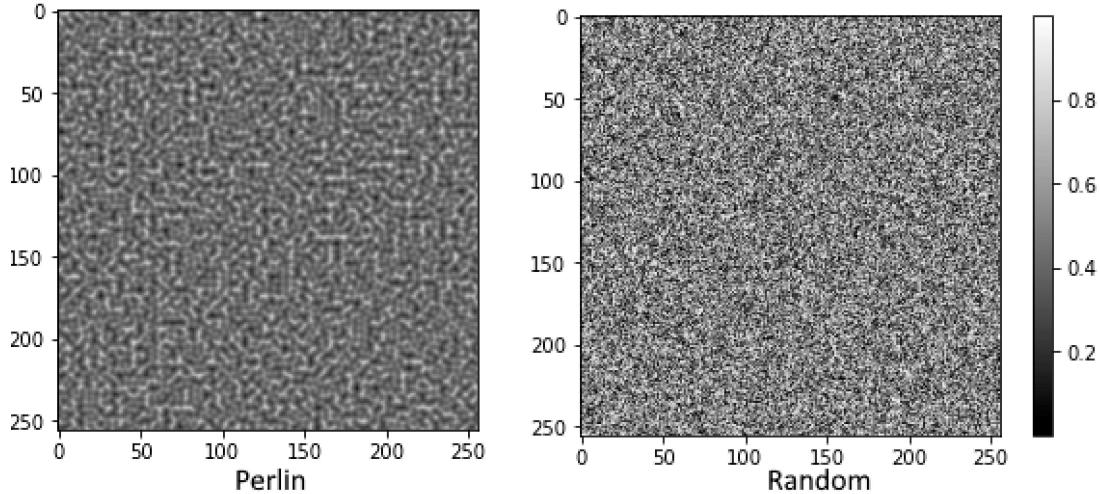


[487]: <mpl\_toolkits.mplot3d.art3d.Poly3DCollection at 0x7efc1ffc8430>



## 1.2 Génération d'un nouveau bruit pseudo-aléatoire

Si on observe un bruit purement aléatoire et un bruit de Perlin à haute fréquence spatiale initiale on voit une forte ressemblance et on ne voit pas la différence si la fréquence est assez haute avec une résolution faible, en effet quand la fréquence augmente les “bulles” du bruit de Perlin vont diminuer en taille et finir par former des points comme pour un bruit Random.



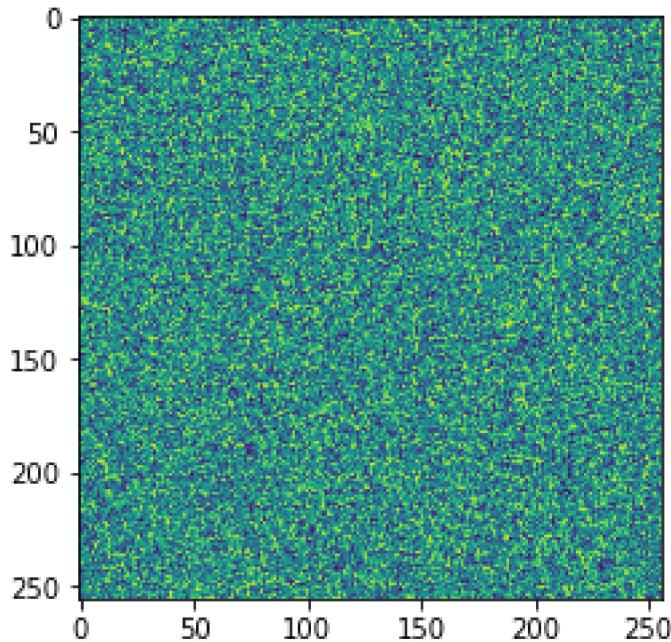
Il serait donc intéressant d'essayer d'appliquer un filtre passe-bas pour un bruit random et voir ce que ça donne.

### 1.2.1 Génération du bruit Random

Commençons par générer un bruit totalement(hypothétiquement) aléatoire.

```
[1]: import numpy as np
import matplotlib.pyplot as plt

shape = 2**8
#On génère le bruit aléatoire
noise = np.random.rand(shape, shape)
#Rendu
fig = plt.figure(figsize = (4,4))
ax = plt.axes()
ax.imshow(noise)
plt.show()
```



### 1.2.2 Domaine fréquentiel du bruit

Même étape que précédemment: On applique une première TF au bruit random.

```
[2]: #On applique la tf 2D à l'aide de numpy
fft = np.fft.fft2(noise)

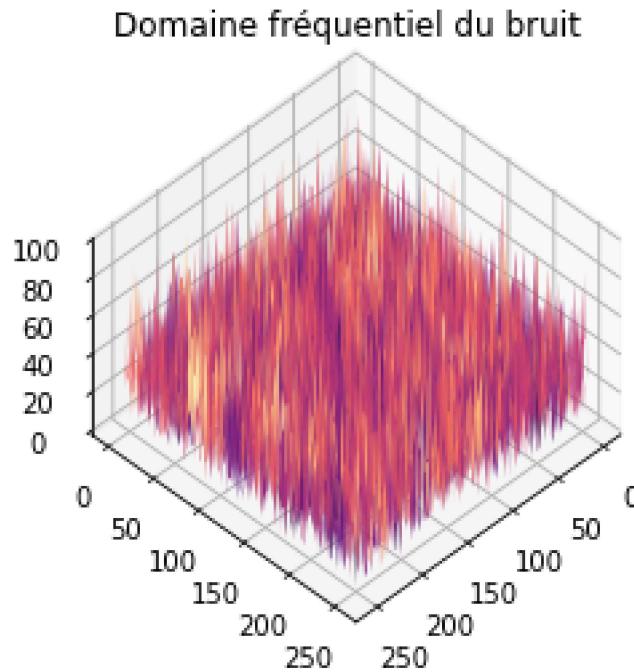
#On centre la TF, seulement pour le côté visuel d'où une variable à côté.
dfft = np.fft.fftshift(fft/2)

#paramètres du rendu
fig = plt.figure(figsize = (4,4))
ax = plt.axes(projection ='3d')
ax.set_proj_type('ortho')
ax.view_init(45, 45)
ax.set_zlim3d(0,100)
plt.title("Domaine fréquentiel du bruit")

#coordonnées
x, y = np.mgrid[:shape,:shape]

#On rend les valeurs absolues des valeurs de la matrice du bruit car ça permet
#d'avoir une belle répartition de couleur.
ax.plot_surface(x, y, np.abs(dfft[:len(x),:len(y)]), cmap="magma", linewidth=0,
antialiased=True)
```

[2]: <mpl\_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f959e554220>



On observe que ça a l'air tout autant aléatoire que le bruit d'origine. Mais que se passe t-il si on applique un filtre passe-bas ? Le bruit sera t-il cohérent dans l'espace ?

### 1.2.3 Filtre et retour au domaine spatial

```
[4]: #Bornes du filtre
limit_f = [0,5]
fX, fY = np.mgrid[0:shape, 0:shape]

#On vérifie si on est bien dans les bornes, sinon on réduit le bruit à 0
l_fft = np.copy(fft[:len(fX),:len(fY)])
for i in range(len(fX)):
    for j in range(len(fY)):
        if(fX[i][0] > limit_f[1] or fY[0][j] > limit_f[1] or fX[i][0] < limit_f[0] or fY[0][j] < limit_f[0]):
            l_fft[i][j] = 0

#On retourne dans le domaine spatiale avec la TF inverse
ifft = np.fft.ifft2(l_fft)
#On prend les valeurs réels, celles qui ont un réel sens physique.
low_noise = np.real(ifft)

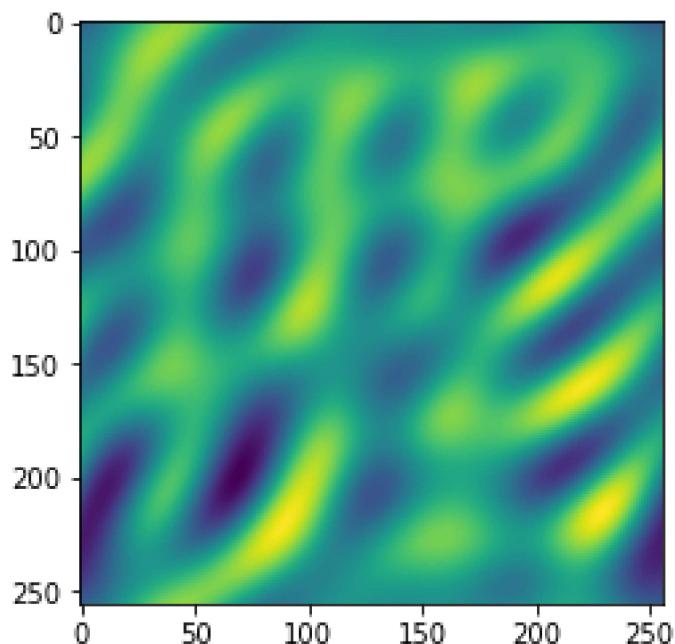
#Rendu 2D du bruit
```

```

fig = plt.figure(figsize = (4,4))
ax = plt.axes()
ax.imshow(low_noise)
plt.show()

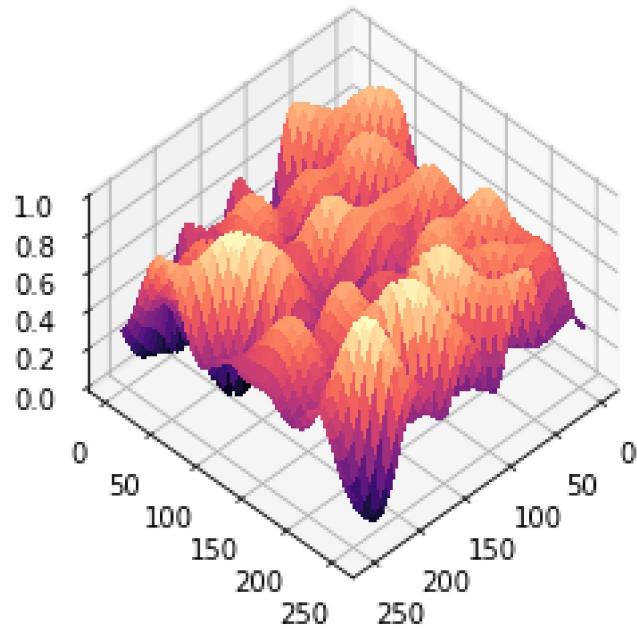
#rendu et normalisation du bruit pour la 3D
X, Y = np.mgrid[0:len(low_noise), 0:len(low_noise)]
low_noise = low_noise - low_noise.min()
low_noise = low_noise / low_noise.max()
fig = plt.figure(figsize = (4,4))
ax = plt.axes(projection ='3d')
ax.set_proj_type('ortho')
ax.view_init(45, 45)
plt.title('Bruit Random réduit')
ax.plot_surface(X, Y, low_noise, cmap="magma", linewidth=0, antialiased=False)

```

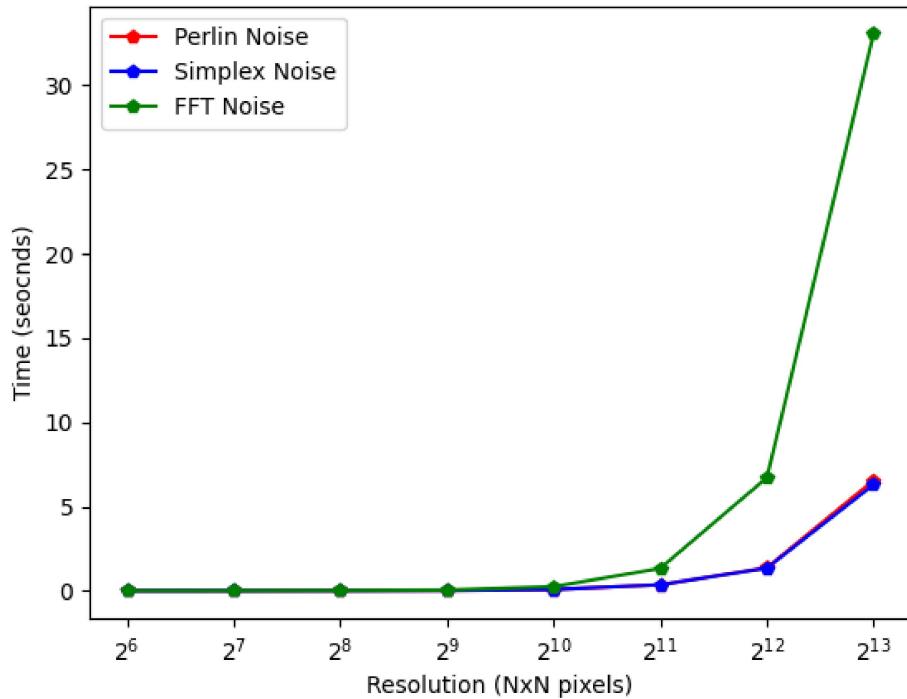


[4]: <mpl\_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f959e040820>

Bruit Random réduit



On finit par observer un bruit très similaire au bruit de Perlin, tellement qu'on ne voit pas la différence. Nous avons donc générée une alternative au bruit de Fourier. Pourquoi n'est-elle pas plus utilisée ? Car faire 2 transformées de Fourier demande bien plus d'opérations que celles utilisées pour le bruit de Perlin ou le bruit Simplex(version améliorée du bruit de Perlin(dev aussi par Ken Perlin)).



On observe qu'en effet plus notre taille est grande plus la méthode avec la TF va être cher en performance. Néanmoins pour des petites résolutions le temps de rendu est similaire, par exemple pour notre cas:  $512 = 2^8$ . Néanmoins pour une réelle utilisation les résolutions sont bien plus grande, d'où le grand besoin d'optimisations pour avoir du temps réel nécessaire dans les jeux vidéos par exemple.

# Perlin - Introduction to Domain Warping

December 2, 2022

## 1 Application du bruit de Perlin : Domain Warping

Un des **majeurs outils** qu'on peut utiliser pour avoir le résultat que l'on souhaite et très puissant est la **déformation** du bruit par une fonction / manipulation spatiale. C'est à dire que si on voit un bruit comme une fonction  $n_1(\vec{r})$  avec  $\vec{r}$  le vecteur position tel que  $\vec{r} = (x, y)$  alors faire un "**warp**" signifie faire :  $n_2(g(\vec{r}))$  avec  $g$  pouvant par exemple être lui même un bruit ou une fonction.

À l'aide de celà, on peut obtenir des rendus très **organiques** ou à l'inverse quelque chose de très "**machine**" : c'est à dire formes géométriques.

Ce chapitre va être certe **rapide et court** mais le concept présenté est **fondamental** et c'est celui qui permet d'agrandir le plus les possibilités avec le bruit de Perlin (et d'autres bruits).

### 1.1 Texture de géante gazeuse

Afin de montrer la puissance de cette méthode on va directement donner l'exemple avec  $g(\vec{r}) = n_2(\vec{r}) + \vec{r}$  avec  $n_2$  **un deuxième bruit de Perlin**. Nous allons utiliser un **bruit de Perlin** que l'on va appeler "**fondamental**" qui sera notre bruit que l'on va **déformer** et un autre **bruit** qui est nommé "**transf**" dans le code qui sera le bruit avec lequel on va déformer. On appliquera en plus certaines opérations déjà décrites dans la première partie afin d'améliorer le rendu. L'objectif ici est de donner un style très "**fluide**" à notre bruit, quelque chose qui est difficilement obtenable avec des opérations "basiques".

```
[3]: import numpy as np
import perlin2D as perlin
import matplotlib.pyplot as plt
from matplotlib import animation, rc
import matplotlib.cm as cm
rc('animation', html='jshtml')

#On commence par coder une fonction "warp" avec comme paramètres :
#fundamental : n_1
# "transf" : Bruit qui va nous servir pour la transformation = n_2
# vec : Coefficients pour avoir notre +r dans g(r)
def warp(fundamental, transf, vec = (1,1)):
    #On initialise notre variable que l'on va renvoyer à la fin
    result = np.zeros([shape,shape])

    #On normalise le bruit que l'on transforme si jamais il ne l'était pas
```

```

transf = transf - transf.min()
transf = transf / transf.max() - 0.000001
transf = np.floor(transf * shape)

#On applique l'opération que l'on souhaite pour tous les pixels de l'espace.
for x in range(shape):
    for y in range(shape):
        #Ici on reconnaît bien la forme  $f(n(x,y) + x + y)$ :
        result[x][y] = fundamental[int(transf[x][y]) + x * ↵
→vec[1]] [int(transf[x][y]) + y *vec[1]]

#On normalise le bruit obtenu
result = result - result.min()
result = result / result.max()
return result

```

On applique maintenant les opérations :

[5]: #Notre fondamental qui a une taille plus grande que "shape". Pour avoir des ↵ détails ça sera un bruit fractal à 3 octaves

```

fundamental = perlin.Perlin2D([2**11,2**11],[2**3,2**3]).fractal2D(3, 0.5, 1, 2)

#Notre taille finale
shape = 2**10
#La fonction d'interpolation qu'on va utiliser à la fin.
lerp = perlin.Perlin2D([shape,shape],[2**3,2**3]).lerp

#Le bruit que l'on va transformer (c'est une autre façon d'écrire un bruit ↵ fractal)
transf = perlin.Perlin2D([shape,shape],[2**2,2**1]).noise + 0.2*perlin.
    ↵Perlin2D([shape,shape],[2**3,2**1]).noise + 0.05*perlin.
    ↵Perlin2D([shape,shape],[2**5,2**5]).noise

#On fait un premier résultat où on fait ce qu'on a dit plus haut
result = warp(fundamental,transf)
#On travaille un peu plus notre bruit en refaisant le fondamental avec le ridge ↵ de notre précédent résultat (évidemment cette idée vient avec plusieurs ↵ essais, il est donc important que notre code de génération soit rapide afin ↵ de procéder à beaucoup de tests et adapter rapidement)
result2 = warp(fundamental,-1 * np.abs(result-0.5) + 1) + 0.05*perlin.
    ↵Perlin2D([shape,shape],[2**5,2**5]).noise
#On finalise le bruit en l'interpolant.
result3 = lerp(result2)

```

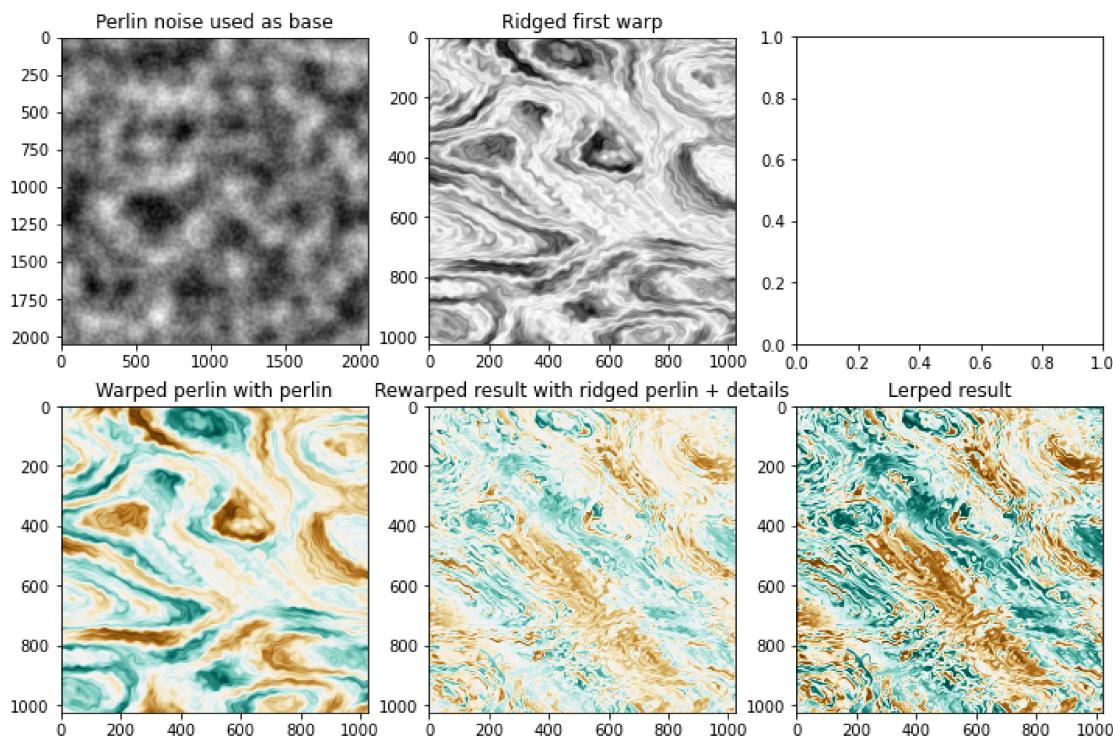
```

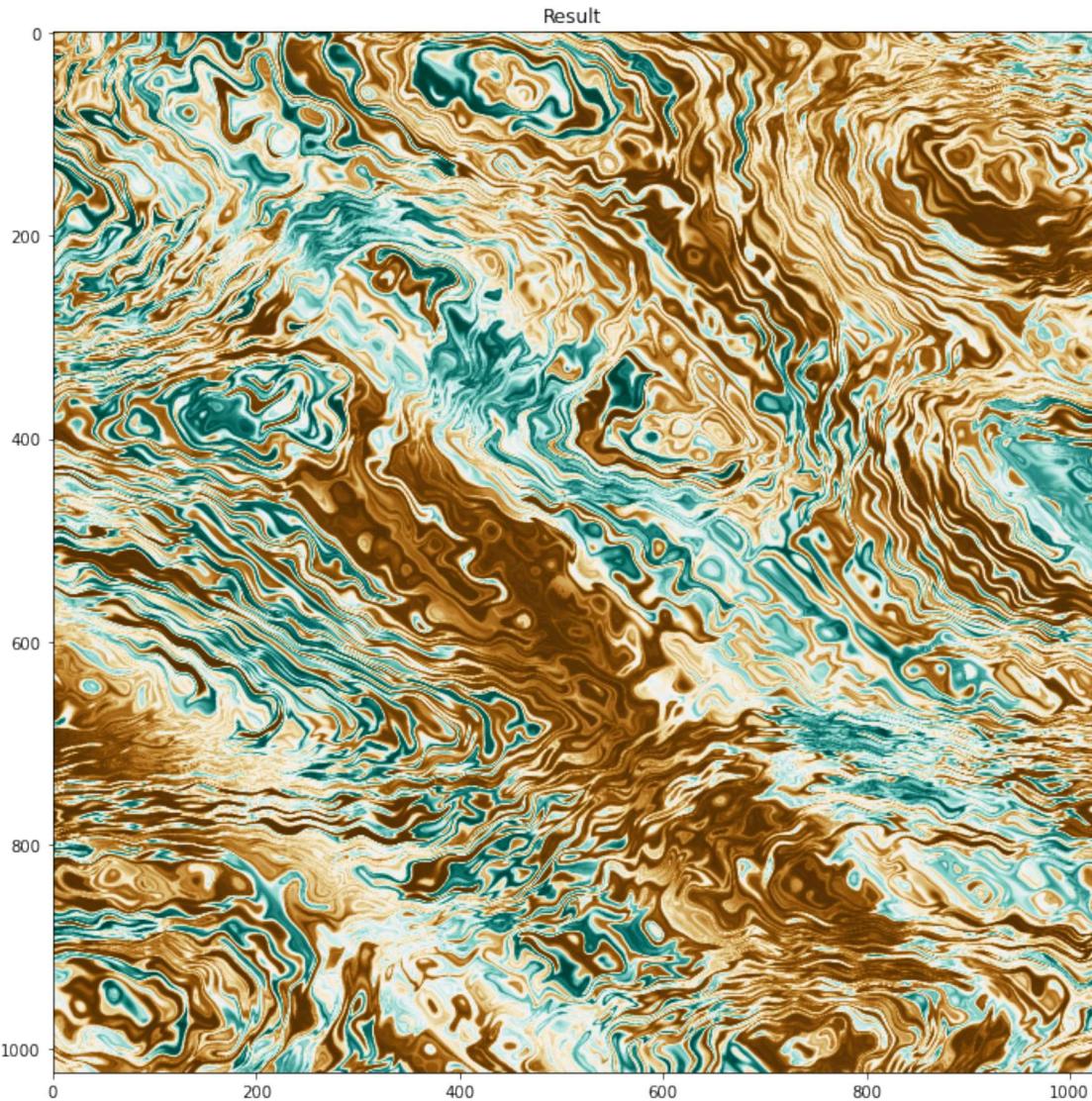
#Rendu du bruit par matplotlib; On souhaite afficher toutes les étapes que l'on a réalisée.
fig, ax = plt.subplots(2,3,figsize=(12,8))
ax[0][0].imshow(fundamental, cmap="gray")
ax[0][0].set_title("Perlin noise used as base")
ax[0][1].imshow(-1 * np.abs(result-0.5) + 1, cmap="gray")
ax[0][1].set_title("Ridged first warp")

ax[1][0].imshow(result, cmap="BrBG")
ax[1][0].set_title("Warped perlin with perlin")
ax[1][1].imshow(result2, cmap="BrBG")
ax[1][1].set_title("Rewarped result with ridged perlin + details")
ax[1][2].imshow(result3, cmap="BrBG")
ax[1][2].set_title("Lerped result")
plt.show()

#Le même dernier rendu mais en plus gros et avec plus de contraste (exponentiation).
fig, ax = plt.subplots(figsize=(12,12))
ax.imshow(result3**1.7, cmap="BrBG")
ax.set_title("Result")
plt.show()

```





Rajouter dans  $g(\vec{r})$  **un**  $+\vec{r}$  nous permet de rajouter un **effet de “bandes”** comme sur une **géante gazeuse**, si on warp que avec un **+x ou +y** alors on aura soit des **bandes verticales ou des bandes horizontales**. Mais ici on a des bandes en diagonales. C'est l'**utilité des coefficients** que l'on a mis dans la première fonction (“**vec**”) permettant d'**orienter** ces bandes comme on le souhaite.

# Bruit\_de\_Worley

December 2, 2022

## 1 Le bruit de Worley

Dans cette partie nous allons nous intéresser à un **autre bruit** produisant des résultats **très différents du bruit de Perlin**. De plus son algorithme de génération est très simple. Ce chapitre va aussi nous permettre de reprendre ce qu'on a fait pour Perlin.

### Plan 1. Première approche du bruit de Worley

- Bruit de Worley en 2D
  - Bruit de Worley en 3D
2. Optimisation et Applications simples
- Reécriture du code avec scipy
  - Approfondissement du bruit
  - Quelques applications
3. Essais divers
- Eau
  - Bruit Fractal
  - Domain Warping

### 1.1 Première approche du bruit de Worley

Le **bruit de Worley** se construit en générant un nombre de points à des coordonnées purement aléatoires dans un espace d'une certaine taille. Ensuite il suffit pour chaque pixel de l'espace de calculer la distance à chacun des points de l'espace et de garder la taille la plus proche et de la mettre dans une matrice(tenseur si 3D) de la taille de l'espace(on pourra prendre la 2nd distance la plus proche pour avoir un résultat différent...).

#### 1.1.1 Bruit de Worley en 2D

```
[1]: import math
import numpy as np
import matplotlib.pyplot as plt

#d = ordre / distance
def worley2d(shape,n,d=0):
    #n: Nombre de points placés aléatoirement dans l'espace
    if(d > n-1):
        d = n-1

    points = [] #liste de tous les points
```

```

for i in range (n): #Pour tous les points:
    x = np.random.randint(0,shape) #On crée une coordonnée x aléatoire
    y = np.random.randint(0,shape) #On crée une coordonnée y aléatoire
    coords = [x, y] # Coordonnées (x,y)
    points.append(coords) #Ajout des coordonées de chaque points dans la
    → liste "points"

M = np.zeros((shape,shape)) #On crée la matrice vide du bruit dans laquelle
→ on va placer la distance la plus courte à chaque point

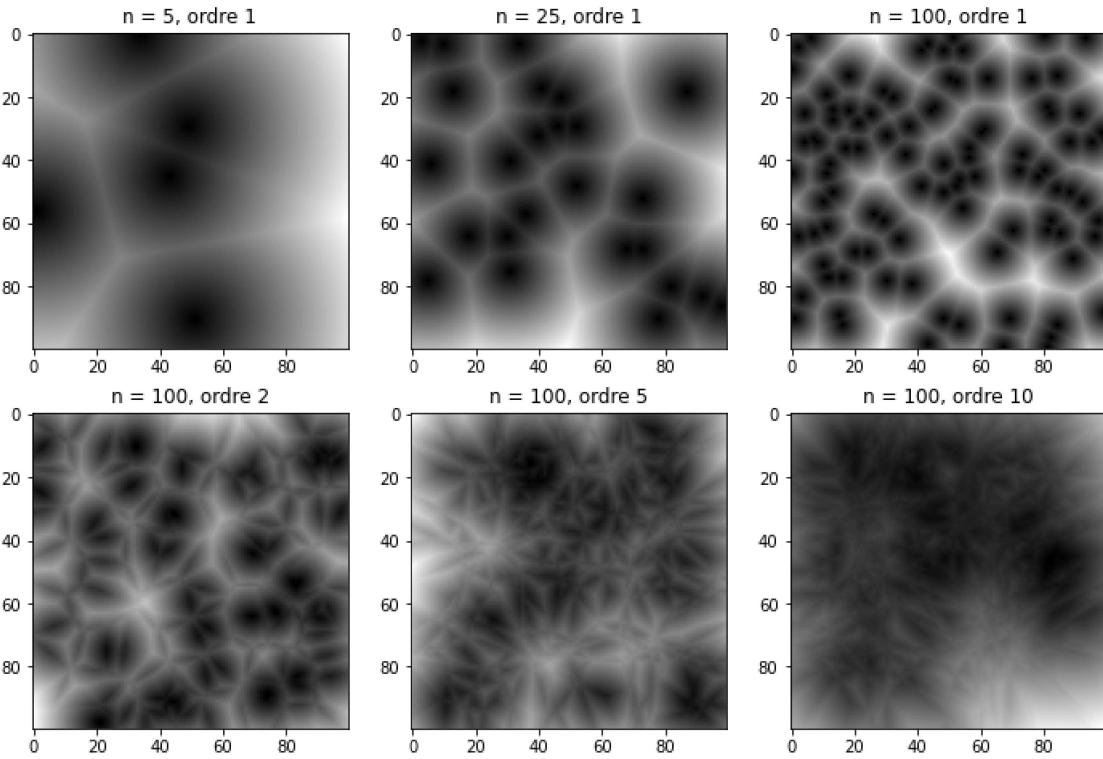
for i in range (0, shape): #on parcourt tous les x
    for j in range (0, shape): #on parcourt tous les y
        #distance min
        dist_min = []
        for p in points:
            dist_min.append(np.sqrt((p[0]-i)**2 + (p[1]-j)**2))
        dist_min.sort()
        value_min = dist_min[d]
        M[i][j] = value_min

return M

#Rendu matplotlib
fig, ax = plt.subplots(2,3,figsize=(12,8))
ax[0][0].imshow(worley2d(100,5,0), cmap="gray")
ax[0][0].set_title("n = 5, ordre 1")
ax[0][1].imshow(worley2d(100,25,0), cmap="gray")
ax[0][1].set_title("n = 25, ordre 1")
ax[0][2].imshow(worley2d(100,100,0), cmap="gray")
ax[0][2].set_title("n = 100, ordre 1")
ax[1][0].imshow(worley2d(100,100,1), cmap="gray")
ax[1][0].set_title("n = 100, ordre 2")
ax[1][1].imshow(worley2d(100,100,4), cmap="gray")
ax[1][1].set_title("n = 100, ordre 5")
ax[1][2].imshow(worley2d(100,100,9), cmap="gray")
ax[1][2].set_title("n = 100, ordre 10")

```

[1]: Text(0.5, 1.0, 'n = 100, ordre 10')



En premier lieu on observe un effet de “**cristallisation**” quand on prend une distance qui n'est pas la plus proche. Cette effet pourrait être utile pour un artiste. Ensuite pour l'ordre 1 donc la distance minimale, on a quelque chose ressemblant à des cellules d'où un nom très commun donné au bruit de Worley : Le **bruit cellulaire** / “Cellular Noise”

Et en effet la programmation de ce **bruit est rapide et efficace**(c'est à dire la seule chose pouvant être optimisée est le calcul des distances que l'on fera plus tard).

Passons maintenant en 3D pour faire quelques animations et voir comment ça évolue.

### 1.1.2 Bruit de Worley en 3D

Le principe est le même, il suffit de générer une **coordonnée supplémentaire et de calculer la distance sur un axe en plus**.

```
[3]: #d = ordre / distance
def worley3d(shape,n,d=0):
    #n: Nombre de points placés aléatoirement dans l'espace
    if(d > n-1):
        d = n-1

    points = [] #liste de tous les points
    for i in range (n): #Pour tous les points:
        x = np.random.randint(0,shape) #On crée une coordonnée x aléatoire
```

```

y = np.random.randint(0,shape) #On crée une coordonnée y aléatoire
z = np.random.randint(0,shape) #On crée une coordonnée z aléatoire
coords = [x, y, z] # Coordonnées (x,y)
points.append(coords) #Ajout des coordonées de chaque points dans la
→ liste "points"

M = np.zeros((shape,shape,shape)) #On crée la matrice vide du bruit dans
→ laquelle on va placer la distance la plus courte à chaque point

for i in range (0, shape): #on parcourt tous les x
    for j in range (0, shape): #on parcourt tous les y
        for k in range (0, shape): #on parcourt tous les z
            #distance min
            dist_min = []
            for p in points:
                dist_min.append(np.sqrt((p[0]-i)**2 + (p[1]-j)**2 +
→ (p[2]-k)**2))
            dist_min.sort()
            value_min = dist_min[d]
            M[i][j][k] = value_min

return M

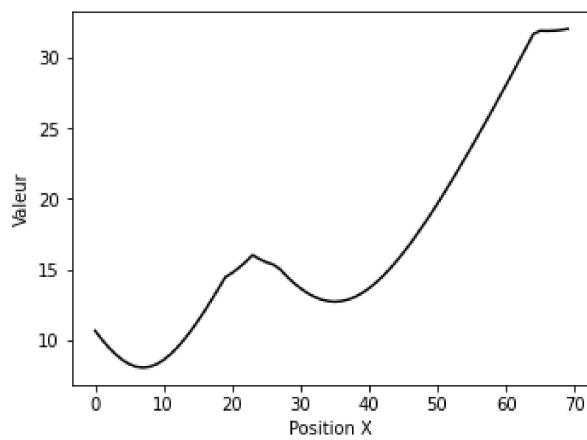
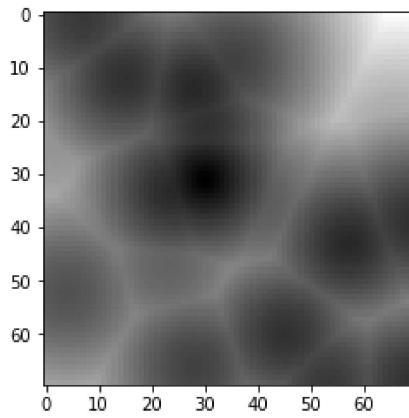
from matplotlib import animation, rc
import matplotlib.cm as cm
rc('animation', html='jshtml')

M = worley3d(70, 100, 0)

#Rendu
fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].imshow(M[0], cmap='gray')
ax[1].plot(M[0][0], color="black")
plt.xlabel("Position X")
plt.ylabel("Valeur")
plt.show()

fig1 = plt.figure()
l = plt.imshow(M[0], cmap="gray", interpolation='quadric', animated=True)
plt.colorbar()
plt.close()
animation.FuncAnimation(fig1, lambda p: l.set_data(M[p]), frames=70,
→ interval=100)

```



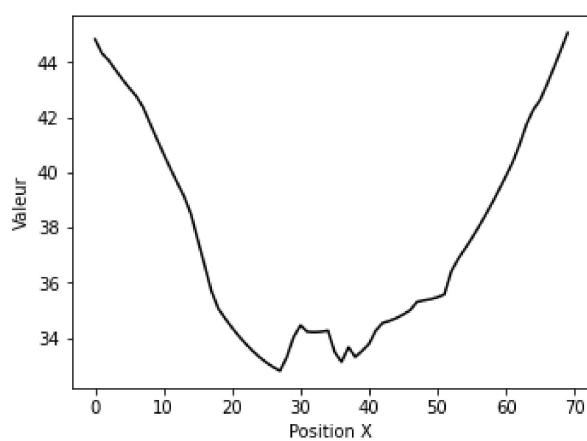
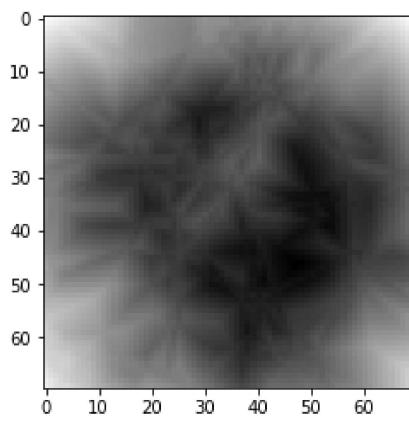
[3]: <matplotlib.animation.FuncAnimation at 0x7f016d1dc130>

Maintenant à l'**ordre 10**:

```
[4]: M = worley3d(70, 100, 9)

fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].imshow(M[0], cmap='gray')
ax[1].plot(M[0][0], color="black")
plt.xlabel("Position X")
plt.ylabel("Valeur")
plt.show()

fig1 = plt.figure()
l = plt.imshow(M[0], cmap="gray", interpolation='quadric', animated=True)
plt.colorbar()
plt.close()
animation.FuncAnimation(fig1, lambda p: l.set_data(M[p]), frames=70, interval=100)
```



```
[4]: <matplotlib.animation.FuncAnimation at 0x7f016cda9be0>
```

## 1.2 Optimisation et Applications simples

### 1.2.1 Reécriture du code avec scipy

Un moyen d'optimiser le calcul des distances est d'organiser les points non en tableau mais **en arbre**. C'est à dire que les points les proches seront sur une branche et les plus éloignés sur une autre. Donc pour chaque pixel on calcule la distance aux points sur notre branche. On va aussi en profiter pour organiser le bruit 3D dans un objet.

```
[1]: import numpy as np
import scipy.spatial as sp
import matplotlib.pyplot as plt
import random
import math
from matplotlib import animation, rc
import matplotlib.cm as cm
rc('animation', html='jshtml')

#Méthode 1 via scripy et numpy
def worley2D(res, density, n=0):

    points = [[np.random.randint(0, res), np.random.randint(0, res)] for _ in
              range(density)]

    #Les coo qu'on aura besoin
    coord = np.dstack(np.mgrid[0:res, 0:res])
    #Création de l'arbre avec scipy
    tree = sp.cKDTree(points)
    #On calcule la distance avec la méthode associée à l'arbre
    dist = tree.query(coord, workers=-1)[n]

    #Normalise le bruit
    return dist / dist.max()

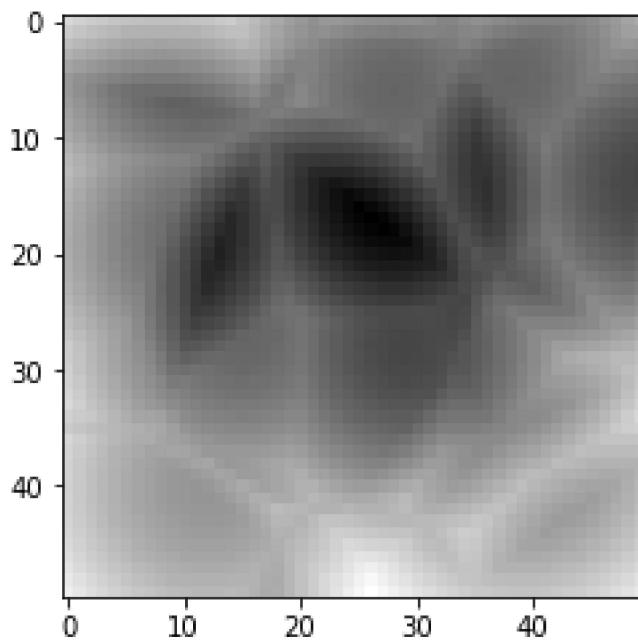
#Méthode 2 sans rien
class Worley3D:
    #Constructeur de l'objet
    def __init__(self, res, density, n_dis=0):
        self.res = res
        self.density = density
        #On génère les points
        self.generatePoints()
        #On calcule le bruit
```

```

        self.calculateNoise(n_dis)
    def generatePoints(self):
        self.points = [(random.randint(0, self.res), random.randint(0, self.
→res), random.randint(0, self.res)) for x in range(self.density)]
    def calculateNoise(self, n_dist):
        self.data = [[[sorted([math.sqrt((p[0]-x)**2 + (p[1]-y)**2 +_
→(p[2]-z)**2) for p in self.points])
            [n_dist] for x in range(self.res)] for y in range(self.
→res)]for z in range(self.res)]
        self.data = np.array(self.data)
        self.data = self.data / self.data.max()
    #Si on veut afficher le bruit 2D ou 3D
    def show2DNoise(self, index=0):
        fig, ax = plt.subplots(figsize=(12,4))
        ax.imshow(self.data[index], cmap='gray')
        plt.show()
    def show3DNoise(self, frames=40, interval=100):
        fig1 = plt.figure()
        l = plt.imshow(self.data[0], cmap="gray", interpolation='quadric',_
→animated=True)
        plt.colorbar()
        animation.FuncAnimation(fig1, lambda p: l.set_data(self.data[p]),_
→frames=frames, interval=interval)

noise = Worley3D(50,50,1)
noise.show2DNoise()

```

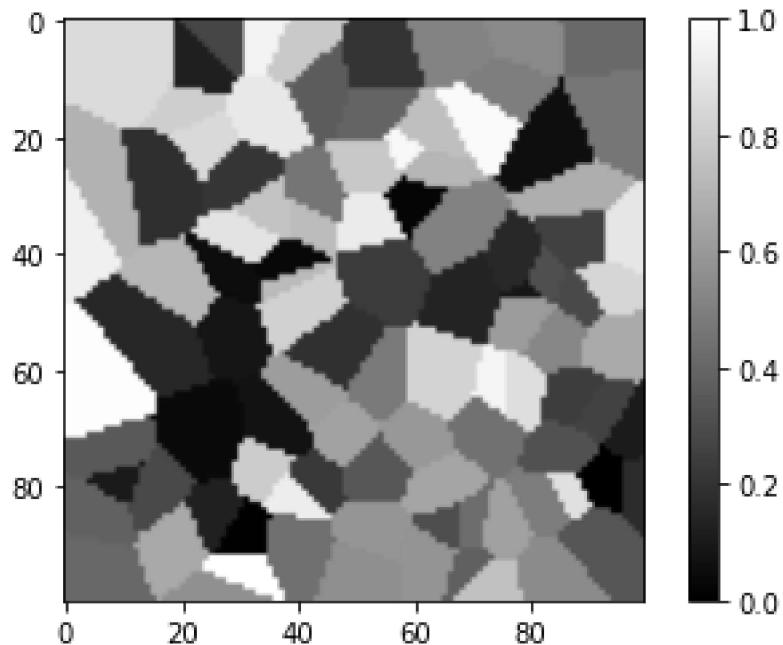


### 1.2.2 Approfondissement du bruit

Avec la méthode de l'arbre on peut aussi afficher ce qu'on appelle un : **Diagramme de Voronoi**(pavage cellulaire):

```
[6]: plt.imshow(worley2D(100,100,1), cmap="gray")
plt.colorbar()
```

```
[6]: <matplotlib.colorbar.Colorbar at 0x7f0167766220>
```



### 1.2.3 Quelques applications

Essayons d'appliquer la méthode du “**Ridge**”(Défini lors de la manipulation du bruit de Perlin) pour le bruit de Worley et de voir ce que ça donne:

```
[16]: shape = 50
#On calcule le bruit
M = Worley3D(shape,shape,0).data

#On calcule le ridge et on le normalise
M_ridge = (-1 * np.abs(M-0.3)+1) #M2 matrice normalisée et application du ridge
→ en passant par la valeur absolue (pour obtenir des sphères), même principe
→ que le schéma du sinus
M_ridge = M_ridge - M_ridge.min()
```

```

M_ridge = M_ridge / M_ridge.max()

#Rendu du bruit en 2D et 1D
fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].set_title("Ridged Worley : Offset 0.3")
ax[0].imshow(M_ridge[0], cmap='gray')
ax[1].plot(M_ridge[0][0], color="black")
plt.xlabel("Position X")
plt.ylabel("Valeur")
plt.show()

#Rendu d'une déformation d'un plan par le bruit
X,Y = np.mgrid[0:shape,0:shape] #on crée une grille
fig = plt.figure(figsize = (6,6))
ax = plt.axes(projection ='3d')
ax.set_title("Déformation d'un plan par le bruit ridged de Worley")
ax.set_proj_type('ortho')
ax.view_init(45, 45)
ax.set_zlim3d(0,1.3)
ax.plot_surface(X, Y, M_ridge[0], cmap="gray", linewidth=2, antialiased=True)

#Fonction déjà décrite dans le chapitre sur le bruit de Perlin : Permet de
→réarranger nos paramètres pour qu'ils soient compatibles avec la fonction de
→matplotlib
def rearrange_list(pic, val, infe):
    X = []
    Y = []
    Z = []
    values = []
    for k in range(len(pic)):
        for j in range(len(pic[k])):
            for i in range (len(pic[k][j])):
                value = pic[k][j][i]
                if((infe and (value < val)) or (not(infe) and (value > val))):
                    values.append(value)
                    X.append(i / len(pic[k][j]))
                    Y.append(j / len(pic[k]))
                    Z.append(k / len(pic))
    return (X,Y,Z,values)

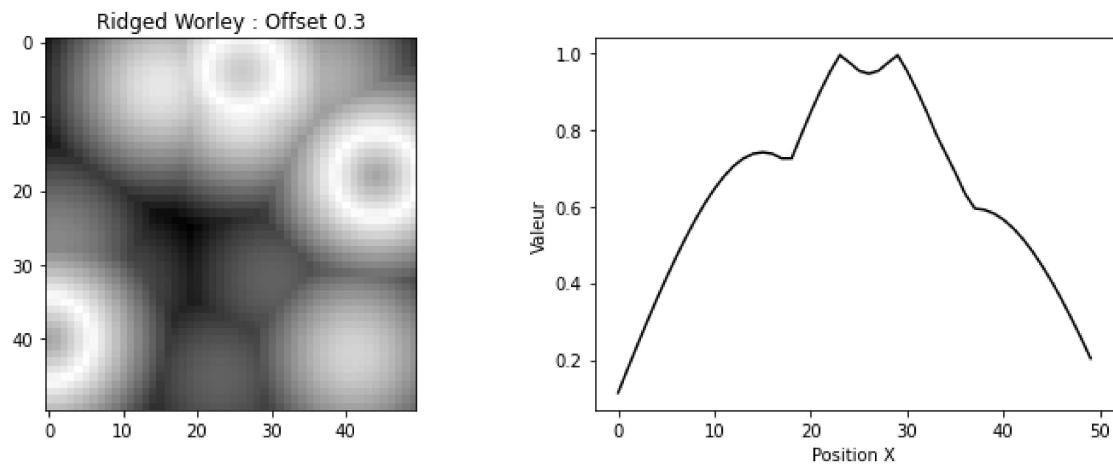
#Rendu 3D du bruit en enlevant les valeurs les plus hautes (pour obtenir des
→sphères)
X,Y,Z,values = rearrange_list(M_ridge,0.8,False)#1-0,8=0.2 est le rayon des
→sphères
fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.view_init(45, 45)

```

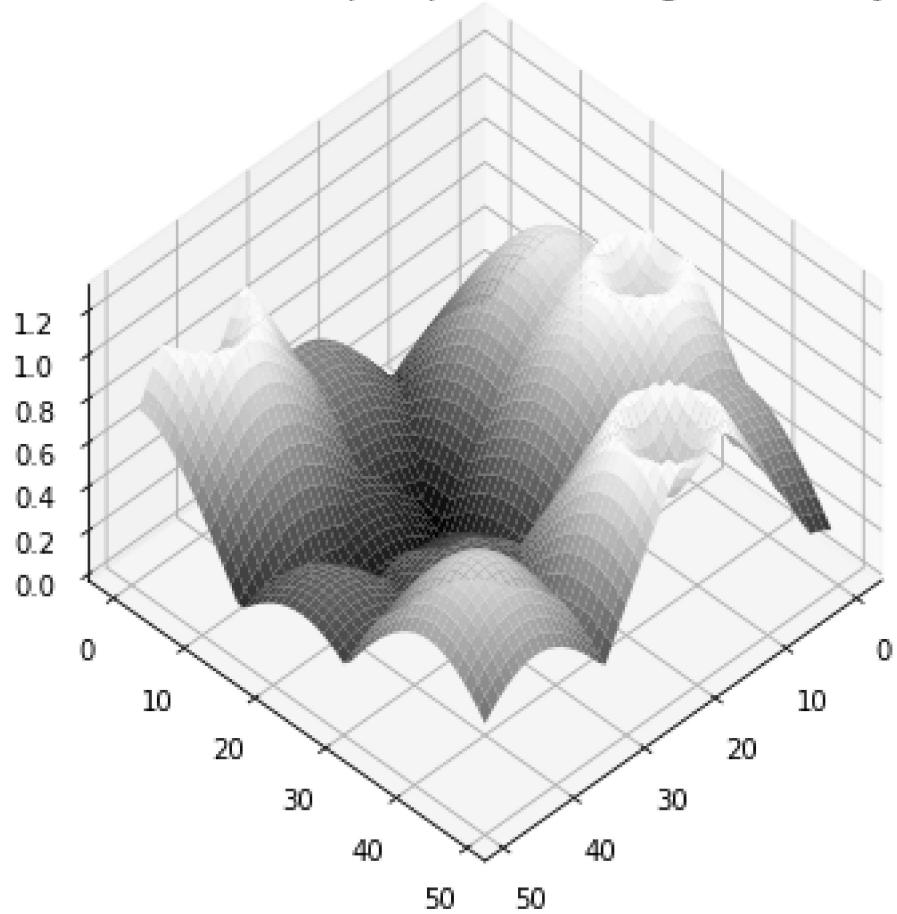
```

ax.scatter3D(X, Y, Z, c=values, cmap='gray')
ax.set_title("Rendu 3D du bruit de Worley")
plt.show() #On remarque que les points sont au centre des sphères

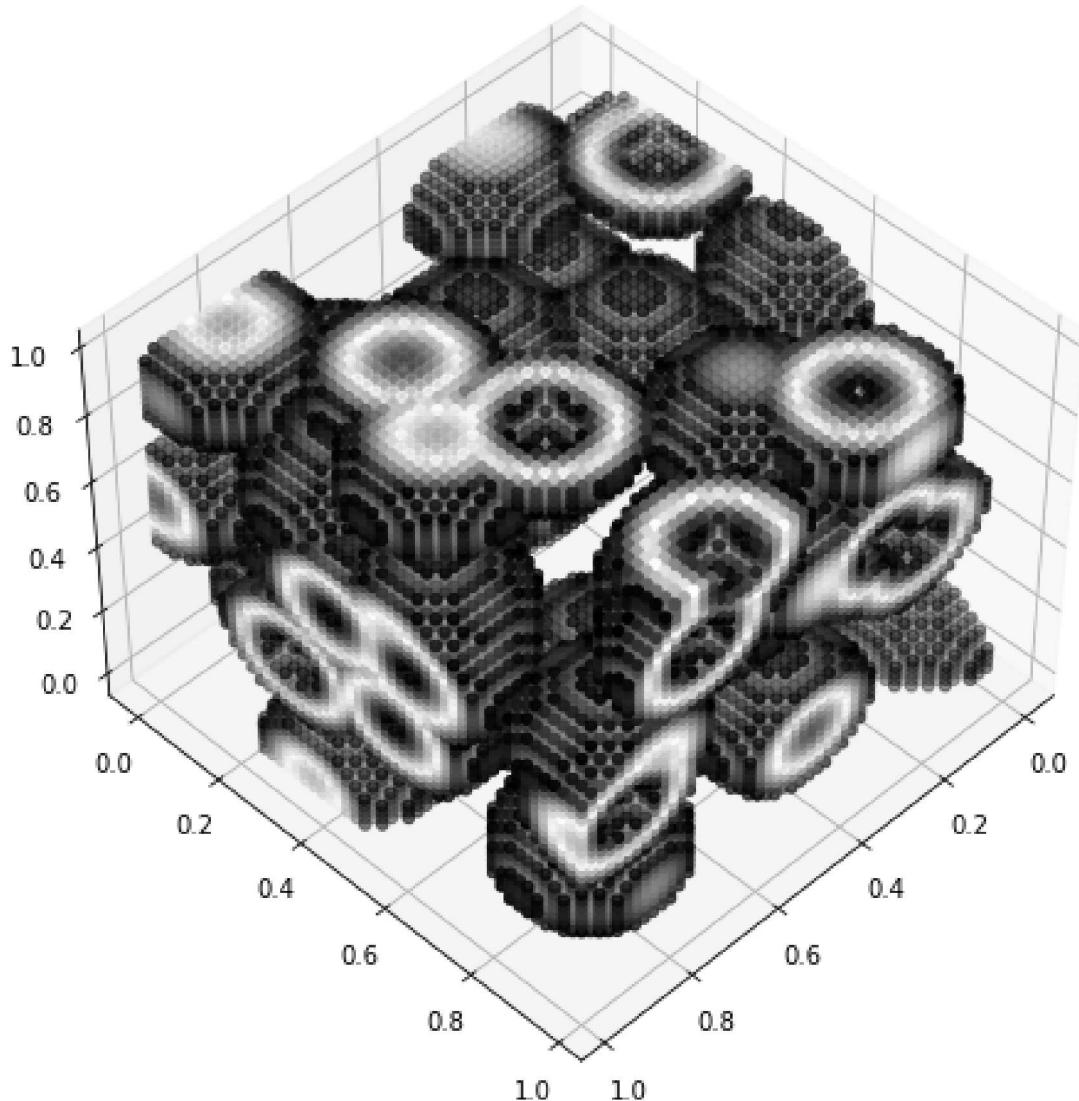
```



Déformation d'un plan par le bruit ridged de Worley



Rendu 3D du bruit de Worley



### 1.3 Essaies diverses

#### 1.3.1 Eau

Réaliser une première itération d'un shader d'eau peut se faire en une étape avec le bruit de Worley. Evidemment les artistes travaillent bien plus cet effet en lui rajoutant beaucoup de choses. Pour réaliser de l'eau on va rendre la déformation d'un plan avec le bruit de worley et l'animer avec un axe temps + une translation sur un axe.

```
[41]: from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt

#On, a besoin de bouger dans le bruit donc shape > taille qu'on va rendre
shape = 220
pic_worley = Worley3D(shape,50).data

def f(x, y, i):
    #On rend une taille de 100x100 dans un espace de 500x500
    return np.zeros([100,100]) + pic_worley[i][0:100,i:100+i]

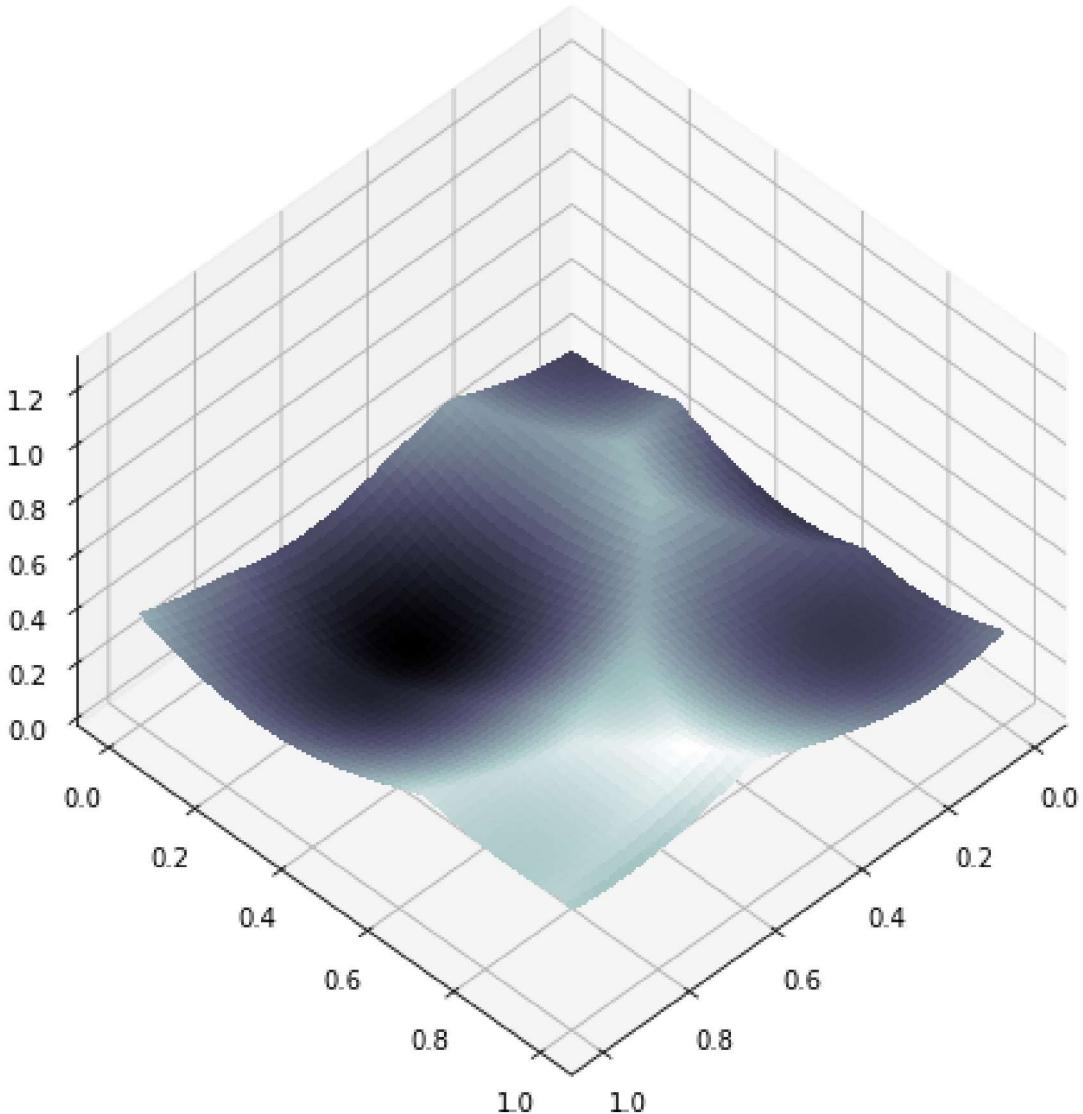
#Rendu 3D
x = np.linspace(0, 1, 100)
y = np.linspace(0, 1, 100)
X, Y = np.meshgrid(x, y)
Z = f(X, Y, 0)

fig = plt.figure(figsize = (8,8))
ax = plt.axes(projection ='3d')
ax.set_proj_type('ortho')
ax.view_init(45, 45)
ax.set_zlim3d(0,1.3)
ax.plot_surface(X, Y, Z, cmap="bone", linewidth=2, antialiased=True)

def update(p):
    ax.clear()
    ax.set_zlim3d(0,1.3)
    ax.plot_surface(X, Y, f(X,Y,p), cmap="bone", linewidth=0, antialiased=False)

animation.FuncAnimation(fig, update, frames=100, interval=50)
```

[41]: <matplotlib.animation.FuncAnimation at 0x7f106271fb50>



### 1.3.2 Bruit Fractal

Essayons maintenant de faire la **somme** de plusieurs bruits de **Worley** à **différentes fréquences**(= nombre de points) et différentes amplitudes. (= **Fractal**, la méthode qu'on avait appliquée pour le bruit de Perlin).

```
[39]: def fractal2D(shape, density, voronoi=0, octaves=1, persistance=0.5, u
→exponentiation=1, lacunarity=1, norm=True, noise = []):
    #Si jamais on n'a pas mis de bruit en argument de la fonction, on crée une u
→matrice
    if(len(noise) == 0):
        noise = np.zeros([shape,shape])
    frequency = 1
```

```

amplitude = 1
#Tous les paramètres sont expliqués plus bas
for _ in range(octaves):
    noise += amplitude * worley2D(shape, density*frequency, voronoi)
    frequency *= lacunarity
    amplitude *= persistance
noise = noise**exponentiation
#On normalise le bruit
return noise/noise.max() if(norm) else noise

shape = 2**11

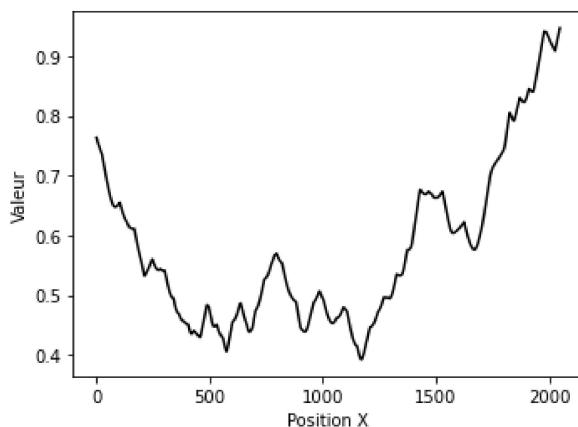
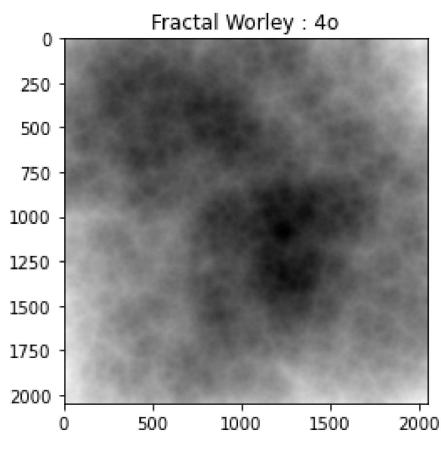
M = fractal2D(shape,3,0,4,0.5,1,10)

fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].set_title("Fractal Worley : 4o")
ax[0].imshow(M, cmap='gray')
ax[1].plot(M[0], color="black")
plt.xlabel("Position X")
plt.ylabel("Valeur")
plt.show()

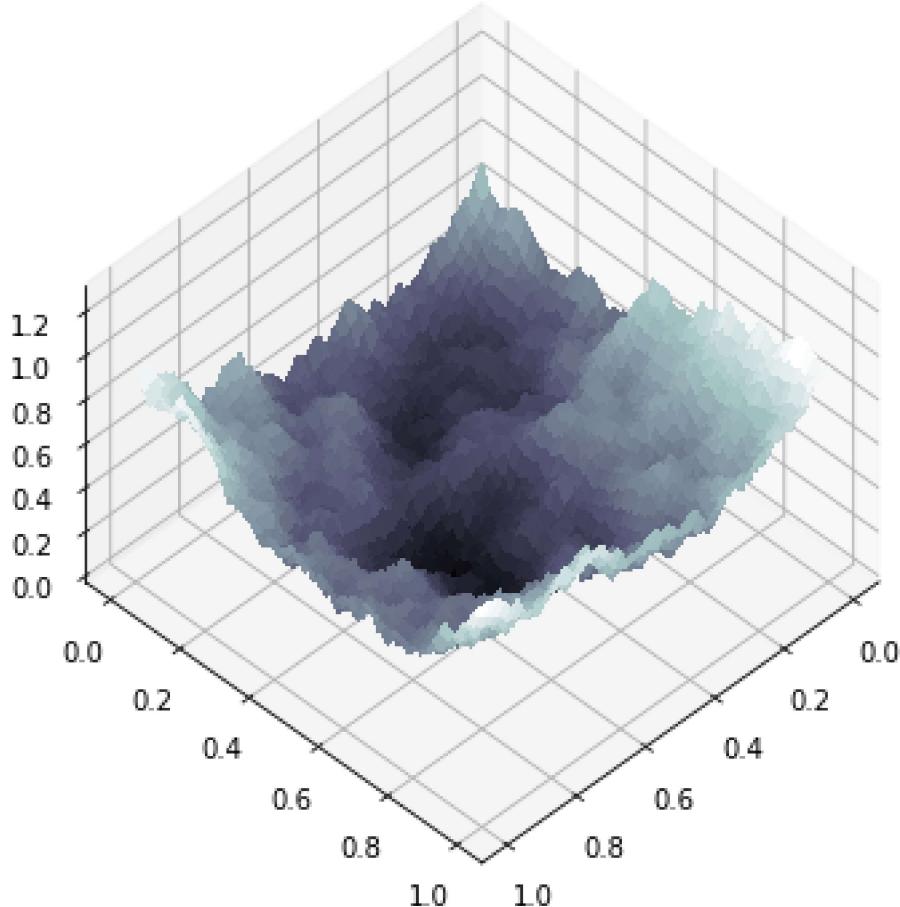
#Rendu 3D
x = np.linspace(0, 1, shape)
y = np.linspace(0, 1, shape)
X, Y = np.meshgrid(x, y)

fig = plt.figure(figsize = (6,6))
ax = plt.axes(projection ='3d')
ax.set_proj_type('ortho')
ax.view_init(45, 45)
ax.set_zlim3d(0,1.3)
ax.plot_surface(X, Y, M, cmap="bone", linewidth=0, antialiased=False)

```



[39]: <mpl\_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f1062baa730>



On pourrait utiliser la version fractale comme **base de nuage** par exemple. On remarque cependant que c'est vite limité si on veut l'appliquer pour générer un terrain. En effet on **remarque très bien des formes de cercles/demi-sphères** dans la déformation. Néanmoins son aspect est **très différent** de celui de **Perlin** donc le bruit de Worley permet de **combler certaines lacunes** du bruit de Perlin/ de faire des choses plus facilement. Beaucoup d'artistes utilisent d'ailleurs le bruit de Worley pour mettre en évidence le contour d'un objet 3D à l'aide de quelques opérations supplémentaires.

### 1.3.3 Domain Warping

Lors du dernier chapitre, nous avons vu un outil très utile qui est la **déformation spatiale du bruit par un autre bruit**. On va essayer de l'appliquer sur le bruit de Worley. On va combiner les cas possibles si on déforme un bruit de Perlin par un bruit de Worley ou l'inverse et ça avec et sans dépendance en x et y.

```
[49]: import perlin2D as perlin

#Rappel de la fonction warp :
#fundamental : n_1
# "transf" : Bruit qui va nous servir pour la transformation = n_2
# vec : Coefficients pour avoir notre +r dans g(r)
def warp(fundamental, transf, vec = (0,0)):
    #On initialise notre variable que l'on va renvoyer à la fin
    result = np.zeros([shape,shape])

    #On normalise le bruit que l'on transforme si jamais il ne l'était pas
    transf = transf - transf.min()
    transf = transf / transf.max() -0.000001
    transf = np.floor(transf * shape)

    #On applique l'opération que l'on souhaite pour tous les pixels de l'espace.
    for x in range(shape):
        for y in range(shape):
            #Ici on reconnaît bien la forme f(n(x,y) + x + y):
            result[x][y] = fundamental[int(transf[x][y]) + x *vec[0]
                                         +vec[1]] [int(transf[x][y]) + y *vec[1]]

    #On normalise le bruit obtenu
    result = result - result.min()
    result = result / result.max()
    return result

fundamental_worley = fractal2D(2**11,3,0,4,0.5,1,10)
fundamental_perlin = perlin.Perlin2D([2**11,2**11],[2**3,2**3]).fractal2D(3, 0.
                           ↪5, 1, 2)

#Notre taille finale
shape = 2**10

print("SANS DEPENDANCE EN X ET Y")

#Calcul
transf_worley = worley2D(shape,50,0)
transf_perlin = perlin.Perlin2D([shape,shape],[2**3,2**3]).noise

worley_worley = warp(fundamental_worley,transf_worley)
perlin_worley = warp(fundamental_perlin,transf_worley)
worley_perlin = warp(fundamental_worley,transf_perlin)

worley_worley2 = warp(fundamental_worley,worley_worley)
perlin_worley2 = warp(fundamental_perlin,perlin_worley)
```

```
worley_perlin2 = warp(fundamental_worley,worley_perlin)

worley_worley3 = warp(fundamental_worley,worley_worley2)
perlin_worley3 = warp(fundamental_perlin,perlin_worley2)
worley_perlin3 = warp(fundamental_worley,worley_perlin2)

#Rendu
fig, ax = plt.subplots(3,3, figsize=(12,12))
ax[0][0].imshow(worley_worley)
ax[0][0].set_title("Worley / Worley")
ax[0][1].imshow(worley_worley2)
ax[0][1].set_title("Rewarp result")
ax[0][2].imshow(worley_worley3)
ax[0][2].set_title("Rewarp*2 result")

ax[1][0].imshow(perlin_worley)
ax[1][0].set_title("Perlin / Worley")
ax[1][1].imshow(perlin_worley2)
ax[1][1].set_title("Rewarp result")
ax[1][2].imshow(perlin_worley3)
ax[1][2].set_title("Rewarp*2 result")

ax[2][0].imshow(worley_perlin)
ax[2][0].set_title("Worley / Perlin")
ax[2][1].imshow(worley_perlin2)
ax[2][1].set_title("Rewarp result")
ax[2][2].imshow(worley_perlin3)
ax[2][2].set_title("Rewarp*2 result")
plt.show()

print("AVEC WARP DEPENDANT DE X ET Y")
#Calcul
worley_worley = warp(fundamental_worley,transf_worley, (1,1))
perlin_worley = warp(fundamental_perlin,transf_worley, (1,1))
worley_perlin = warp(fundamental_worley,transf_perlin, (1,1))

worley_worley2 = warp(fundamental_worley,worley_worley, (1,1))
perlin_worley2 = warp(fundamental_perlin,perlin_worley, (1,1))
worley_perlin2 = warp(fundamental_worley,worley_perlin, (1,1))

worley_worley3 = warp(fundamental_worley,worley_worley2, (1,1))
perlin_worley3 = warp(fundamental_perlin,perlin_worley2, (1,1))
worley_perlin3 = warp(fundamental_worley,worley_perlin2, (1,1))

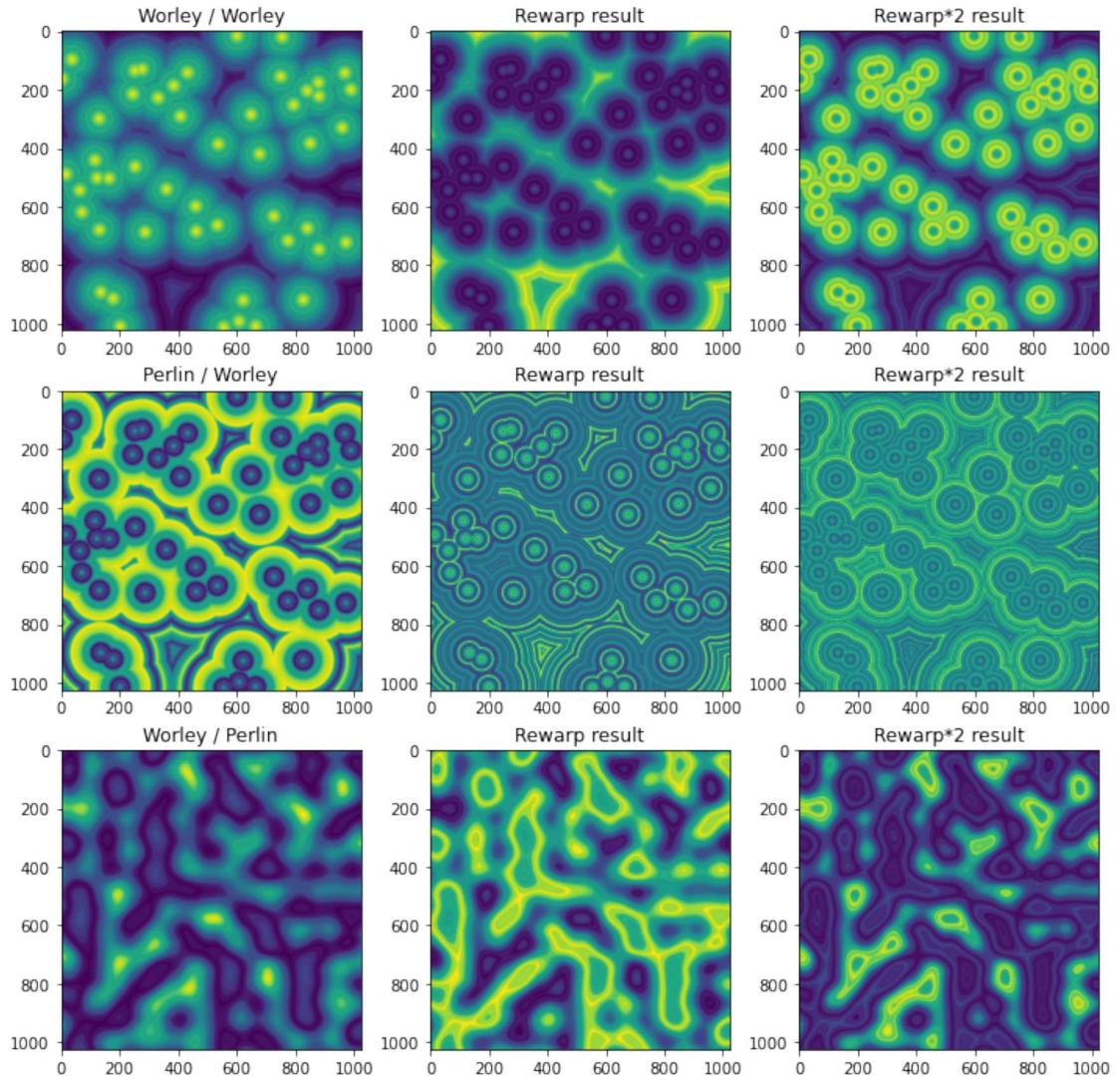
#Rendu
fig, ax = plt.subplots(3,3, figsize=(12,12))
ax[0][0].imshow(worley_worley)
```

```
ax[0][0].set_title("Worley / Worley")
ax[0][1].imshow(worley_worley2)
ax[0][1].set_title("Rewarp result")
ax[0][2].imshow(worley_worley3)
ax[0][2].set_title("Rewarp*2 result")

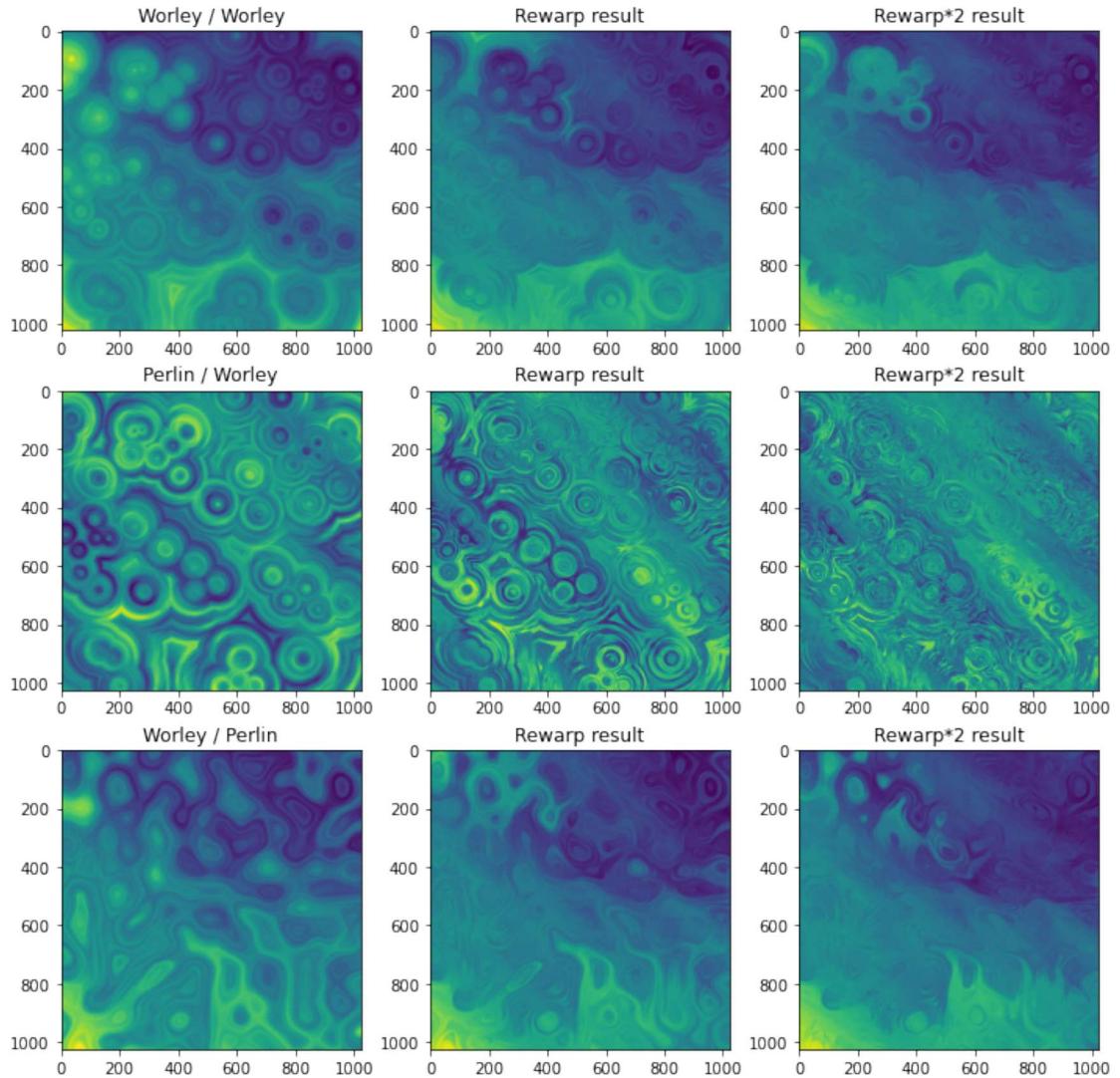
ax[1][0].imshow(perlin_worley)
ax[1][0].set_title("Perlin / Worley")
ax[1][1].imshow(perlin_worley2)
ax[1][1].set_title("Rewarp result")
ax[1][2].imshow(perlin_worley3)
ax[1][2].set_title("Rewarp*2 result")

ax[2][0].imshow(worley_perlin)
ax[2][0].set_title("Worley / Perlin")
ax[2][1].imshow(worley_perlin2)
ax[2][1].set_title("Rewarp result")
ax[2][2].imshow(worley_perlin3)
ax[2][2].set_title("Rewarp*2 result")
plt.show()
```

SANS DEPENDANCE EN X ET Y



AVEC WARP DEPENDANT DE X ET Y



On en conclut que le bruit de Worley étant très différent du bruit de Perlin il permet de nous constituer une base supplémentaire pour la formation d'effets.

# Conclusion

December 2, 2022

## 1 Conclusion

### 1.1 Un point sur les mathématiques

Avant de terminer avec la conclusion ce projet, donner un peu de théorie peut-être une bonne idée.

Lors de la présentation du bruit de Perlin, nous avions parlé de version “fractal”(somme de plusieurs bruits à différentes fréquences et amplitudes). Ce terme “fractal” vient d’un terme plus large : “Mouvement brownien fractionnaire”(mbf en français : fbm en anglais, c'est pour cela que si on lit pas mal de documents anglais sur les bruits on va retrouver comme nom de fonction : fbm(x,y)). Ce terme provient du domaine des probabilités et est une généralisation des mouvements browniens. Et c'est dans ces mouvements qu'apparaissent le pseudo-aléatoire. Ce domaine étant très large( et je doute d'avoir les capacités pour le comprendre correctement et donc pour l'expliquer) nous ne pouvons pas le voir ici. Cependant les ffbm ont une définition mathématique et donc peuvent être générés via des méthodes se basant sur la théorie et les formules (donc en général + complexe et + long à générer). Donc c'est quelque chose de purement mathématique ne trouvant que dans de rares cas des applications par les artistes dans le domaine de la génération pseudo-aléatoire. Mais cet objet défini mathématiquement a donc des propriétés mathématiques propres pouvant être utiles pour générer de nouveaux bruits et bien +. Liens pour en savoir + : - [https://en.wikipedia.org/wiki/Multifractal\\_system](https://en.wikipedia.org/wiki/Multifractal_system) - [https://en.wikipedia.org/wiki/Fractional\\_Brownian\\_motion](https://en.wikipedia.org/wiki/Fractional_Brownian_motion) - [https://en.wikipedia.org/wiki/Brownian\\_motion](https://en.wikipedia.org/wiki/Brownian_motion)

Un exemple d'intégration : Compréhensible niveau code, beaucoup moins niveau théorique. (Ici surface brownien : **Source** <https://gist.github.com/radarsat1/6f8b9b50d1ecd2546d8a765e8a144631> ; **Auteur** : <https://gist.github.com/radarsat1>).

```
[4]: import numpy as np
# embedding of covariance function on a [0,R]^2 grid
def rho(x,y,R,alpha):

    if alpha <= 1.5:
        # alpha=2*H, where H is the Hurst parameter
        beta = 0
        c2 = alpha/2
        c0 = 1-alpha/2
    else:
        # parameters ensure piecewise function twice differentiable
        beta = alpha*(2-alpha)/(3*R*(R**2-1))
        c2 = (alpha-beta*(R-1)**2*(R+2))/2
```

```

c0 = beta*(R-1)**3+1-c2

# create continuous isotropic function
r = np.sqrt((x[0]-y[0])**2+(x[1]-y[1])**2)
if r<=1:
    out=c0-r**alpha+c2*r**2
elif r<=R:
    out=beta*(R-r)**3/r
else:
    out=0

return out, c0, c2

# The main control is the Hurst parameter: H should be between 0 and
# 1, where 0 is very noisy, and 1 is smoother.
def brownian_surface(N=1000, H=0.98):
    R = 2 # [0,R]^2 grid, may have to extract only [0,R/2]^2

    # size of grid is m*n; covariance matrix is m^2*n^2
    M = N

    # create grid for field
    tx = np.linspace(0, R, M)
    ty = np.linspace(0, R, N)
    rows = np.zeros((M,N))

    for i in range(N):
        for j in range(M):
            # rows of blocks of cov matrix
            rows[j,i] = rho([tx[i],ty[j]],
                            [tx[0],ty[0]],
                            R, 2*H)[0]

    BlkCirc_row = np.vstack(
        [np.hstack([rows, rows[:, -1:-1]]),
         np.hstack([rows[-1:-1, :], rows[-1:-1, -1:-1]])])

    # compute eigen-values
    lam = np.real(np.fft.fft2(BlkCirc_row))/(4*(M-1)*(N-1))
    lam = np.sqrt(lam)

    # generate field with covariance given by block circular matrix
    Z = np.vectorize(complex)(np.random.randn(2*(M-1), 2*(M-1)),
                               np.random.randn(2*(M-1), 2*(M-1)))
    F = np.fft.fft2(lam*Z)
    F = F[:M, :N] # extract sub-block with desired covariance

```

```

out,c0,c2 = rho([0,0],[0,0],R,2*H)

field1 = np.real(F) # two independent fields
field2 = np.imag(F)
field1 = field1 - field1[0,0] # set field zero at origin
field2 = field2 - field2[0,0] # set field zero at origin

# make correction for embedding with a term c2*r^2
field1 = field1 + np.kron(np.array([ty]).T * np.random.randn(),
                          np.array([tx]) * np.random.randn())*np.sqrt(2*c2)
field2 = field2 + np.kron(np.array([ty]).T * np.random.randn(),
                          np.array([tx]) * np.random.randn())*np.
→sqrt(2*c2)
X,Y = np.meshgrid(tx,ty)

field1 = field1[:N//2, :M//2]
field2 = field2[:N//2, :M//2]
return (field1, field2)

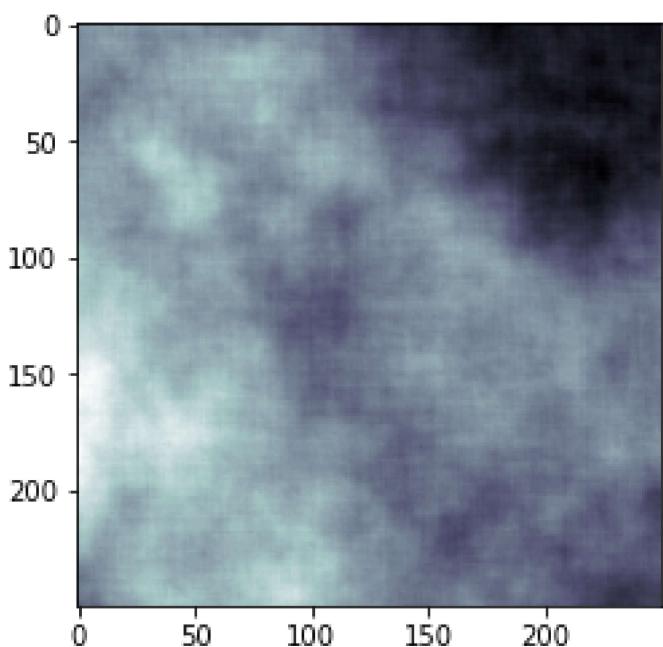
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

field = brownian_surface(N=500, H=0.8)

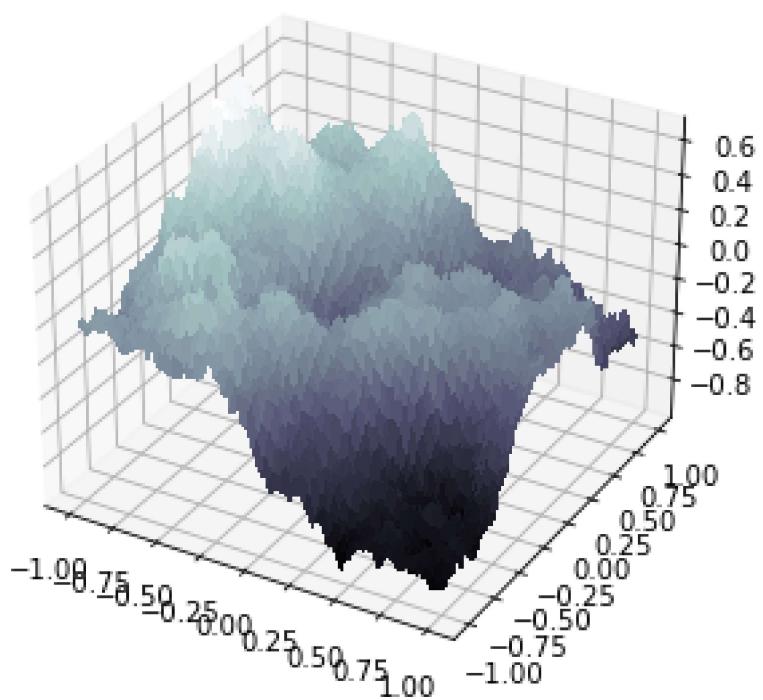
plt.imshow(field[0], cmap="bone")
plt.show()

fig = plt.figure(figsize = (5,5))
ax = fig.add_subplot(111, projection='3d')
X, Y = np.meshgrid(np.linspace(-1,1,field[0].shape[0]),
                   np.linspace(-1,1,field[0].shape[1]))
ax.plot_surface(X, Y, field[0], cmap="bone", linewidth=0, antialiased=False)
plt.title('Fractional Gaussian Field')
plt.show()

```



Fractional Gaussian Field



## 1.2 Conclusion

Ce projet avait pour objectif de donner les **bases des bruits pseudo-aléatoires**. Non mathématiquement mais par divers applications. En présentant les principaux **bruits**, leurs **générations**, des **opérations** typiques et **élémentaires** ainsi que des **exemples** concrets avec au final un large éventail d'opérations et d'idées disponibles pour un artiste novice dans la manipulation des bruits. L'idée, comme répétée plusieurs fois depuis le début, est de **combiner diverses bruits et opérations afin de former le résultat souhaité**. Au final le projet avait aussi pour but de mettre en évidence la présence de ces bruits dans le monde audio-visuel. C'est à dire que quand vous regardez des films avec des effets spéciaux, de la génération(SciFi par exemple) ou lorsque vous jouez à des jeux-vidéos vous aurez de grandes chances d'avoir un bruit de Perlin + ou - transformé devant vos yeux.

La note qui peut-être bon de rajouter est que durant tous ce projet nous avons généré les bruits comme des matrices, c'est à dire que l'algorithme génère une matrice. Mais dans la plupart des cas on génère la valeur du bruit que pour un seul point. C'est à dire que notre algorithme est la fonction  $n(x,y)$  par exemple. En effet faire cela permet de plus facilement le manipuler. Exemple lors du domain warping quand on avait besoin de générer une plus grande matrice que celle que l'on veut transformer si on veut pouvoir avoir une transformation dépendant de  $x$  et  $y$ . À noter aussi que si on souhaite générer un bruit identique lors de la 2 ème itération il faut coder un système de seed(graine). C'est à dire une liste de nombres aléatoires = permutations pouvant être regénérées via un nombre "seed" que l'on donne. Et ceux sont ces permutations qui vont donner les gradients pour le bruit de Perlin par exemple. Faire ce système est très important car un des objectifs secondaires des bruits est que ça n'a pas besoin d'être sauvegardé sur le disque. Par exemple afin de générer un univers entier et que lorsqu'on se déplace les planètes sont identiques. (Donc on regénère à chaque fois la portion de l'espace où le joueur se situe).

**Merci d'avoir porter attention à ce projet.**

## 1.3 Références

- <https://thebookofshaders.com/>
- <https://adriamb.io/2014/08/09/perlinnoise.html>
- [https://en.wikipedia.org/wiki/Perlin\\_noise](https://en.wikipedia.org/wiki/Perlin_noise)
- [https://en.wikipedia.org/wiki/Simplex\\_noise](https://en.wikipedia.org/wiki/Simplex_noise)
- <https://catlikecoding.com/unity/tutorials/pseudorandom-noise/noise-variants/>
- <https://hal.inria.fr/inria-00402079/document> (*pour la simulation d'érosion ratée*)
- <https://threejs.org/> (*Moteur 3D pour JavaScript(Rendus des planètes)*)

### Aller plus loin:

- <https://hal.archives-ouvertes.fr/hal-00695670/document> (*Gabor noise*)
- <https://www.cs.ubc.ca/~rbridson/docs/bridson-siggraph2007-curlnoise.pdf> (*Incompressible noise*)
- <http://paulbourke.net/fractals/randomwalk/> (*Autre méthode pour générer un bruit*)
- [https://en.wikipedia.org/wiki/Fractional\\_Brownian\\_motion](https://en.wikipedia.org/wiki/Fractional_Brownian_motion) (*La théorie mathématique*)