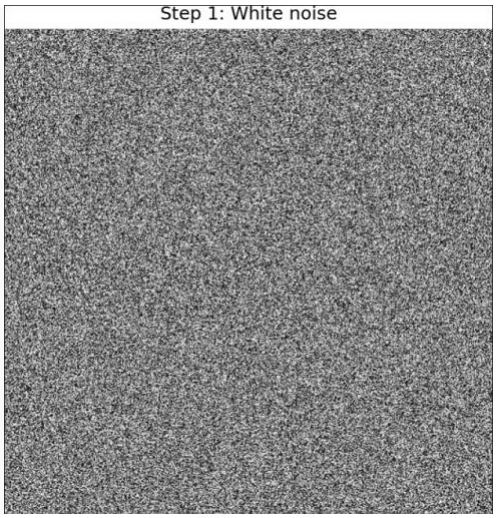
2025-10-18

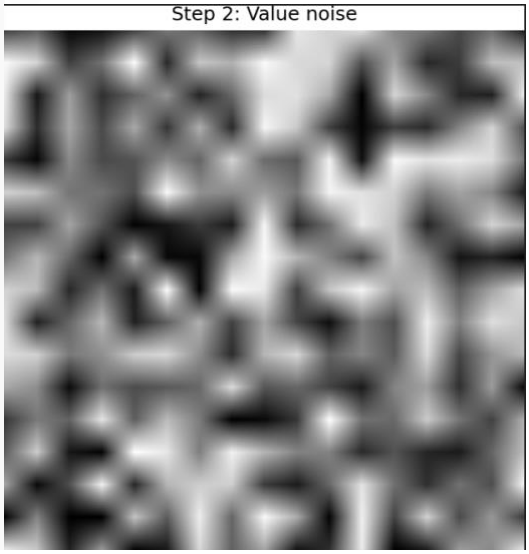
Step1:

```python
import numpy as np
import matplotlib.pyplot as plt
import os

H, W = 512, 512
seed = 123
rng = np.random.RandomState(seed)

def save_img(path, img, title=None, cmap='gray'):
    plt.figure(figsize=(5,5))
    plt.imshow(img, cmap=cmap)
    if title: plt.title(title)
    plt.axis('off'); plt.tight_layout()
    plt.savefig(path, dpi=150, bbox_inches='tight', pad_inches=0)
    plt.close()

# ----------------------------
# STEP 1 — WHITE NOISE
# ----------------------------
white = rng.rand(H, W)
save_img("step1_white.png", white, "Step 1: White noise")
```



Step2:

```python
# ----------------------------
# STEP 2 — VALUE NOISE (coarse grid + bilinear)
#   Random VALUES at lattice corners, then bilinear interpolation.
# ----------------------------
cell = 32                           # larger cell => bigger blobs
Gh, Gw = H // cell + 2, W // cell + 2
grid_vals = rng.rand(Gh, Gw)

# Coordinates in "grid space"
y = np.linspace(0, Gh-2, H)
x = np.linspace(0, Gw-2, W)
yy, xx = np.meshgrid(y, x, indexing='ij')

xi = np.floor(xx).astype(int)
yi = np.floor(yy).astype(int)
xf = xx - xi
yf = yy - yi

# Bilinear interpolation of corner VALUES
v00 = grid_vals[yi,     xi    ]
v10 = grid_vals[yi,     xi + 1]
v01 = grid_vals[yi + 1, xi    ]
v11 = grid_vals[yi + 1, xi + 1]

val_x1 = v00*(1-xf) + v10*xf
val_x2 = v01*(1-xf) + v11*xf
value_noise = val_x1*(1-yf) + val_x2*yf

save_img("step2_value_noise.png", value_noise, "Step 2: Value noise")
```



Ok, so the big idea is:

1) Keep pixel values still random, but only at the corner of 'bigger blobs'.
2) For pixels within each blob, pixel values are going to be chosen through linear interpolation, in other words, instead of being random, they would be predictable, hence revealing the hint of 'structure' if we see them.
3) So the question is, how to replace the 512x512 pixel value with :
   - Random values at the corner
   - Linear interpolation in between?
   - I used to do coding in C, VBA, Fortran, and I can tell you, you have to control the indexing. Could be annoying. So how does Python achieves the goal here without too much 'do loop'?

If I understand correctly, we have 512*512 pixels, now we divide it into 32 blocks, each block will have a different pixel number, but within the block, the pixel values will be the same. (That's how I defer from the pictures drawn). 512/32 will tell us how many pixels each block have, roughly 16 pixels. Hmm, but then it does not make sense to have grid_vals to be a range of random numbers of (16*16). So what is cell=32 mean here? The number of pixels each way for the block? Because then, 512/32=16 is the number of big blocks, 16 each way, so 16*16 blocks in total, and we need one random number for each. Because only this way it makes sense the size of grid_vals. Let's proceed.

Once for each block the pixel value is the same, now we need to create some structures: varying difference according to bilinear rules? What is 'coordinate in grid space'? let's skip the coordinates part and check the bilinear interpolation. So V00, V10, V01, V11 seems to be counter clockwise. They are extracting the random pixel value from each block. So this confirms our thinking that the size of grid_vals is the size of big blocks: 16*16 chunks.

Val_x1, val_x2 is telling us how to calculate pixels within each block based on a linear model. It uses the values at four corners and change them linearly according to the distance from the corner looks like. Remember the four corner values belong to 4 blocks, hence, 4 different random pixel values. But remember val_x1, val_x2 is for 'all the pixels within each block', an old programming language would have required you to code each one using a for loop, after all, there are 32 pixels along the row and column direction, so how come the 'loop' is avoided here? I heard about the 'vectorization' in machine learning course, so this is it. But they key is, is there a way for the computer to know that you are referring to a vector, rather than a single value?

Because the size of the grid_vals is 16*16 since we have 16*16 big blocks, now calling grid_vals [yi, xi] will give us block (i,i). Remember in old programming language, you can't leave i as an undetermined variable, you have to write loop, specify i to be from 1 to 16, then within each cycle it actualizes i into a number. But here in vector-ized language, you leave it un specified, and Python knows you want all the (i,j) combos? This way it knows v00, v10 will be a vector of pairs, you don't need to specify its dimension!

Then what are the codes of coordinates in grid space doing? Ok, looks like they are kind of specifying the indices for handling grid_vals? Are they real coordinates, or they are merely integer values specifying the ith row jth column of pixel blocks?