1 Natural image statistics

```
import numpy as np
import numpy.linalg as linalg
import matplotlib.pyplot as plt
from munkres import Munkres
from scipy import stats
from PIL import Image, ImageFilter, ImageOps
from src.utilities import *
from src.intensity_entropy import *
from src.kernels import *
plt.rcParams['image.cmap'] = 'gray'
plt.rcParams['figure.figsize'] = (12.8, 9.6)
```

1.1 The usual histograms

```
img = ImageOps.grayscale(Image.open('canyon.jpg'))
scale = max(np.shape(img))
data = np.array(img)
img
```

```
def contrast_renormalize(x):
    mid = np.array(np.shape(x)) // 2
    std = np.std(x)
    return (x.item(*mid) - np.mean(x)) / std if std > 0 else 0
```

```
def log_contrast(x):
    y = np.vectorize(lambda a: np.log(a) if a > 0 else 0)(x)
    return y - np.mean(y)
```

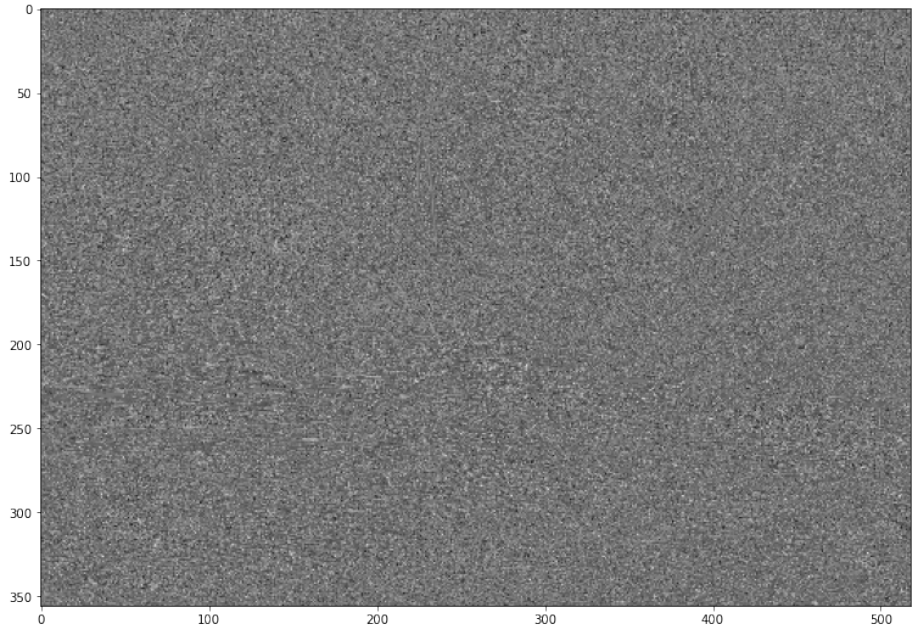
```
np.array(np.shape(img)) // 5
```

```
array([356, 518])
```

```
def renorm_blocks(n, f=contrast_renormalize):
    return lambda x: mapblocks(*(np.array(np.shape(x)) // n), f, np.array(x))
```

```
def iterate(f, n):
    return (lambda x: iterate(f, n-1)(f(x)) if n > 0 else x)
```

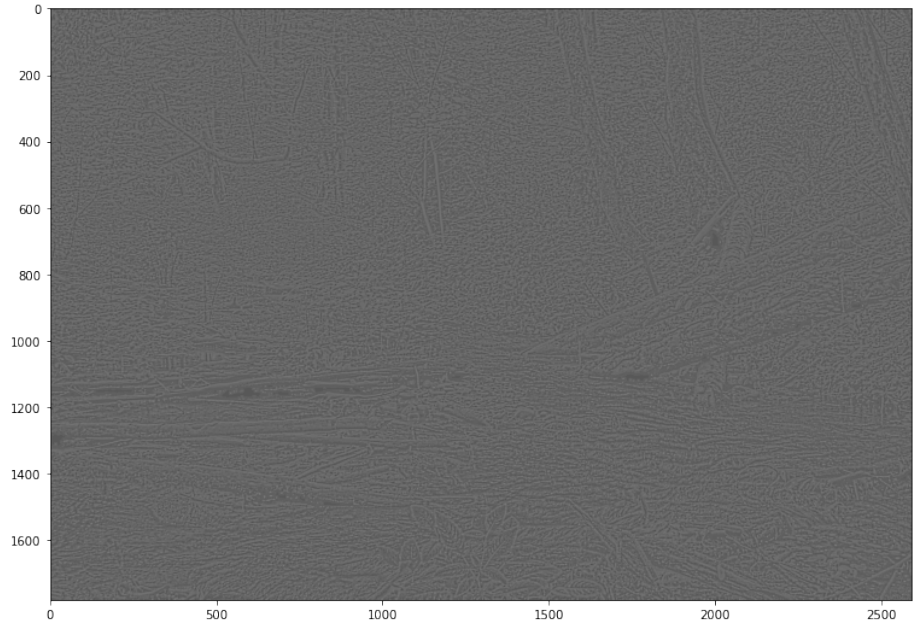
```
rdata = iterate(renorm_blocks(5), 1)(img)
plt.imshow(rdata);
```




```
ldata = log_contrast(1.0*data)
plt.imshow(ldata);
```


```
l1data = iterate(lambda x: mapbox(5, contrast_renormalize, x), 1)(ldata)
l2data = iterate(lambda x: mapbox(5, contrast_renormalize, x), 2)(ldata)
```

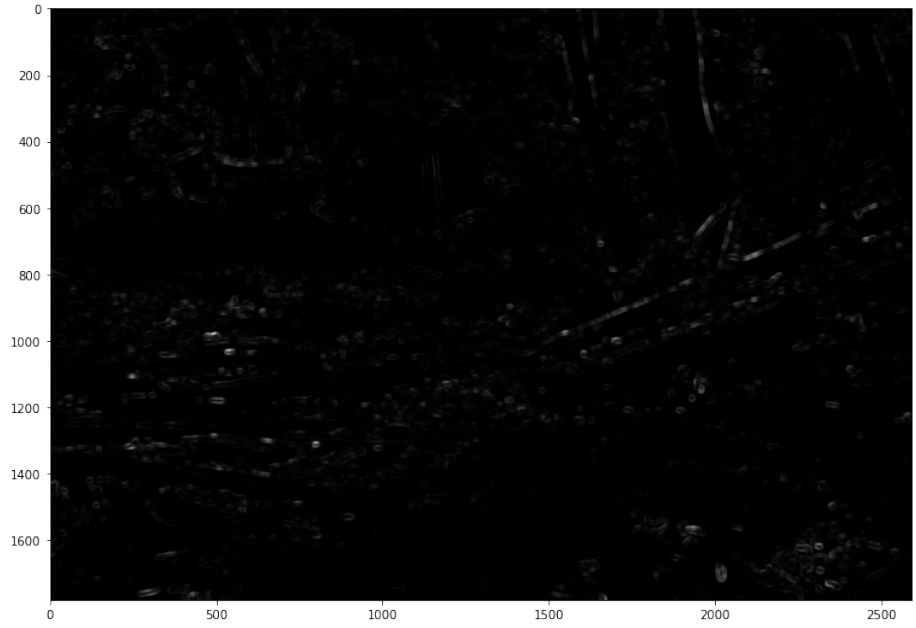
```
plt.imshow(l1data);
```



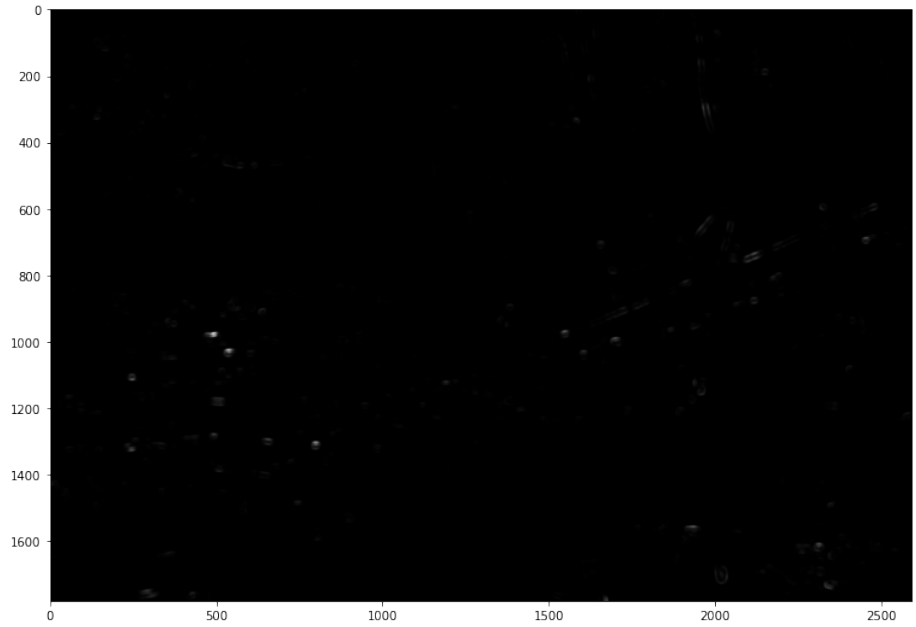

```
v1data = iterate(lambda x: mapbox(5, np.var, x), 1)(data)
v2data = iterate(lambda x: mapbox(5, np.var, x), 1)(v1data)
```

```
v3data = iterate(lambda x: mapbox(5, np.var, x), 1)(v2data)
plt.imshow(v1data);
```


```
plt.imshow(v2data);
```



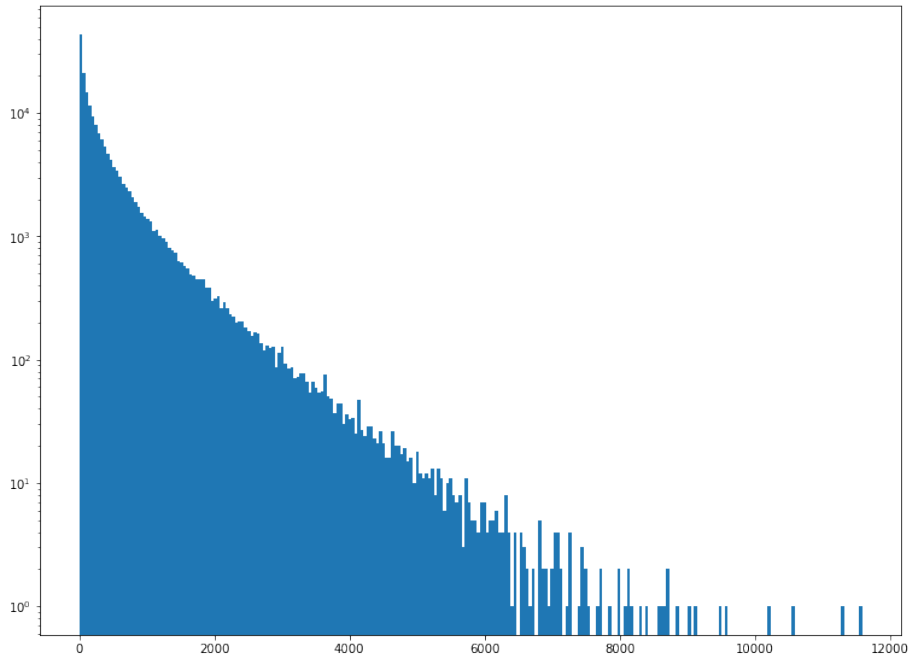
```
plt.imshow(v3data);
```



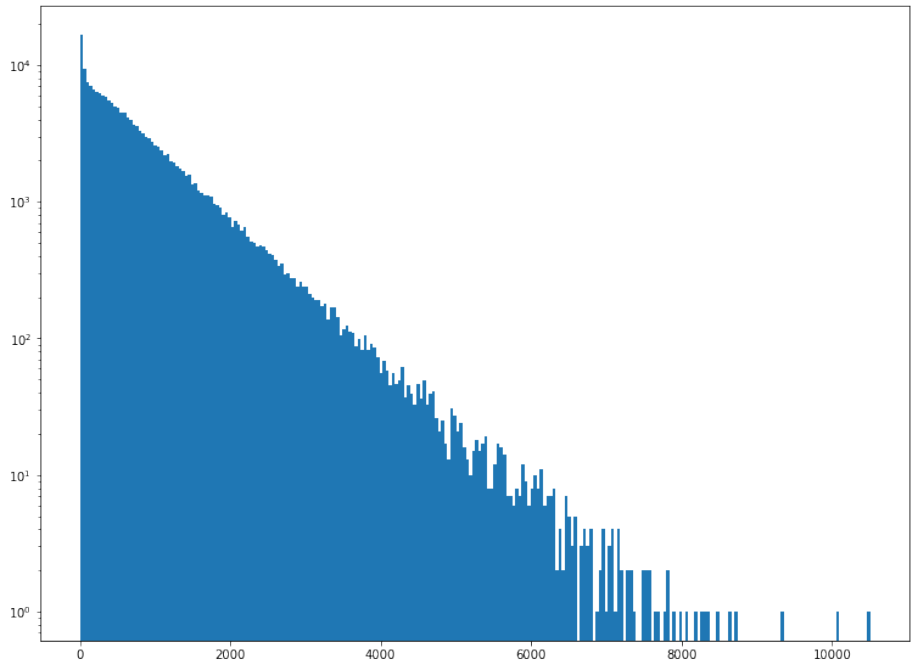

```
sep_vblocks = renorm_blocks(5, np.var)(data)
plt.imshow(sep_vblocks);
```

```
mean_vblocks = renorm_blocks(5, np.mean)(v1data)
plt.imshow(mean_vblocks);
```

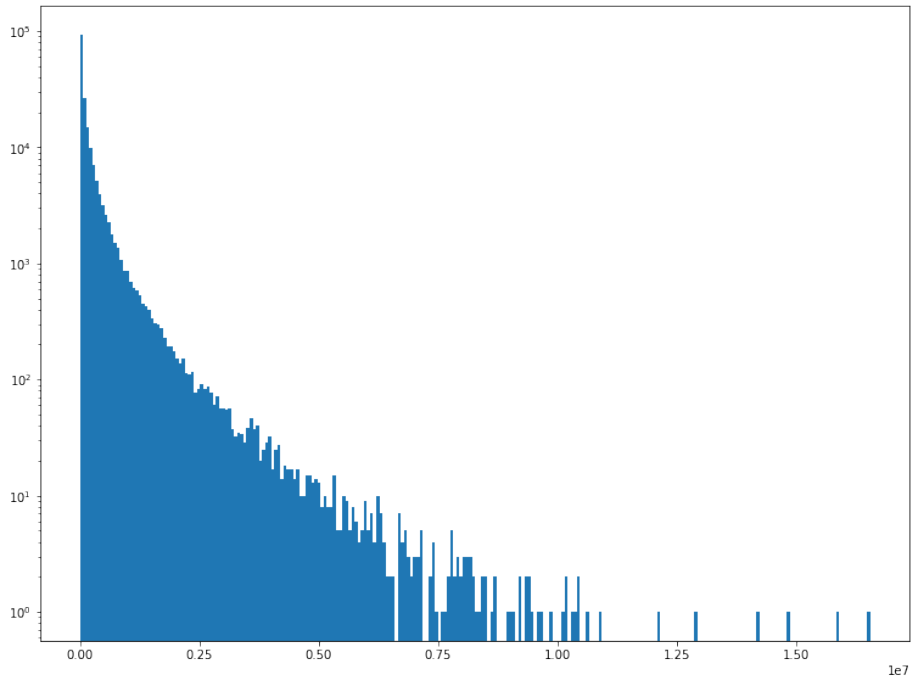

```
plt.hist(sep_vblocks.flat, 256)
plt.yscale('log');
```




```
plt.hist(mean_vblocks.flat, 256)
plt.yscale('log');
```



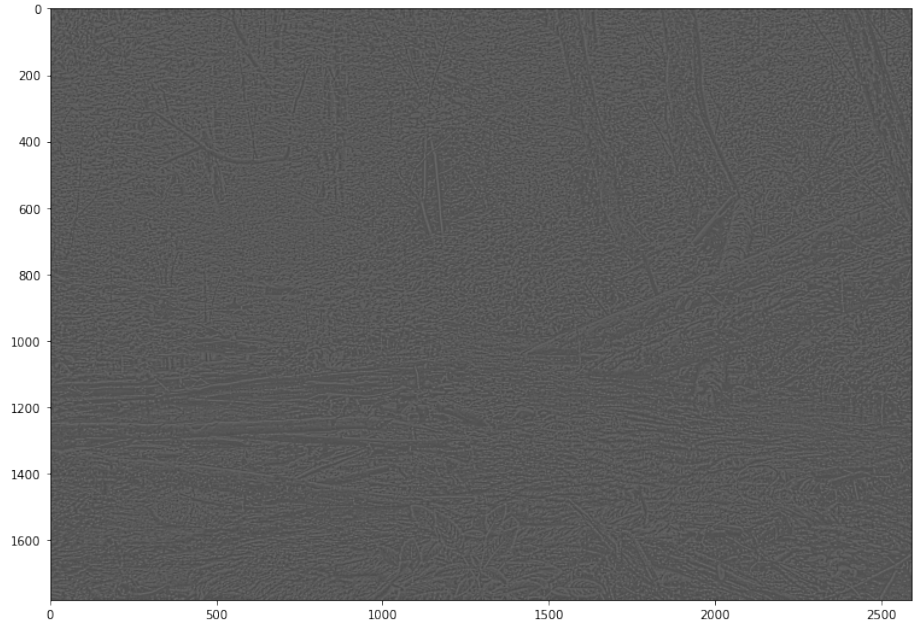
```
mean_v2blocks = renorm_blocks(5, np.mean)(v2data)
plt.hist(mean_v2blocks.flat, 256)
plt.yscale('log');
```



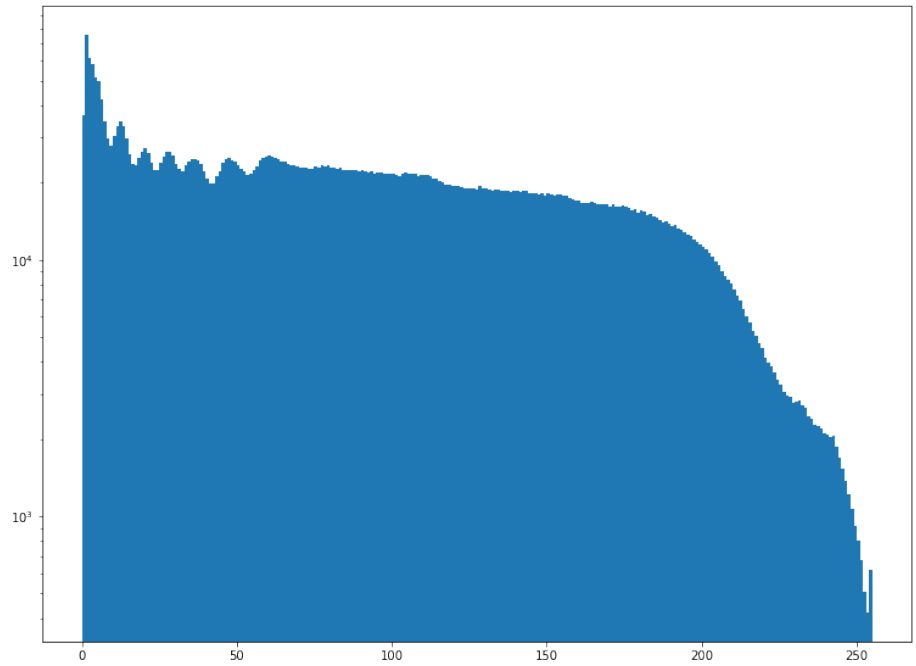

```
vdata = iterate(lambda x: mapbox(5, np.var, x), 1)(ldata)
plt.imshow(vdata);
```


```
g1data = iterate(lambda x: mapbox(5, contrast_renormalize, x), 1)(np.array(img))
g2data = iterate(lambda x: mapbox(5, contrast_renormalize, x), 2)(np.array(img))
```

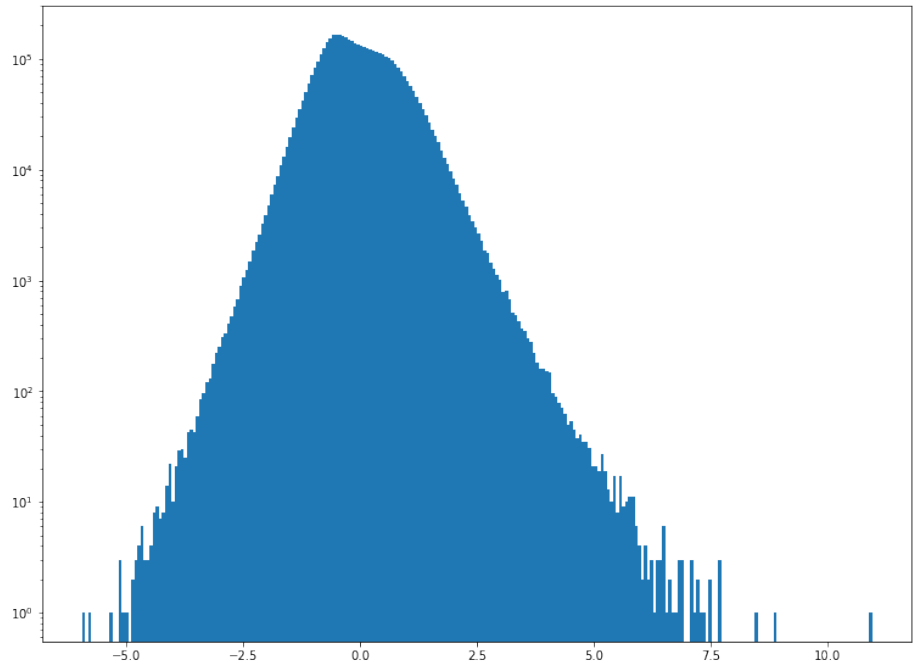
```
plt.imshow(g1data);
```



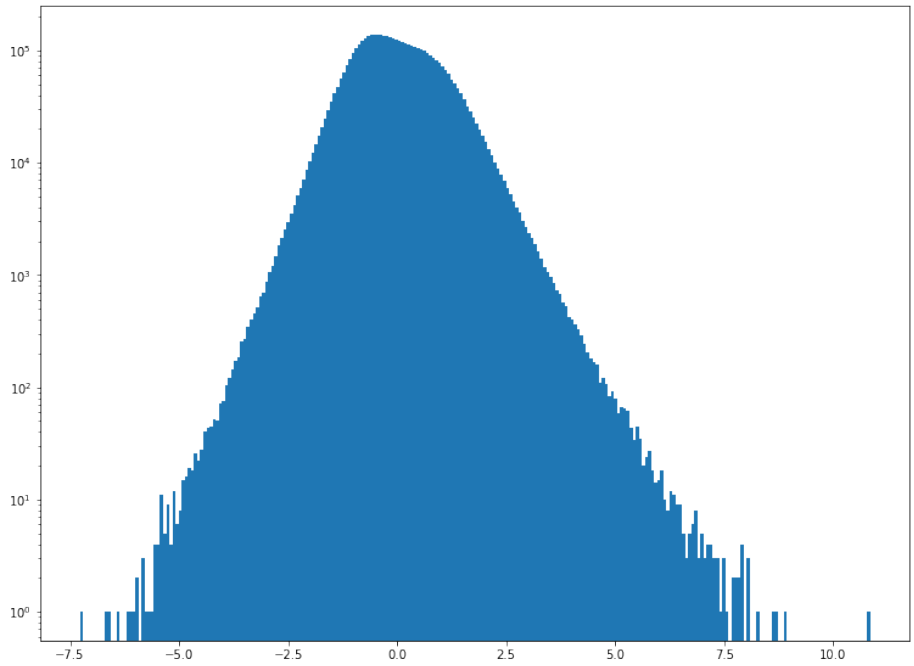

```
plt.hist(np.array(img).flat, range(256))
plt.yscale('log');
```



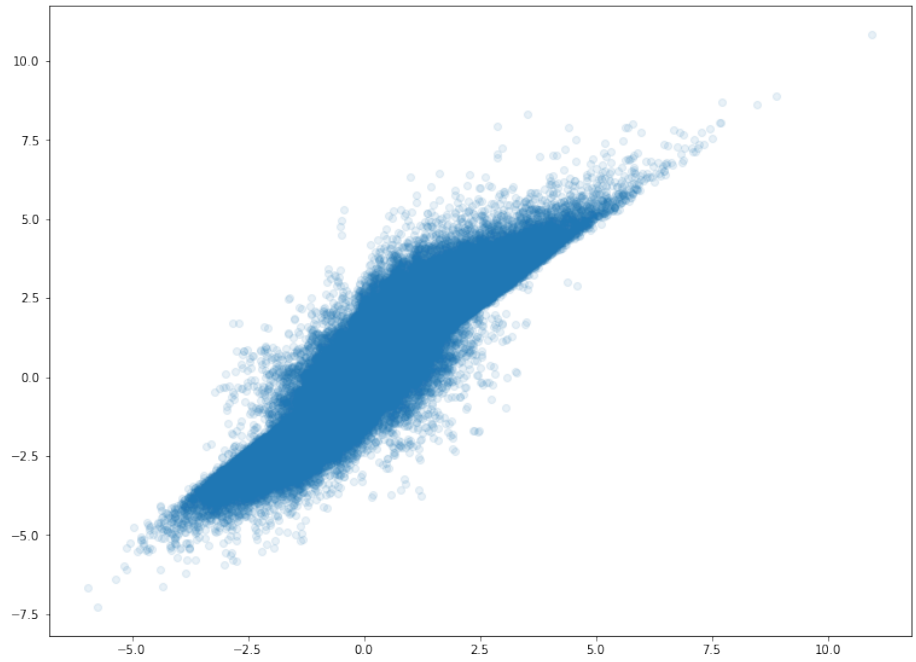

```
plt.hist(g1data.flat, 256)
plt.yscale('log');
```



```
plt.hist(g2data.flat, 256)
plt.yscale('log');
```

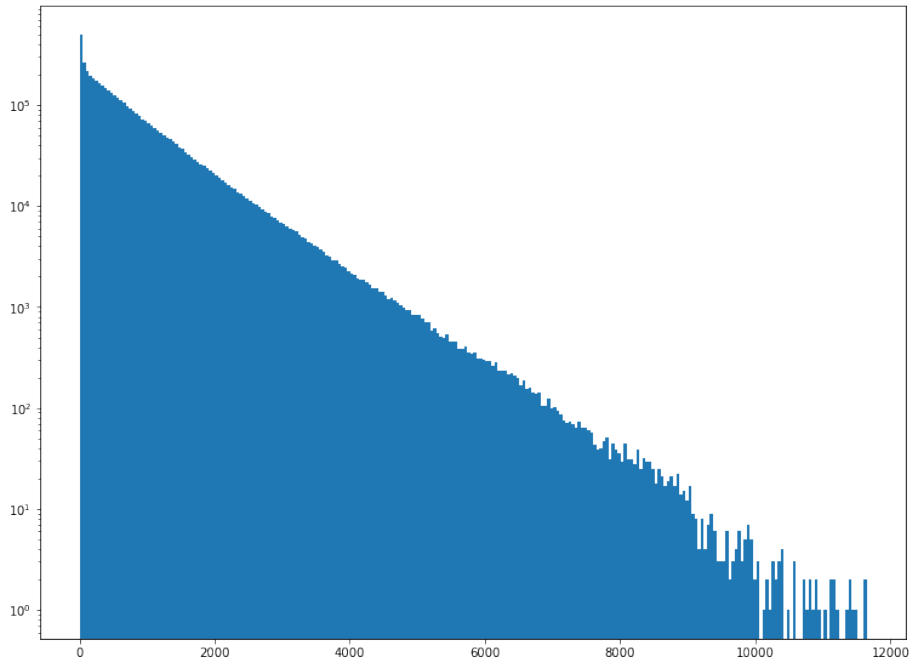



```
plt.scatter(g1data.flat, g2data.flat, alpha=0.1);
```

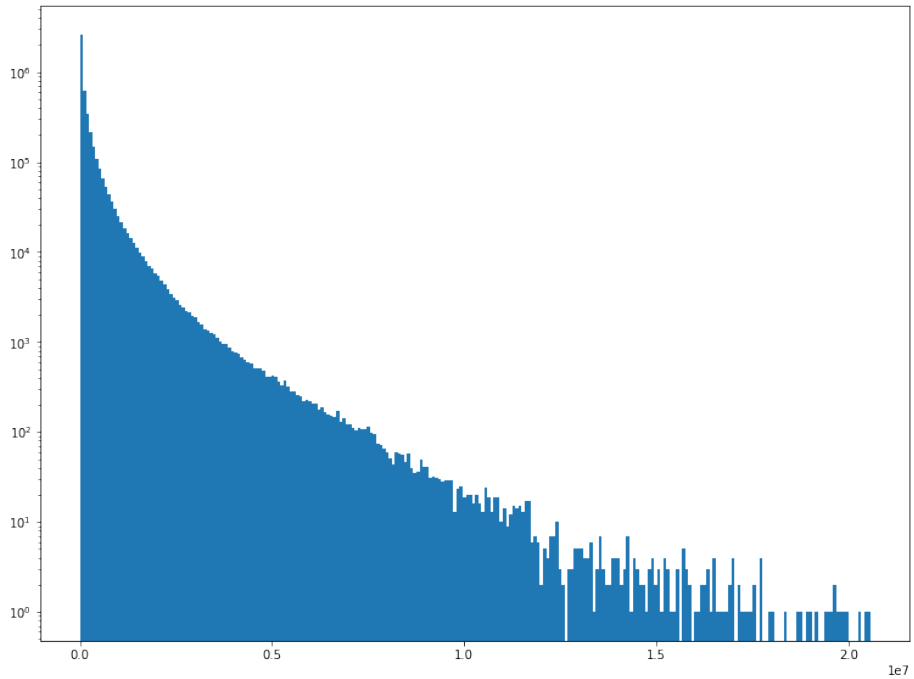



```
plt.imshow(vdata);
```

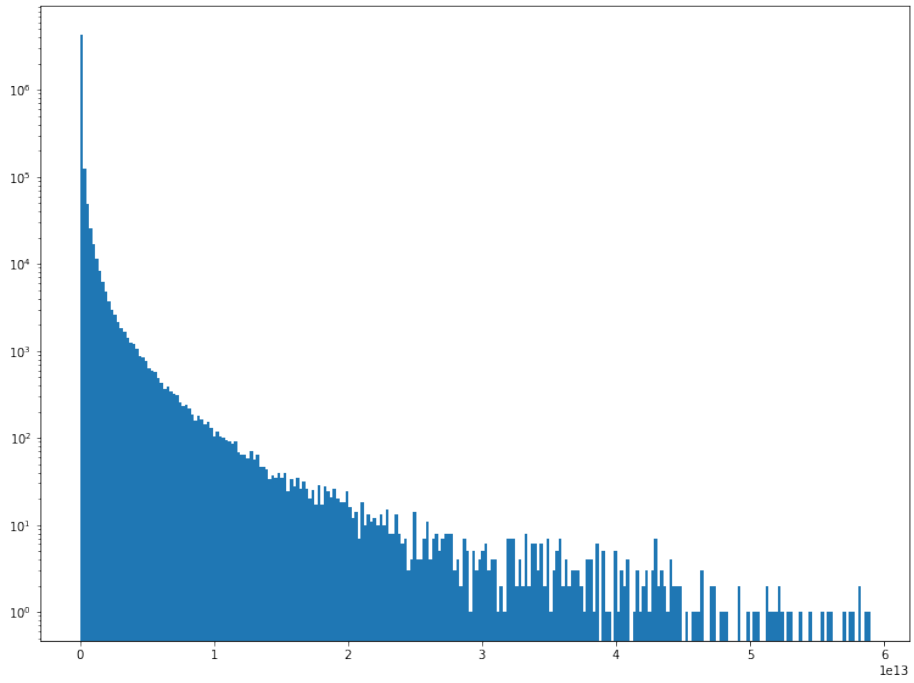

```
plt.hist(v1data.flat, 256)
plt.yscale('log');
```



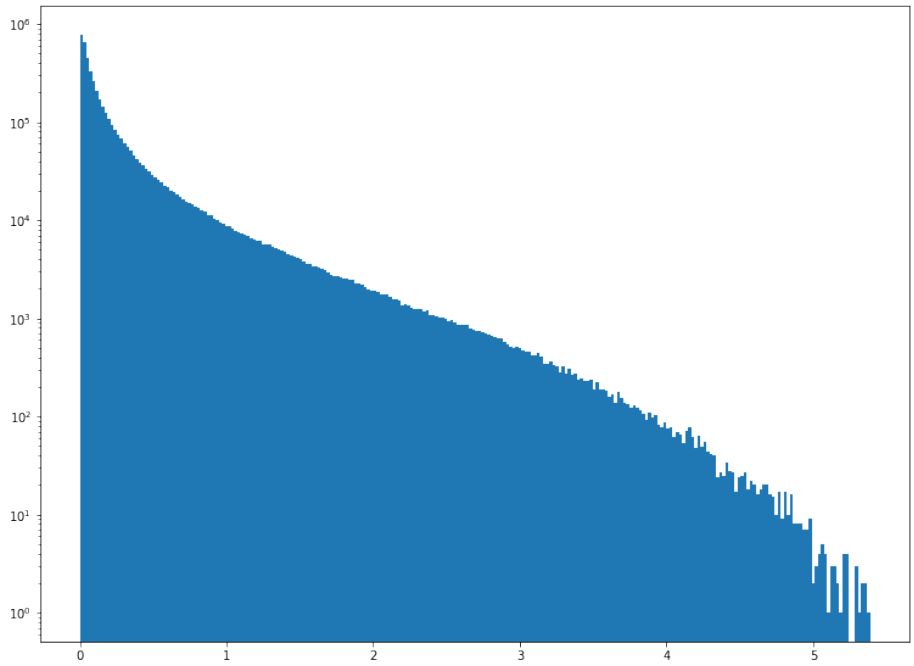

```
plt.hist(v2data.flat, 256)
plt.yscale('log');
```



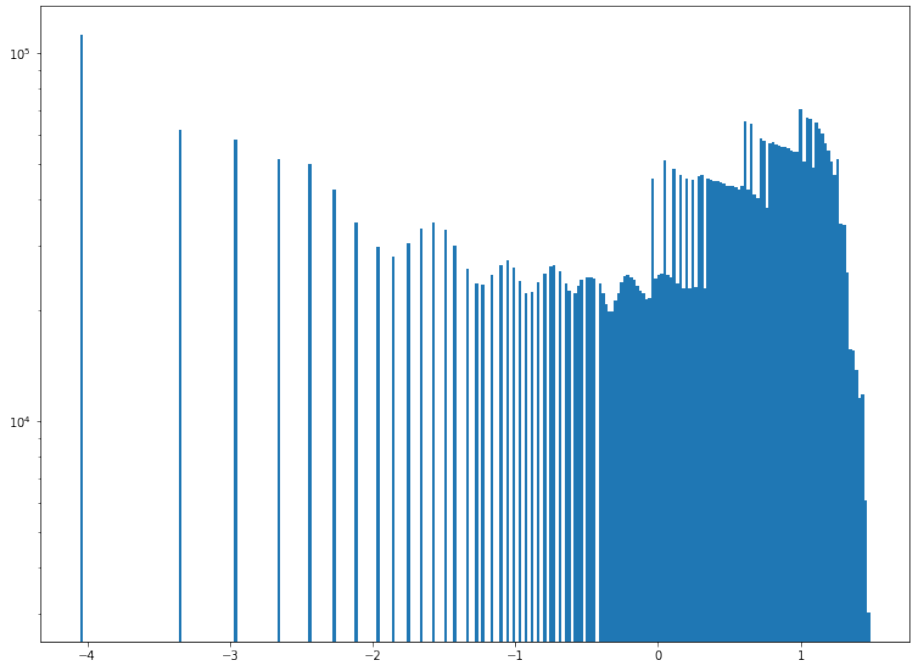
```
plt.hist(v3data.flat, 256)
plt.yscale('log');
```



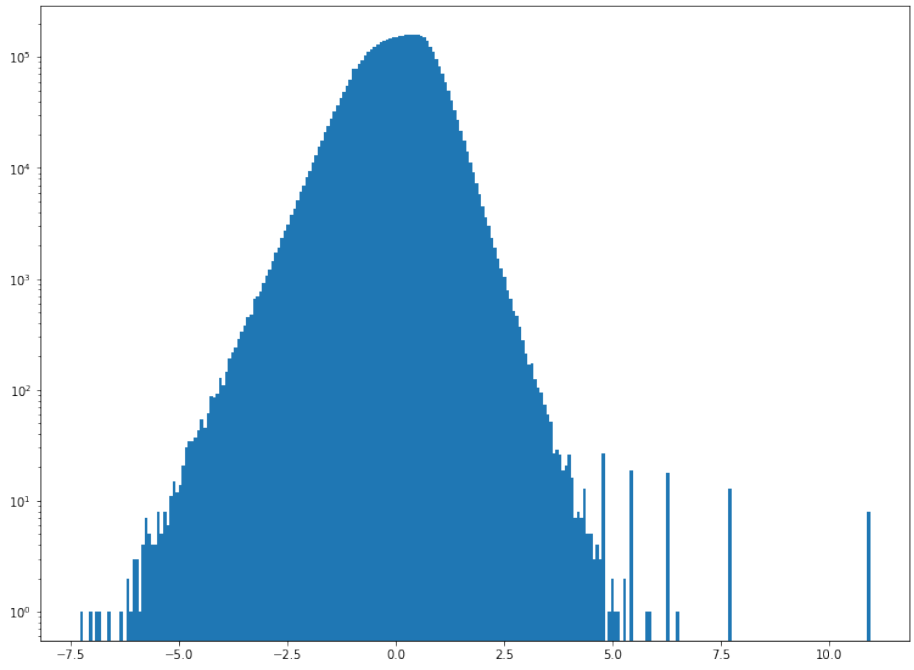

```
plt.hist(vdata.flat, 256)
plt.yscale('log');
```



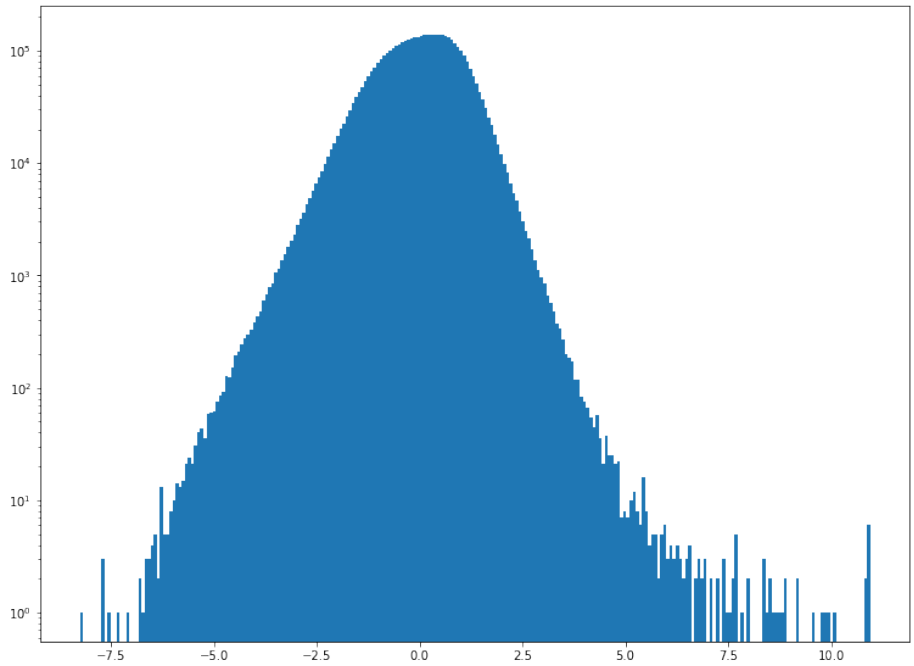

```
plt.hist(ldata.flat, 256)
plt.yscale('log');
```



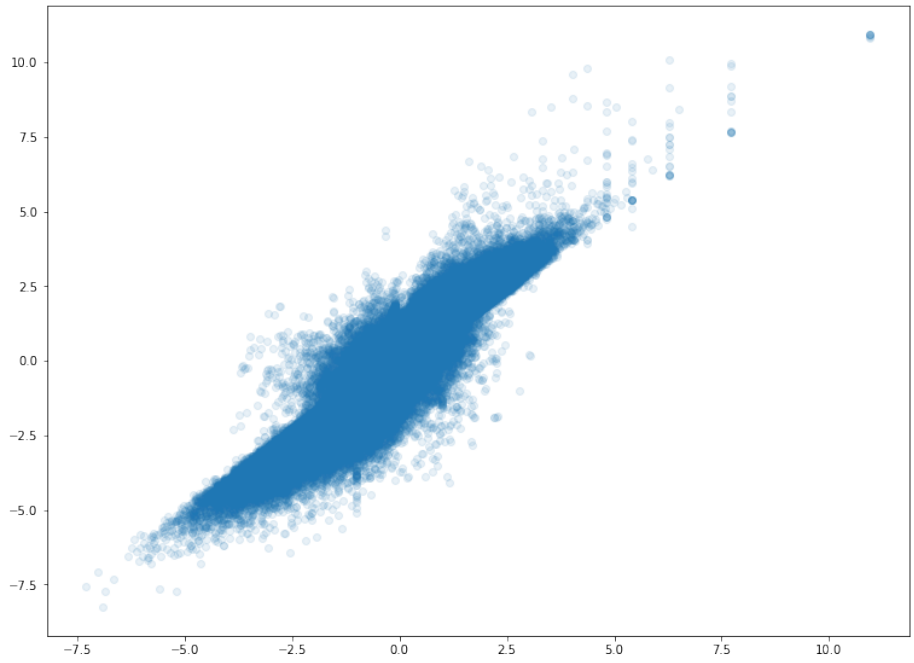
```
plt.hist(l1data.flat, 256)
plt.yscale('log');
```




```
plt.hist(l2data.flat, 256)
plt.yscale('log');
```




```
plt.scatter(l1data.flat, l2data.flat, alpha=0.1);
```



1.2 Fractal textures

```
x = 255 * np.random.rand(2, 2)
```

```
rep2 = lambda x: np.reshape(np.repeat([x], 4), (2, 2))
```

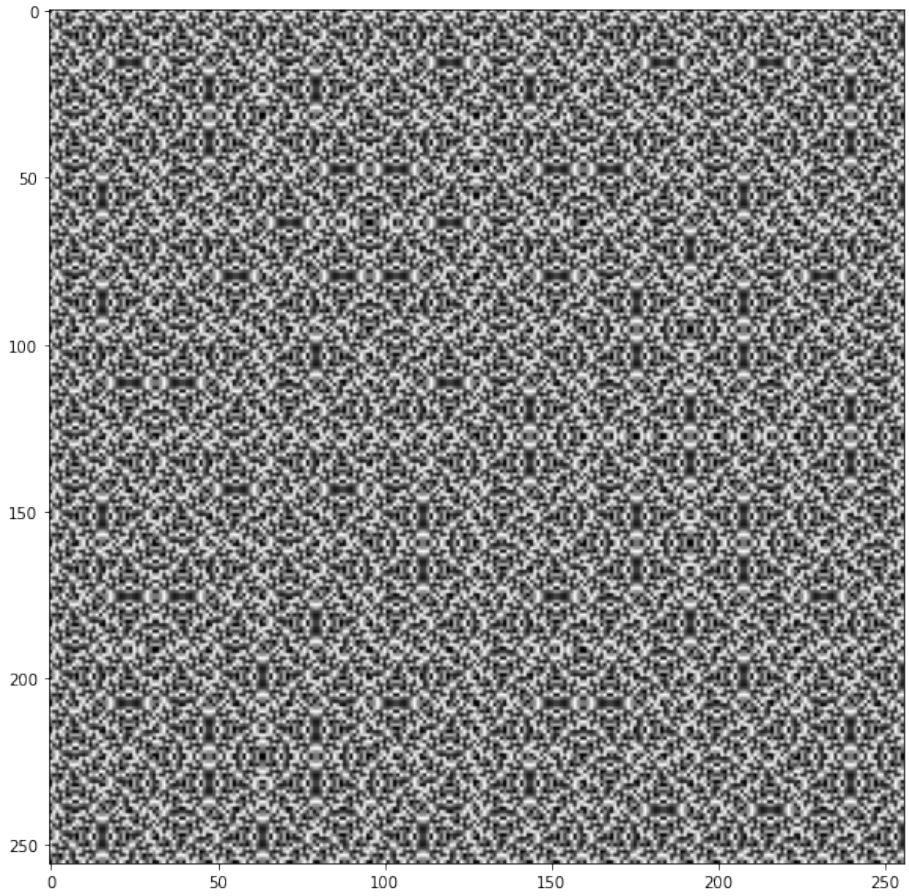
```
np.block([[rep2(b) for b in a] for a in x])
```

```
array([[ 89.75068064,  89.75068064, 203.5007879 , 203.5007879 ],
       [ 89.75068064,  89.75068064, 203.5007879 , 203.5007879 ],
       [131.58072491, 131.58072491,  76.29484712,  76.29484712],
       [131.58072491, 131.58072491,  76.29484712,  76.29484712]])
```

```
def grow(f):
    return lambda x: np.block([[f(b) for b in a] for a in x])
```

```
def agrow(f):
    return lambda x: np.block(f(x))
```

```
def fblock(x):
    return [[x, np.transpose(x)], [np.transpose(x), np.rot90(x)]]

plt.imshow(iterate(agrow(fblock), 5)(np.random.rand(8, 8)));
```



```
from PIL import Image
```

```
img = Image.fromarray((255*np.random.rand(2, 2)).astype('uint8'))
```

```
Tinv = np.invert(np.array([
    [1, 0, 1],
    [0, 1, 1],
    [0, 0, 1]
]))
img.transform((2, 2), Image.AFFINE, data=Tinv.flatten()[:6], resample=Image.NEAREST)
```

▪

1.3 Probabalistic inverse neighborhood reductions

Given a list of values y_i for $1 \leqslant i \leqslant n$ and a function f on any size list of values, we want to determine a new list X_i of size m for each i so that $f(X_i) = y_i$ and so that all of the generated values $x \in X = \cup_i X_i$ are distributed according to a given $p(x)$. This cannot be done exactly, so we must choose whether to prefer correct y_i values or a correct distribution of X.

1.3.1 Distribution-focused algorithm

It is simplest to sample correctly distributed values and approximate the y_i.

Aside: What value should we add to a set so that it has a specified variance?

```
def var_next(var, x, ddof=0):
    n = len(x) + 1
    if n ≤ ddof:
        return np.nan
    if n == 1:
        return 0
    xmean = np.mean(x)
    xvarsum = (n - 1) * np.var(x, ddof=0)
    vardiff = (n-ddof) * var - xvarsum
    if vardiff > 0:
        side = -1 if np.random.random_sample() - 1/2 < 0 else 1
        return xmean + side * np.sqrt(vardiff * n / (n-1))
    else:
        return xmean
```

```
x = list(np.random.rand(10))
for _ in range(10):
    x.append(var_next(1, x, ddof=1))
np.var(x, ddof=1)
```

```
1.0
```

One way to invert a neighborhood reduction function is to sample n values and assign them to the bins in the best way possible. From considering the matrix of all errors in the y-values, we see that this greedy algorithm is a case of the assignment problem, which we may solve with the Kuhn-Munkres algorithm.

```
def inverse_nreduce(y, m, f, sampler):
    n = len(y)
    xs = -np.ones((n, m))
    for s in range(m):
        u = sampler(n)
        v = [[(y[i] - f(x + [x0]))**2 for x0 in u] for (i, x) in enumerate(xs)]
        assignments = Munkres().compute(v)
        for (i, j) in assignments:
            xs[i,s] = u[j]
    return xs
```

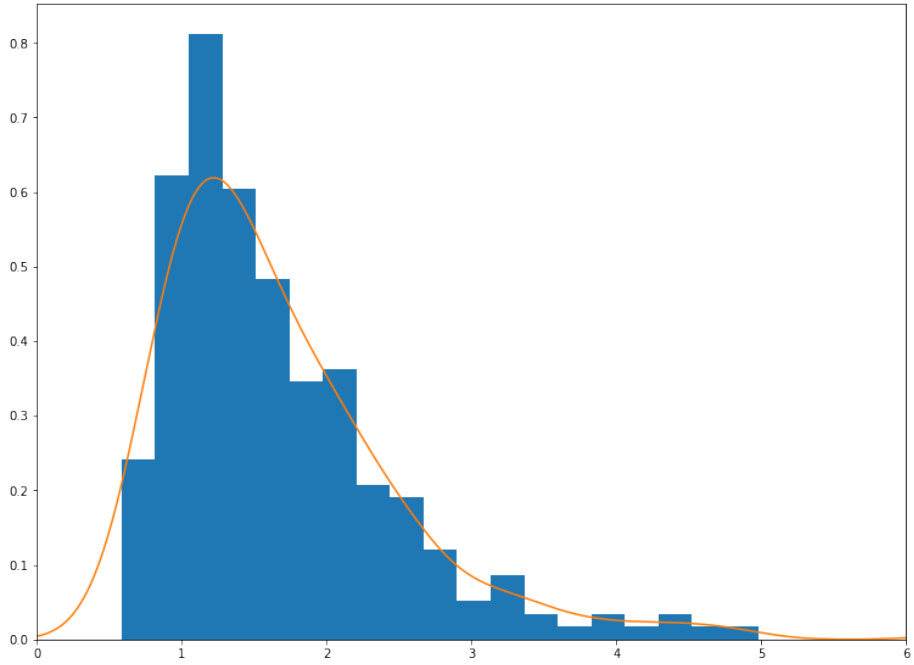
As an example, we sample from a Laplacian distribution with variance 2, but request sets with variance 1. The result is that the variances of the sets are imperfectly nudged away from 2 and towards 1.

```
inr = inverse_nreduce([1.0 for _ in range(250)], 25, np.var, lambda n: stats.laplace.rvs(size=n))
```

```
np.var(np.concatenate(inr))
```

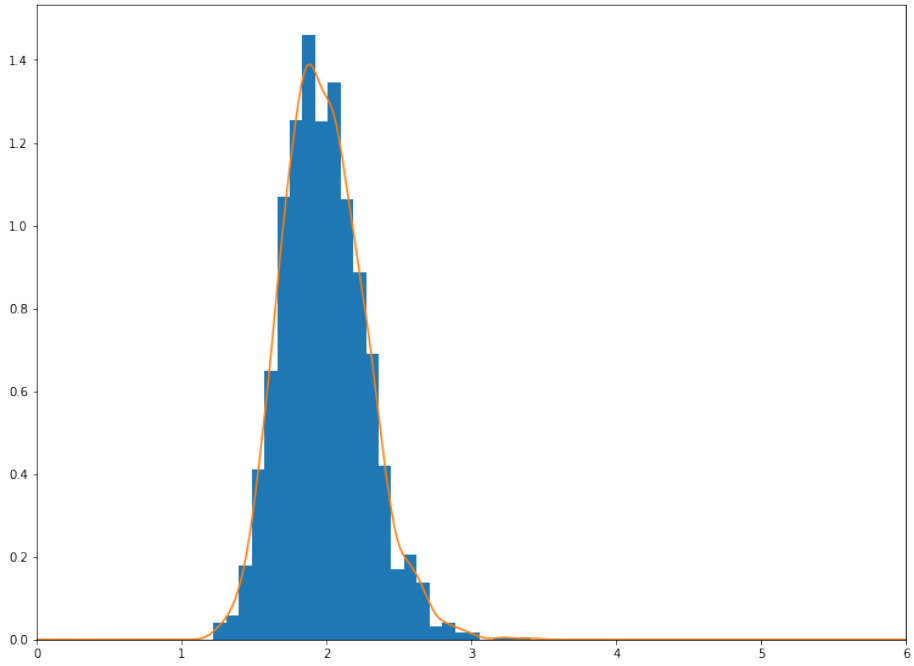
```
1.8887564926323521
```

```
nudged_vars = [np.var(r) for r in inr]
plt.hist(nudged_vars, 25, density=True)
plt.xlim(0, 6)
vs = np.linspace(0, 6, 300)
nudged_kde = stats.gaussian_kde(nudged_vars)
plt.plot(vs, nudged_kde(vs))
plt.show()
```



Compare the result of just sampling without redistributing values.

```
sample_vars = [np.var(stats.laplace.rvs(size=250)) for _ in range(2500)]
plt.hist(sample_vars, 25, density=True)
plt.xlim(0, 6)
vs = np.linspace(0, 6, 300)
sample_kde = stats.gaussian_kde(sample_vars)
plt.plot(vs, sample_kde(vs))
plt.show()
```

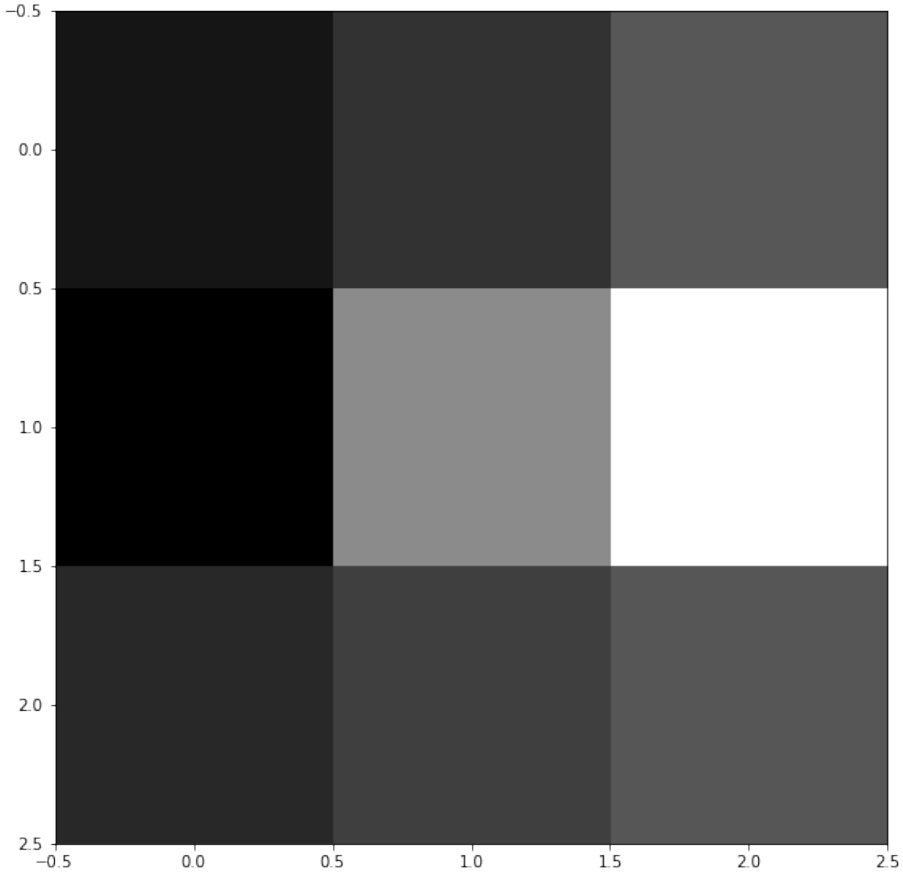


Now what are the distributions of the proposed natural scale-invariant variance images (as in Ruderman's statistics of natural images, doi: 10.1088/0954-898X_5_4_006)? First, we will try an exponential distribution of variances.

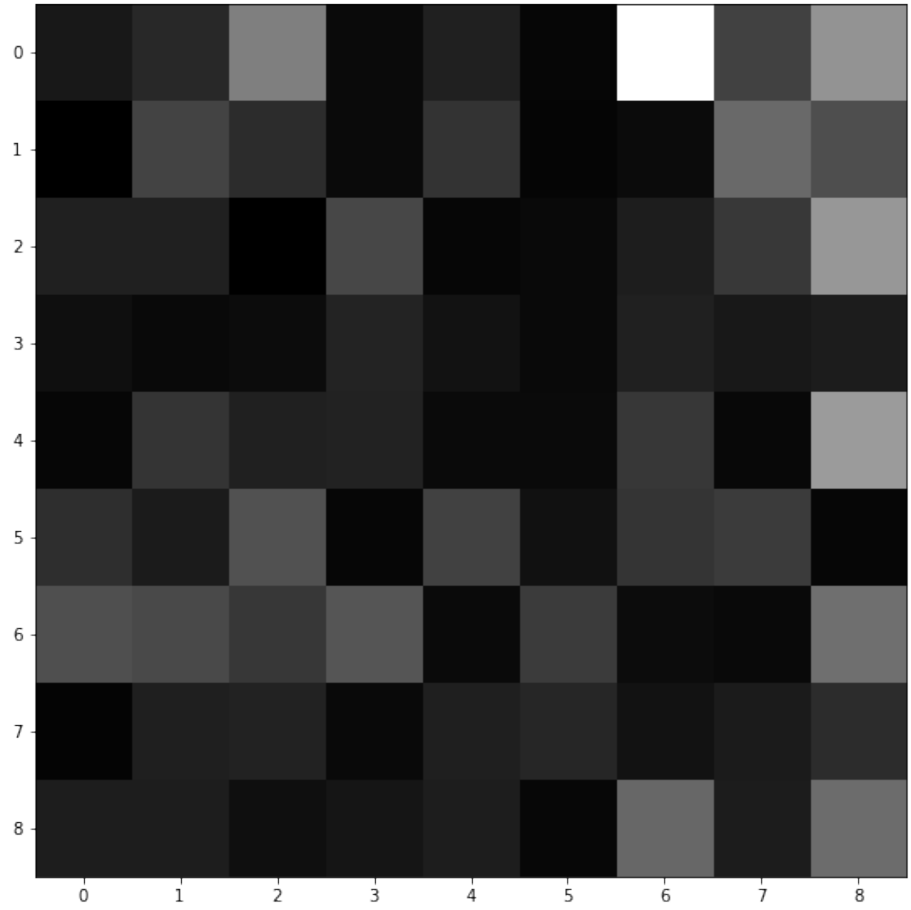
```
n0 = 3
m0 = 3
initial_vars = stats.expon.rvs(size=n0*n0)
inr_vars = inverse_nreduce(initial_vars, m0*m0, np.var, lambda n: (1 / 3) * stats.expon.rvs(size=n))
```

```
inr_mat = np.concatenate([np.concatenate(r, axis=1) for r in np.reshape(inr_vars, (n0, n0, m0, m0))],
↪  axis=0)
new_vars = inr_mat.flat
```

```
plt.imshow(np.reshape(initial_vars, (n0, n0)));
```

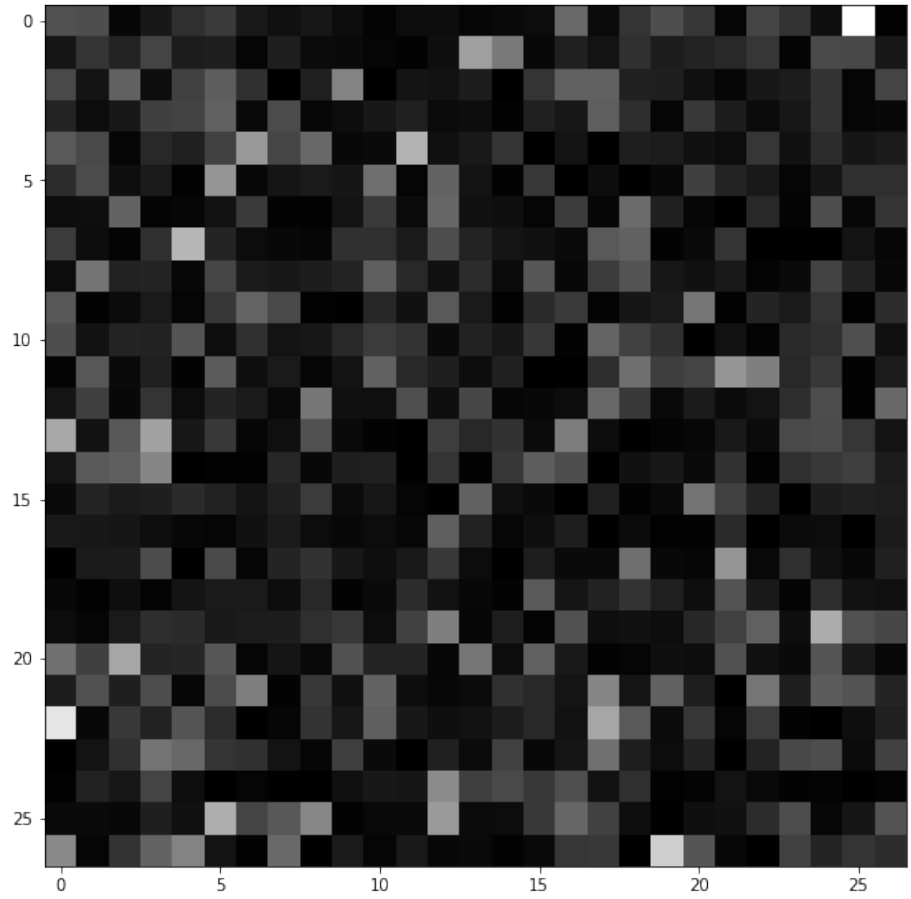



```
plt.imshow(inr_mat);
```



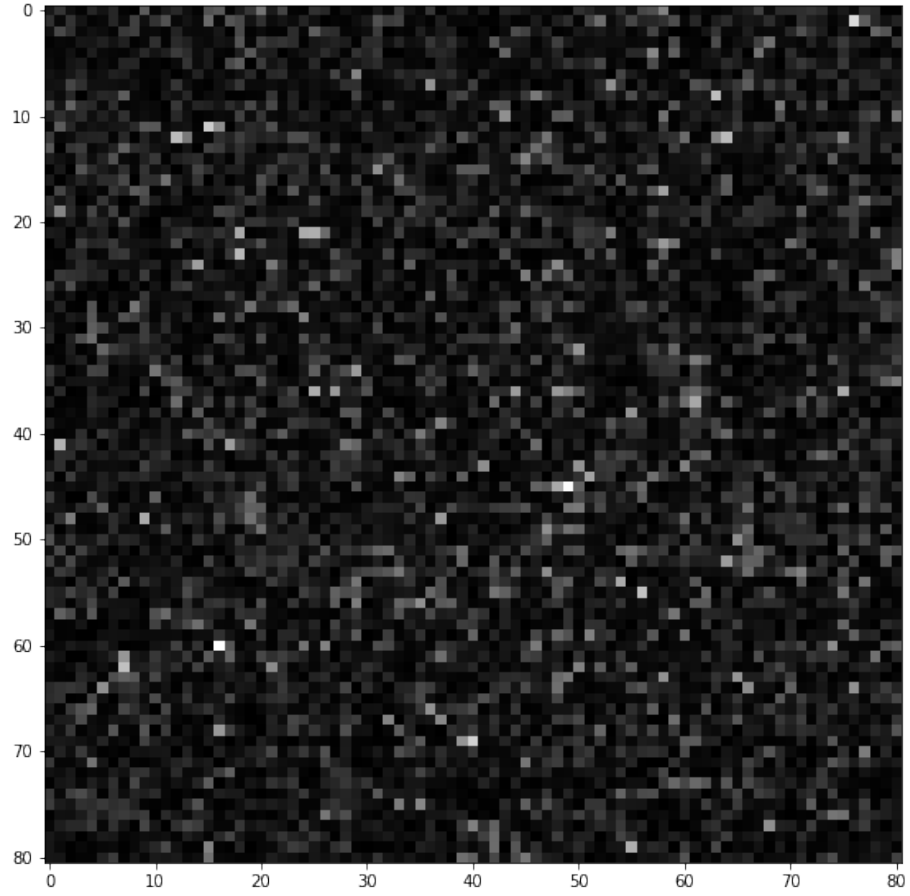

```
inr2_vars = inverse_nreduce(new_vars, m0*m0, np.var, lambda n: (1 / 3) * stats.expon.rvs(size=n))
n1 = int(np.sqrt(np.shape(inr2_vars)[0]))
inr2_mat = np.concatenate([np.concatenate(r, axis=1) for r in np.reshape(inr2_vars, (n1, n1, m0, m0))],
↪  axis=0)
new2_vars = inr2_mat.flat
```

```
plt.imshow(inr2_mat);
```



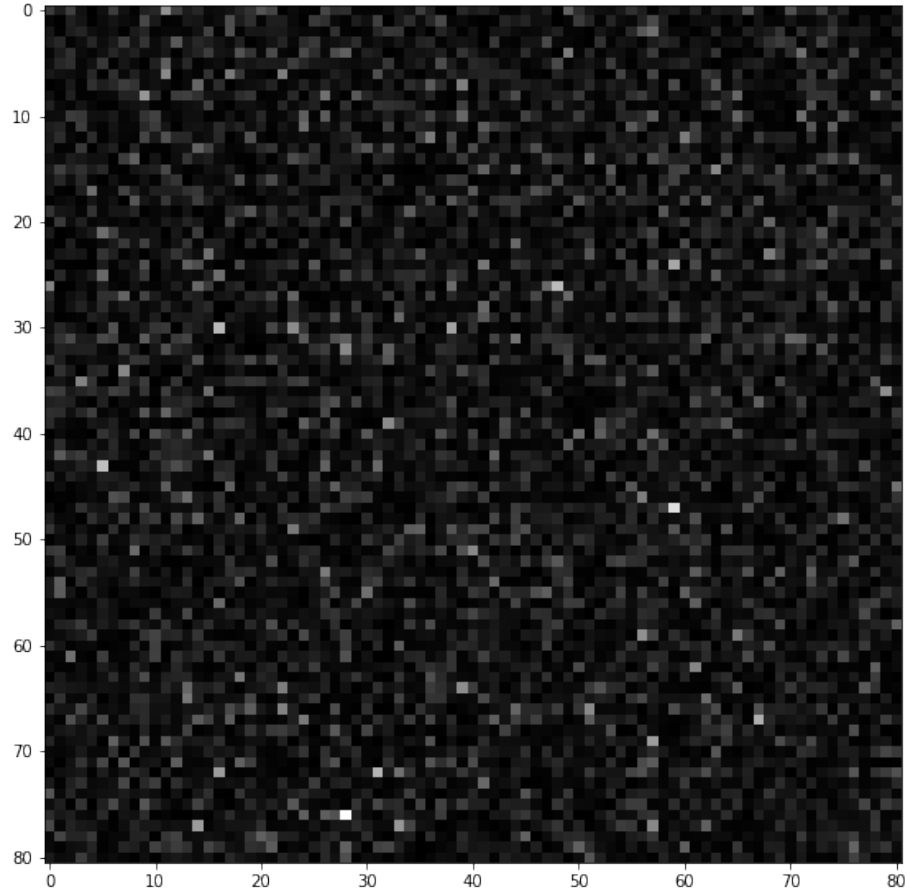

```
inr3_vars = inverse_nreduce(new2_vars, m0*m0, np.var, lambda n: (1 / 3) * stats.expon.rvs(size=n))
n2 = int(np.sqrt(np.shape(inr3_vars)[0]))
inr3_mat = np.concatenate([np.concatenate(r, axis=1) for r in np.reshape(inr3_vars, (n2, n2, m0, m0))],
↪  axis=0)
```

```
plt.imshow(inr3_mat);
```

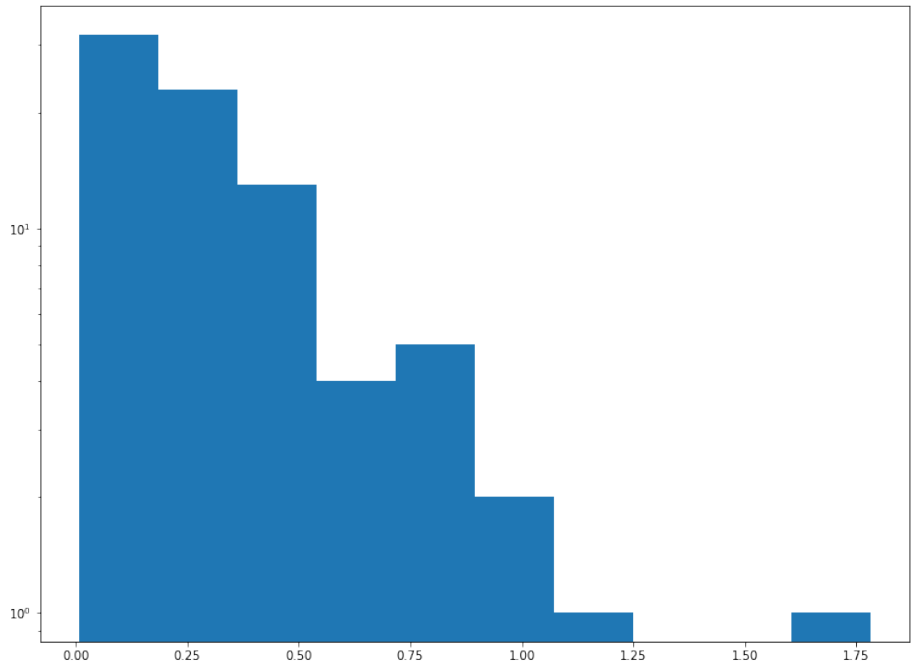



```
control_mat = np.reshape(stats.expon.rvs(size=np.size(inr3_mat)), np.shape(inr3_mat))
```

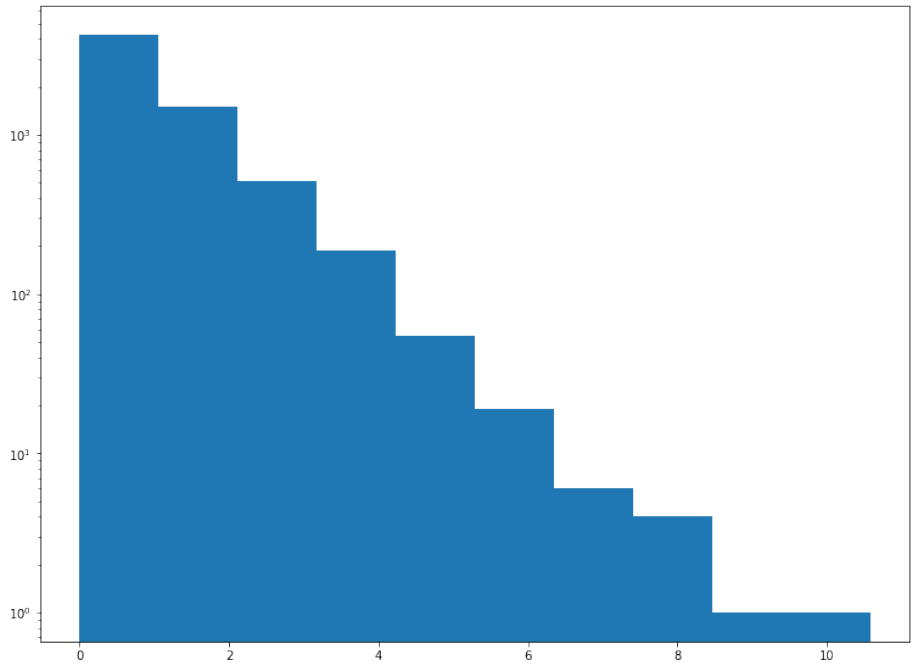
```
plt.imshow(control_mat);
```




```
plt.hist(inr_mat.flat)
plt.yscale('log');
```

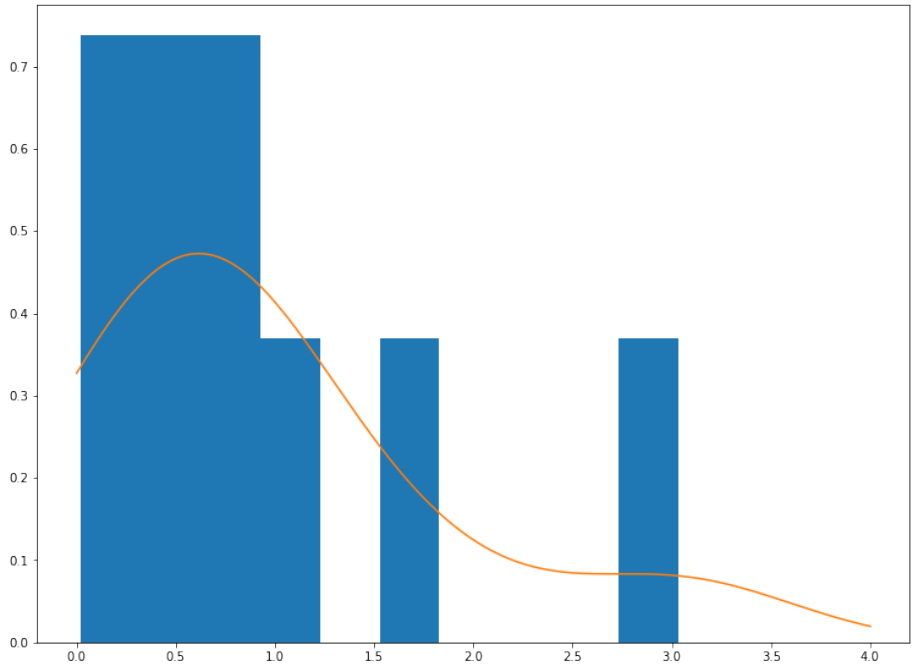



```
plt.hist(control_mat.flat)
plt.yscale('log');
```

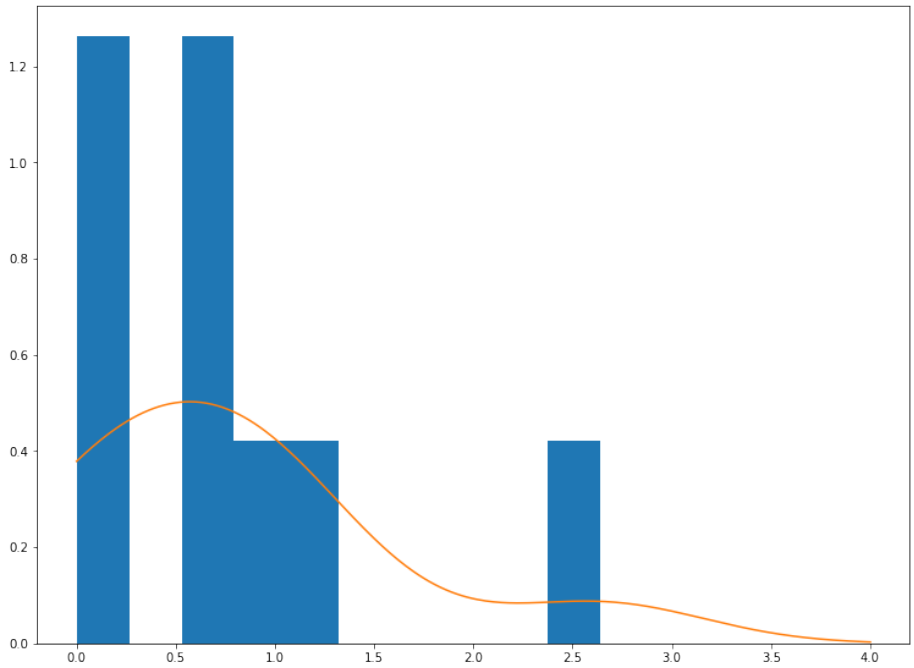


Histograms

```
inr_varerrs = [np.abs(initial_vars[i] - np.var(r)) for (i, r) in enumerate(inr_vars)]
plt.hist(inr_varerrs, density=True)
vs = np.linspace(0, 4, 300)
inr_kde = stats.gaussian_kde(inr_varerrs)
plt.plot(vs, inr_kde(vs))
plt.show()
```




```
inr_varerrs = [np.abs(initial_vars[i] - np.var(stats.expon.rvs(size=5*5))) for i in
↪  range(len(initial_vars))]
plt.hist(inr_varerrs, density=True)
vs = np.linspace(0, 4, 300)
inr_kde = stats.gaussian_kde(inr_varerrs)
plt.plot(vs, inr_kde(vs))
plt.show()
```

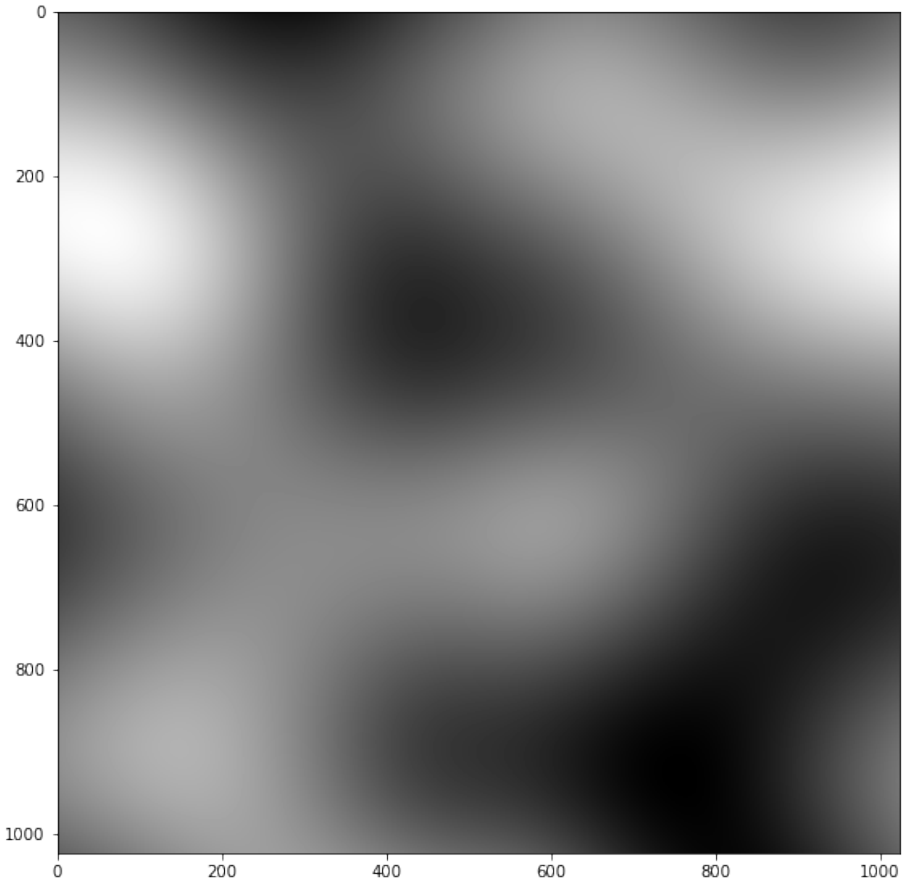


1.4 Perlin noise

```python
# From https://github.com/pvigier/perlin-numpy/blob/8e3ea24a39e938f631f4101294dcda4ef92bc633/perlin2d.py#L3-L41
def generate_perlin_noise_2d(shape, res, tileable=(False, False)):
    def f(t):
        return 6*t**5 - 15*t**4 + 10*t**3

    delta = (res[0] / shape[0], res[1] / shape[1])
    d = (shape[0] // res[0], shape[1] // res[1])
    grid = np.mgrid[0:res[0]:delta[0],0:res[1]:delta[1]].transpose(1, 2, 0) % 1
    # Gradients
    angles = 2*np.pi*np.random.rand(res[0]+1, res[1]+1)
    gradients = np.dstack((np.cos(angles), np.sin(angles)))
    if tileable[0]:
        gradients[-1,:] = gradients[0,:]
    if tileable[1]:
        gradients[:,-1] = gradients[:,0]
    gradients = gradients.repeat(d[0], 0).repeat(d[1], 1)
    g00 = gradients[    :-d[0],    :-d[1]]
    g10 = gradients[d[0]:     ,    :-d[1]]
    g01 = gradients[    :-d[0],d[1]:     ]
    g11 = gradients[d[0]:     ,d[1]:     ]
    # Ramps
    n00 = np.sum(np.dstack((grid[:,:,0]  , grid[:,:,1]  )) * 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
```

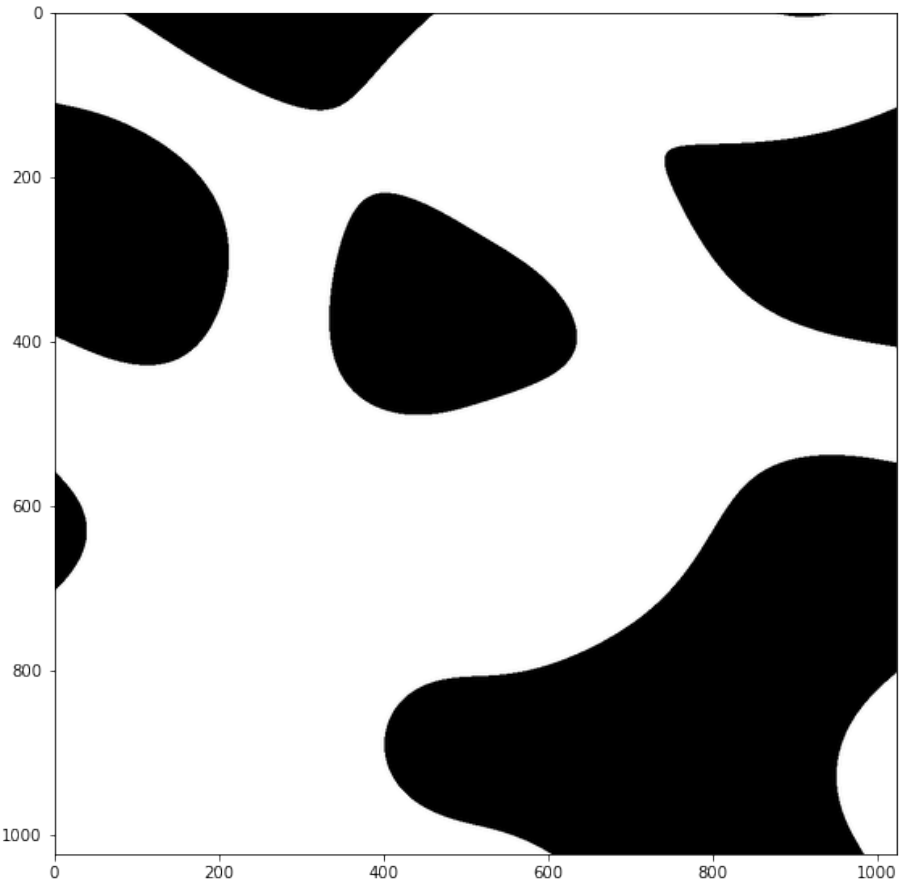
```
    t = f(grid)
    n0 = n00*(1-t[:,:,0]) + t[:,:,0]*n10
    n1 = n01*(1-t[:,:,0]) + t[:,:,0]*n11
    return np.sqrt(2)*((1-t[:,:,1])*n0 + t[:,:,1]*n1)

def generate_fractal_noise_2d(shape, res, octaves=1, persistence=0.5, lacunarity=2, tileable=(False,
↪  False)):
    noise = np.zeros(shape)
    frequency = 1
    amplitude = 1
    for _ in range(octaves):
        noise += amplitude * generate_perlin_noise_2d(shape, (frequency*res[0], frequency*res[1]),
        ↪  tileable)
        frequency *= lacunarity
        amplitude *= persistence
    return noise
```

```
pn = generate_perlin_noise_2d((1024, 1024), (2, 2))
plt.imshow(pn, interpolation='lanczos');
```

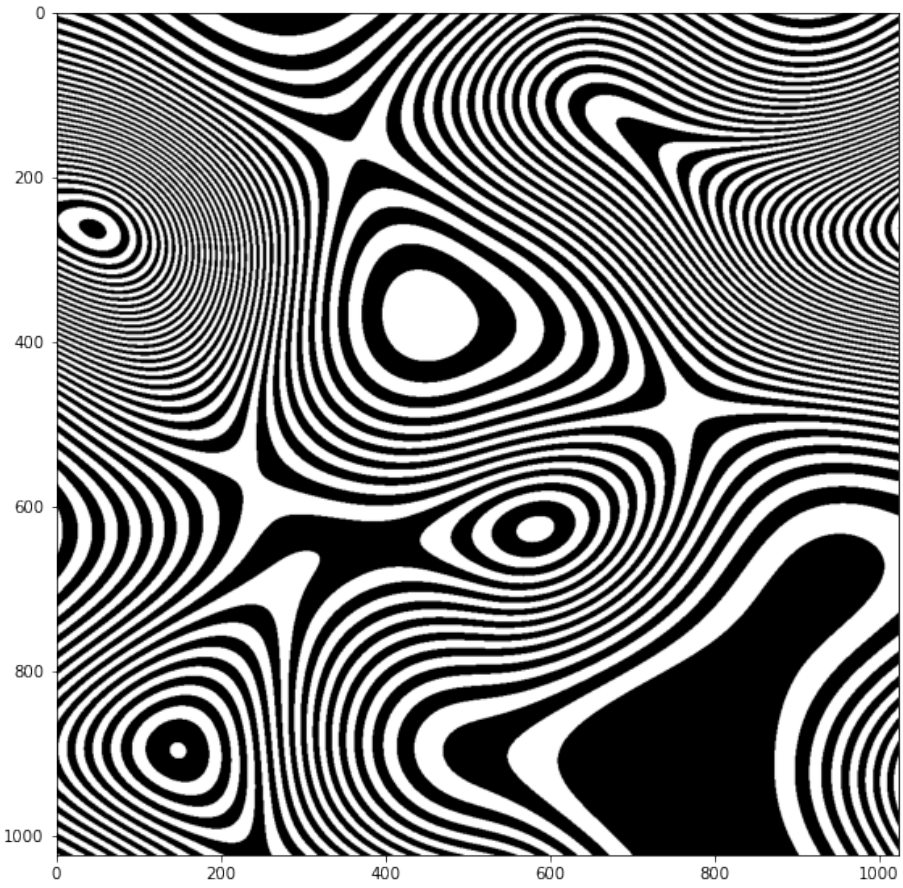



```
pmin, pmax = np.min(pn), np.max(pn)
```

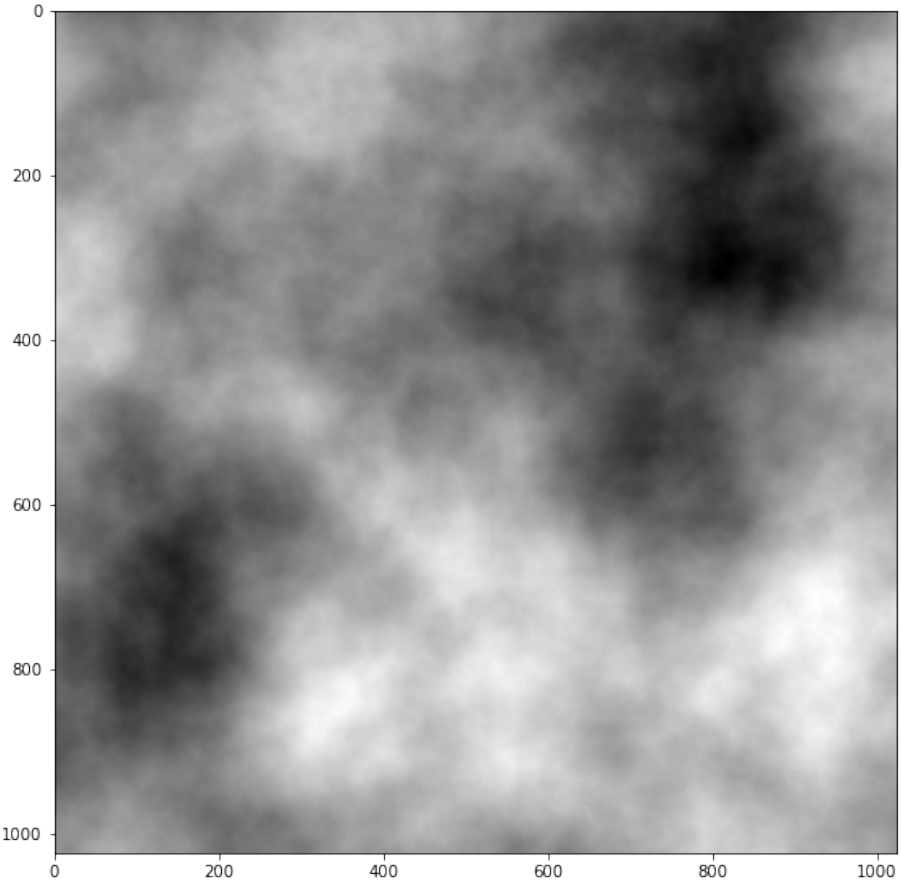
```
pn = (pn - pmin) / (pmax - pmin) # Rescale to exactly [0, 1] for thresholding
plt.imshow((0.3 < pn) & (pn < 0.7), interpolation='lanczos');
```



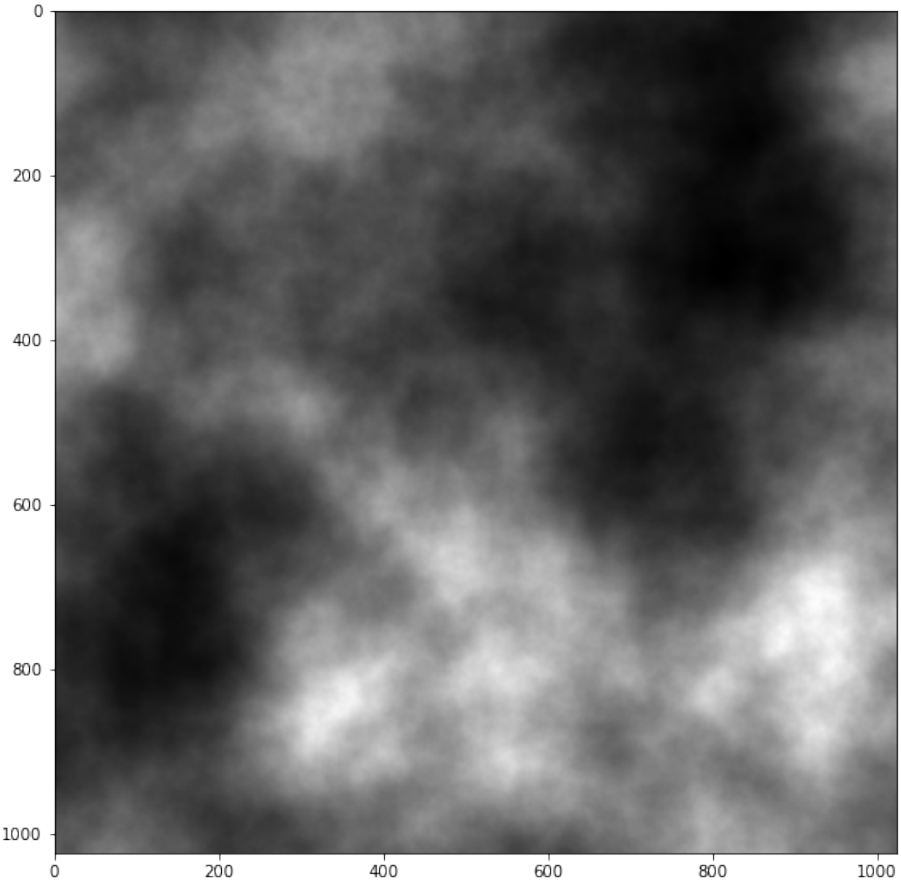

```
plt.imshow(np.floor((pn**2)*64) % 2);
```



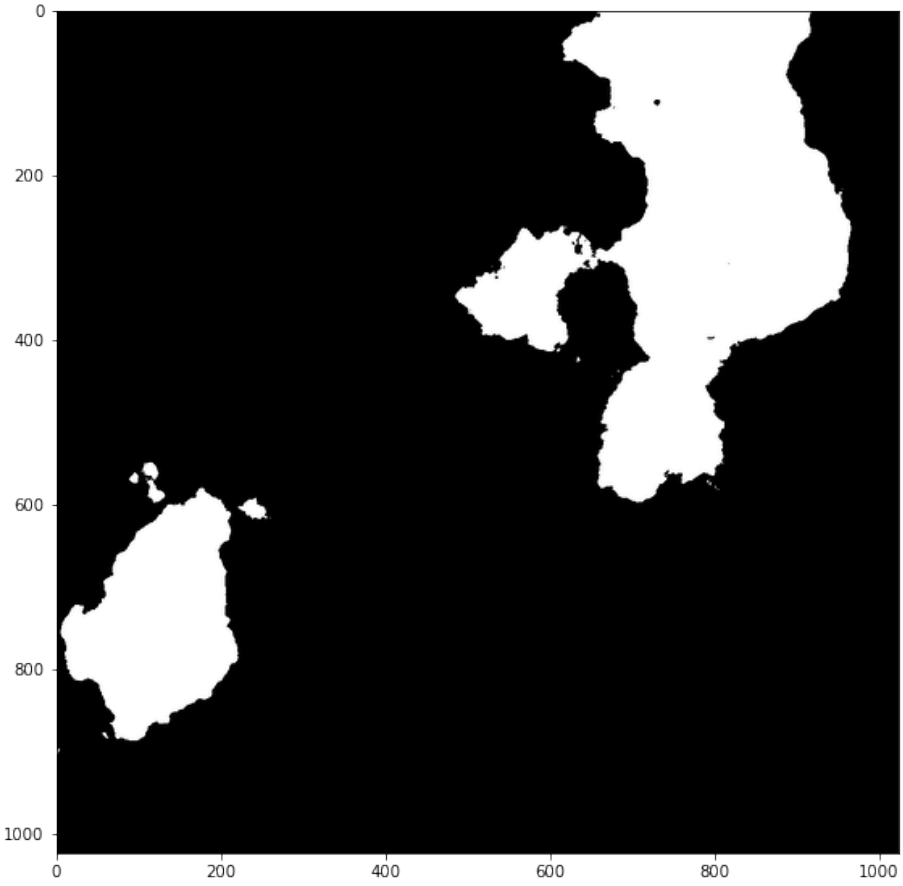

```
pfn = generate_fractal_noise_2d((1024, 1024), (2, 2), octaves=10, persistence=0.5)
pfn = 0.5 + (pfn / 4) # Rescale to within [0, 1]
plt.imshow(pfn, interpolation='lanczos');
```



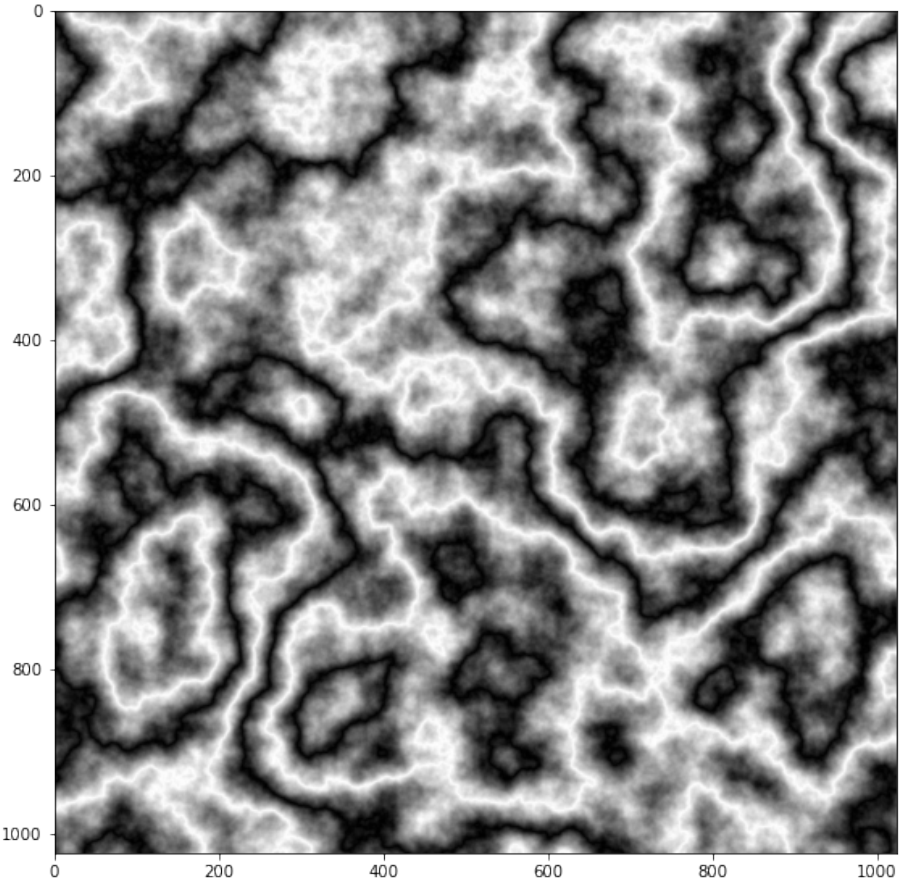

```
plt.imshow(pfn**3, interpolation='lanczos');
```



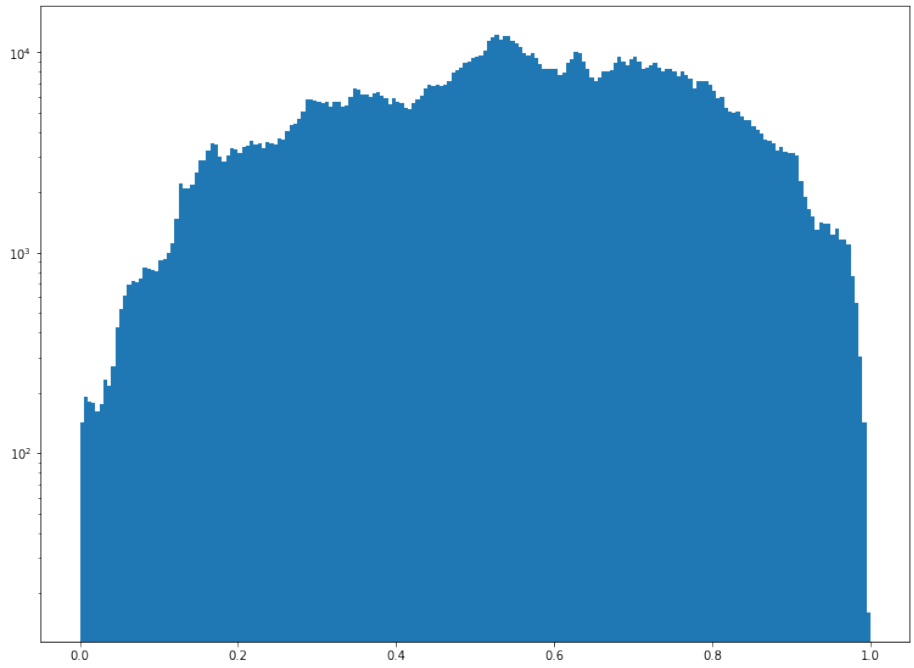

```
pmin, pmax = np.min(pfn), np.max(pfn)
pfn = (pfn - pmin) / (pmax - pmin) # Rescale to exactly [0, 1] for thresholding
plt.imshow((0.0 < pfn) & (pfn < 1/3), interpolation='lanczos');
```




```
plt.imshow(np.abs((((pfn**1)*64) % 16) - 8));
```




```
plt.hist(pfn.flat, 200)
plt.yscale('log');
```


10^4
10^3
10^2
0.0
0.2
0.4
0.6
0.8
1.0