Rensselaer 2020 REU Notebook

Alex Striff

May to July 2020

Contents

1 Project description

May 27, 2020 The aim of this REU project is to quantify the information present in images by the principled application of methods from statistical physics. The approach is to find a suitable notion of entropy which captures the salient features of particular kinds of images. We will consider a variety of features motivated by intuition or domain knowledge, and then move to machine learning as a tool for discovering other features.

2 Getting started

May 27, 2020 The initial goal is to characterize the most naïve calculation, which I'll call the *intensity entropy*. This does *not* take into account the spatial correlation of pixels in an image.

3 Intensity-level entropy

Given a discrete random variable X with support $\mathfrak{X}$, the *Shannon entropy* is

$$H = \sum_{x \in \mathfrak{X}} -P(x) \ln P(x).$$

The *intensity-level entropy* is the Shannon entropy of the empirical distribution of intensity values. Since we are usually dealing with 8-bit image data, we will usually measure the intensity entropy in *bits*.

```
import numpy as np
```

```
def shannon_entropy(h):
    """The Shannon entropy in bits"""
    return -sum(p*np.log2(p) if p > 0 else 0 for p in h)

def intensity_distribution(data, upper=256):
    """The intensity distribution of 8-bit `data`."""
    hist, _ = np.histogram(data, bins=range(upper+1), density=True)
    return hist

def intensity_entropy(data, upper=256):
    """The intensity-level entropy of 8-bit image data"""
    return shannon_entropy(intensity_distribution(data, upper))

def intensity_expected(f, data):
    """The intensity-distribution expected value of `f`."""
    return sum(p*f(p) for p in intensity_distribution(data))
```

4 Effect of smoothing on intensity-level entropy

```
import numpy as np
import numpy.linalg as linalg
import matplotlib.pyplot as plt
from PIL import Image, ImageFilter, ImageOps
from src.utilities import *
from src.intensity_entropy import *
```

4.1 Natural image

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

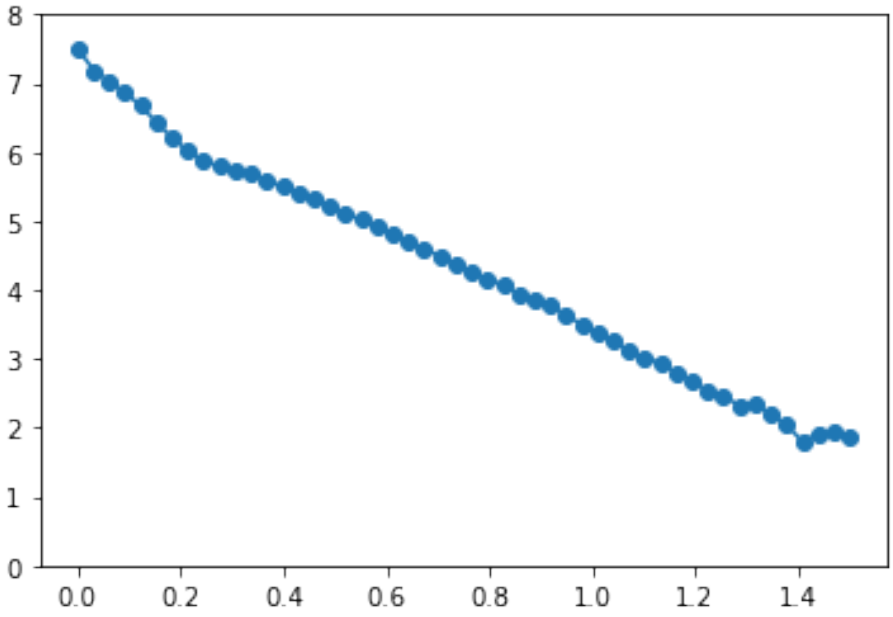
```
intensity_entropy(img)
```

```
7.51132356216608
```

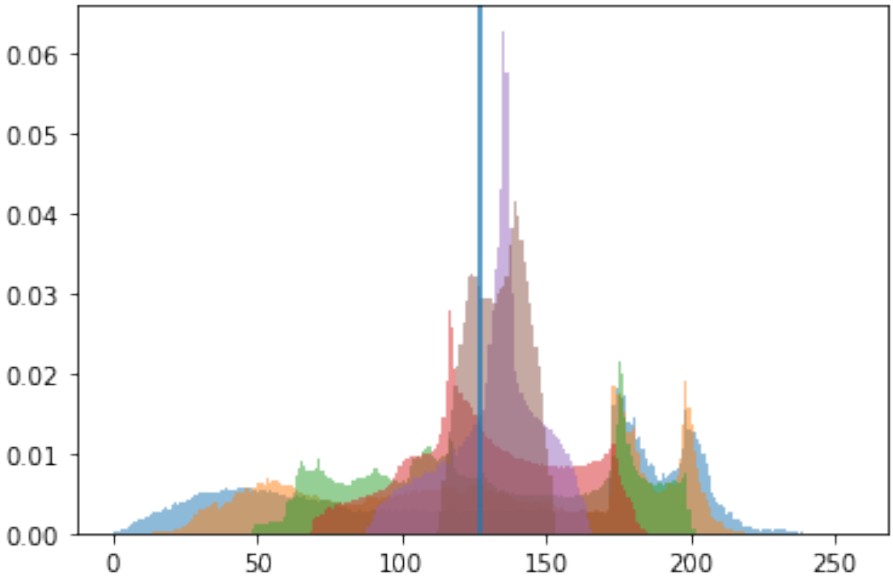
The problem with the intensity entropy is that it is usually near maximum (8 bits for these grayscale images).

```
def intensity_blur(img, scales, display=True):
    scale = max(np.shape(img))

    results = []
    for k in scales:
        simg = img.filter(ImageFilter.GaussianBlur(k * scale))
        data = np.array(simg)
        ihist, ibins = np.histogram(data, bins=range(256+1), density=True)
        S = shannon_entropy(ihist)
        if display:
            hist = plt.hist(ibins[:-1], ibins, weights=ihist, alpha=0.5)
            results.append((k, simg, hist, S))
        else:
            results.append((k, S))

    if display:
        plt.axvline(x=np.mean(np.array(img)))

    return results
```

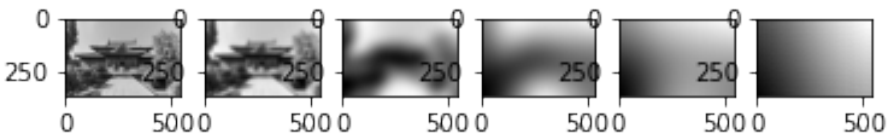
```
results = intensity_blur(img, np.linspace(0, 1.5, num=50), False)

plt.plot(*np.transpose(results), 'o-')
plt.ylim((0, 8))
plt.xlabel = "Smoothing"
plt.ylabel = "Intensity Entropy (bits)"
```




```
rimgs = [img for _, img, _, _ in intensity_blur(img, [0, 0.01, 0.05, 0.125, 0.25, 0.5])]
plt.show()
```

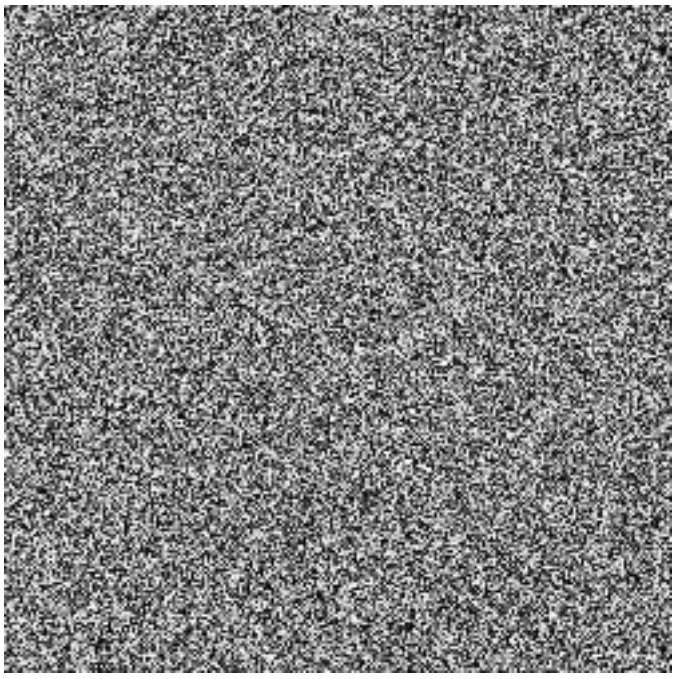



```
_, axarr = plt.subplots(1, len(rimgs))
for i, subimg in enumerate(rimgs):
    axarr[i].imshow(subimg, cmap='gray')
plt.show()
```



4.2 Random pixel values

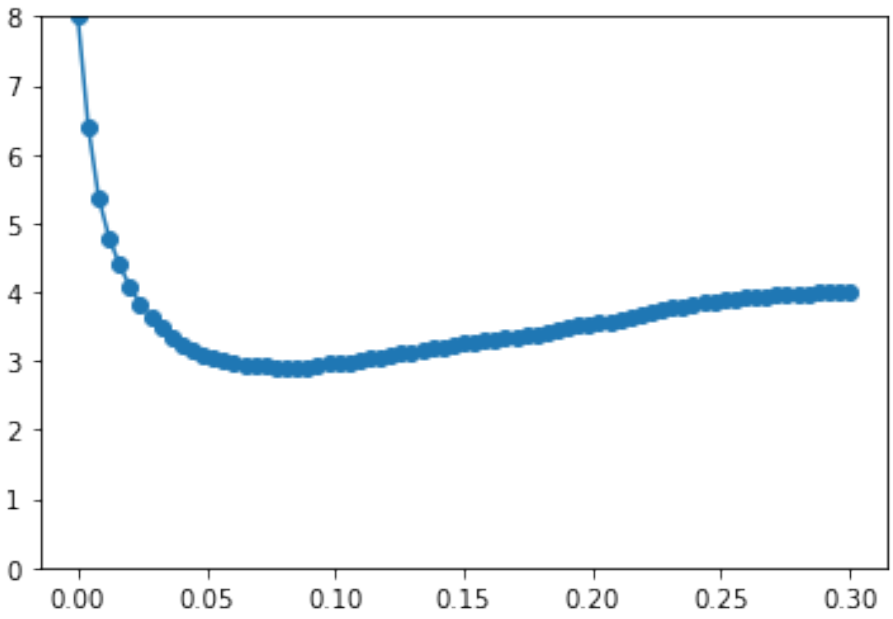
```
rsize = 250
randimg = Image.fromarray((256*np.random.rand(*2*[rsize])).astype('uint8'))
randimg
```



4.2.1 Beware: GIGO

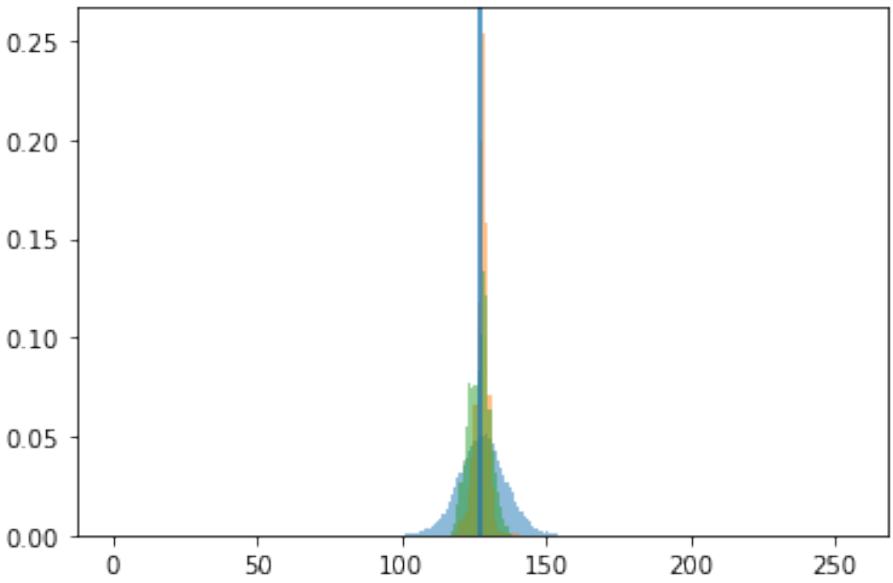
The boundary effects and discrete kernel of `ImageFilter.GaussianBlur` renders the data unreliable after the "minimum" of the intensity entropy with smoothing. This is immediately clear after even small smoothing for random pixel values, since there are no spatial correlations.

```
results = intensity_blur(randimg, np.linspace(0, 0.3, num=75), False)

plt.plot(*np.transpose(results), 'o-')
plt.ylim((0, 8))
plt.xlabel = "Smoothing"
plt.ylabel = "Intensity Entropy (bits)"
```



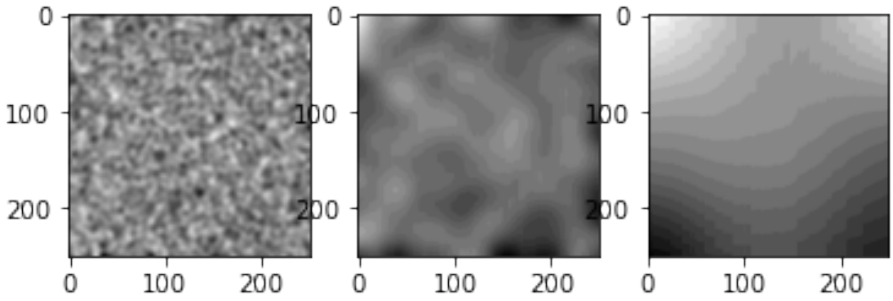

```
rimgs = [img for _, img, _, _ in intensity_blur(randimg, [0.01, 0.05, 0.25])]
```

```
plt.show()
```




```
_, axarr = plt.subplots(1, len(rimgs))
for i, subimg in enumerate(rimgs):
```

```
    axarr[i].imshow(subimg, cmap='gray')
plt.show()
```



The rightmost image should be uniform: the renormalization emphasizes incorrect deviations. These are what keep the intensity entropy from vanishing.

4.3 Comparing different levels of smoothing

Is composing n Gaussian blurs with variance σ^2 the same as doing one with variance $n\sigma^2$ (considering the boundary effects and discrete kernel)?

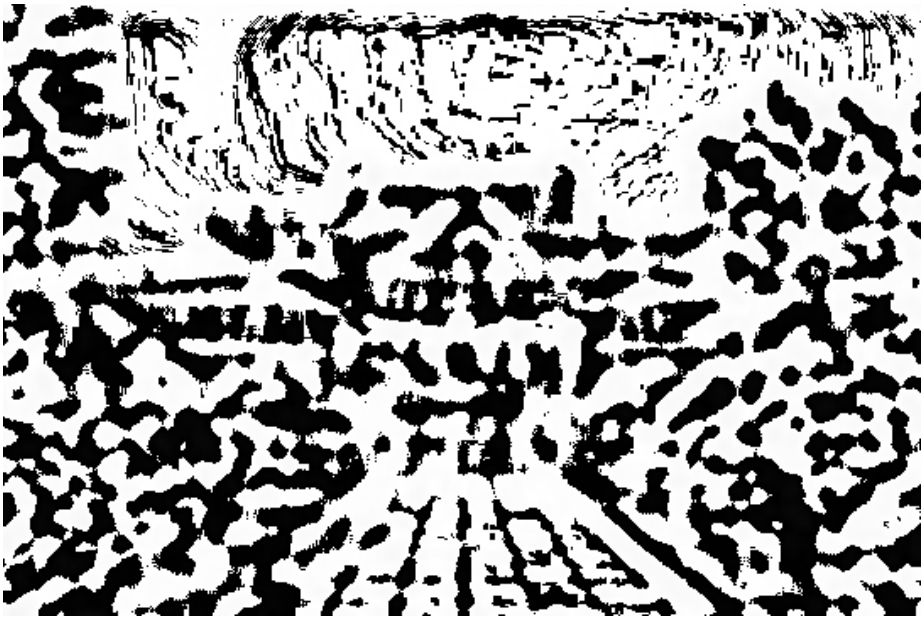
```
nsmooths = 10
cimg = img
oneimg = cimg.filter(ImageFilter.GaussianBlur(np.sqrt(nsmooths)*2))
oneimg
```

```
nimg = cimg
for _ in range(nsmooths):
```

```
    nimg = nimg.filter(ImageFilter.GaussianBlur(2))
nimg
```

Answer: **No**
The differences between results at different scales can be pretty wack.

```
Image.fromarray((255*rescale(np.array(nimg) - np.array(oneimg))).astype('uint8'))
```



```
smimg = img
smdiff = np.array(smimg.filter(ImageFilter.GaussianBlur(2))) -
↪  np.array(smimg.filter(ImageFilter.GaussianBlur(100)))
diffimg = Image.fromarray((255 * rescale(smdiff)).astype('uint8'))
diffimg
```

5 Local metrics

May 28, 2020

Given an image $I : X \times Y \to \mathbb{Z}_n$, we will now consider *local metrics* for the information it contains.

I want to be careful in understanding the statistical assumptions I am making, so I'll try to be explicit about distinguishing true distributions from empirical distributions, and how the assumptions behind postulating the existence of empirical distributions relate to the actual calculation being done. This should also aid in learning more solid probability theory.

5.1 Induced metrics

Definition 1 (Lists). Given a set S, the collection of lists of elements from S is

$$\mathsf{List}(S) = \bigcup_{n \in \mathbb{Z}_{\geq 0}} S^n,$$

where a list (tuple) $s \in S^n$ is a map $s : \mathbb{Z}_n \to S$ and $|s| = n$.

Definition 2 (Image distributions). An *image distribution* is a map D that takes an image I and produces a random variable $D(I) : \Omega \to E$.

We are constructing empirical distributions from image data according to some map $M : \mathsf{Img} \to \mathsf{List}(\Omega)$, which produces the list of values $V = M(I)$. Then the probability of $D(I)$ taking a value in a subset $S \subseteq E$ is

$$P(X \in S) = \frac{1}{|V|} \sum_{s \in S} \left| V^{-1}(\{s\}) \right|.$$

Example 1. The intensity-level entropy is a function of the *nonnegative* random variable from the image distribution of intensity values. That is, the map M takes an image and returns the list of its intensity values.

Definition 3 (Induced image distributions). Given an image distribution D, and a subset $S \subseteq \operatorname{dom} I$, we construct the *induced image distribution* $D|_S$ by

$$D|_S(I) = D(I|_S).$$

Definition 4 (Induced random variable). Given an image I, an image distribution D and collection of subsets $\{S_i\}$ of $\operatorname{dom} I$, a function H admits the random variables

$$H_i = (H \circ D|_{S_i})(I)$$

Definition 5. The *r-box* at $(x,\ y)$ is $B_r(x,\ y) = [x - r,\ x + r] \times [y - r,\ y + r]$.

Given two real random variables A and B with joint PDF $f_{A,B}(a,\ b)$, the PDF of their sum is

$$f_{A+B}(c) = \int_{-\infty}^{\infty} \mathrm{d}a\, f_{A,B}(a,\ a - c) = \int_{-\infty}^{\infty} \mathrm{d}b\, f_{A,B}(b - c,\ b). \tag{1}$$

For independent A and B, EQ. 1 reduces to $f_{A+B} = f_A * f_B$ over the marginals.

6 Kernels

Generalized to arbitrary functions on subregions of images.

```python
import numpy as np
```

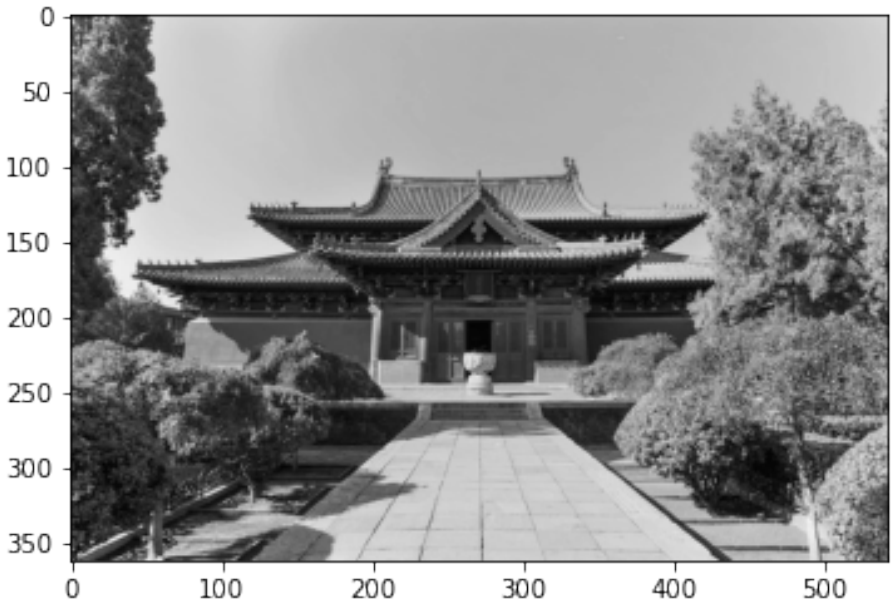
```python
def box(x, y, r):
    return np.s_[max(0, x-r) : x+r+1, max(0, y-r) : y+r+1]
def mapbox(r, f, a):
    return np.reshape([f(a[box(*i, r)]) for i in np.ndindex(np.shape(a))], np.shape(a))
def mapboxes(rs, f, a):
    return (mapbox(r, f, a) for r in rs)
def mapallboxes(f, a):
    return mapboxes(range(max(np.shape(a))), f, a)

def mapblocks(h, w, f, a):
    return np.array([[f(y) for y in np.array_split(x, w, axis=1)]
                     for x in np.array_split(a, h)])
```

7 Boxcar intensity-level entropy

```
import numpy as np
import numpy.linalg as linalg
import matplotlib.pyplot as plt
from PIL import Image, ImageFilter, ImageOps
from src.utilities import *
from src.intensity_entropy import *
from src.kernels import *
plt.rcParams['image.cmap'] = 'gray'
```

Let's compare the boxcar images for intensity entropy to those for a positive function on an image (the standard deviation) and for different functions of the induced intensity distribution.

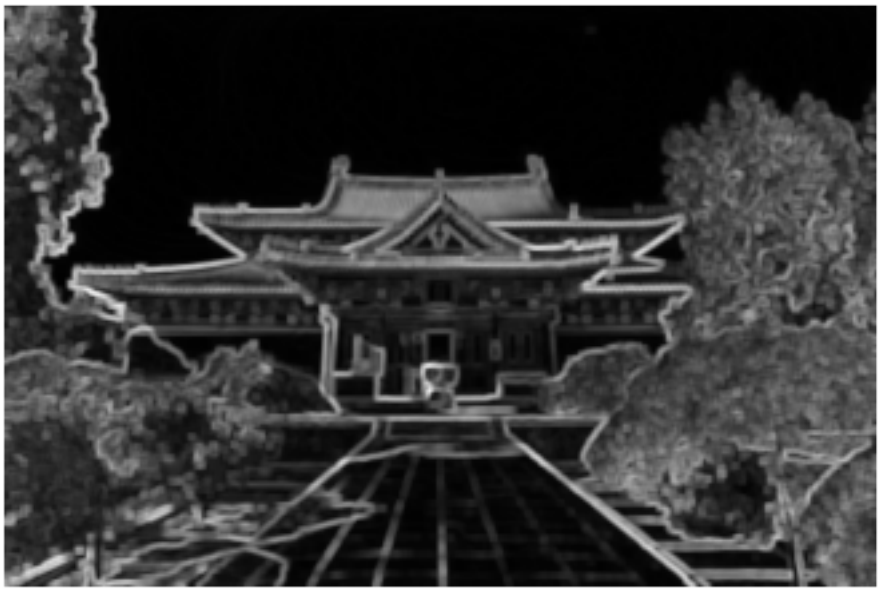
```
img = ImageOps.grayscale(Image.open('test.jpg'))
scale = max(np.shape(img))
data = np.array(img)
plt.imshow(img);
```




```
plt.imshow(img)
plt.axis('off')
plt.savefig('house_cmap.png', dpi=600, pad_inches=0, bbox_inches='tight')
```

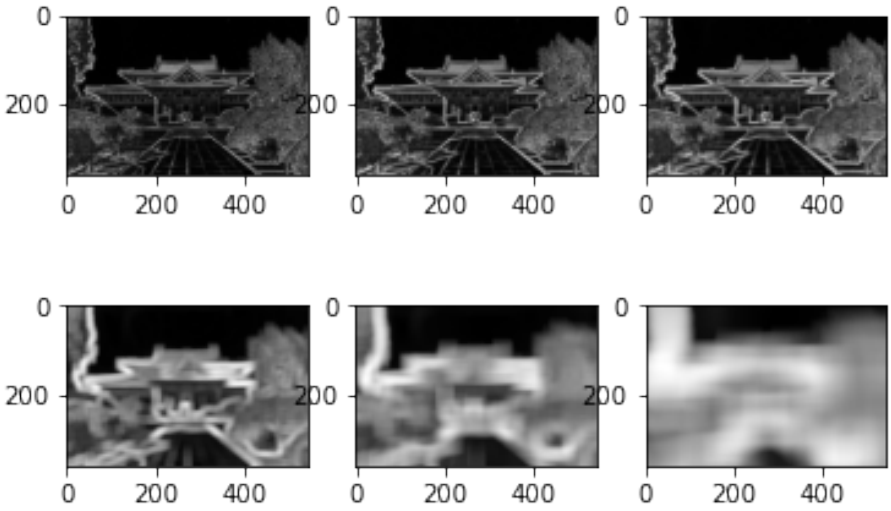
7.1 Standard deviation

```
plt.imshow(mapbox(2, np.std, np.array(img)))
plt.axis('off')
plt.savefig('house_std.png', dpi=600, pad_inches=0, bbox_inches='tight')
```



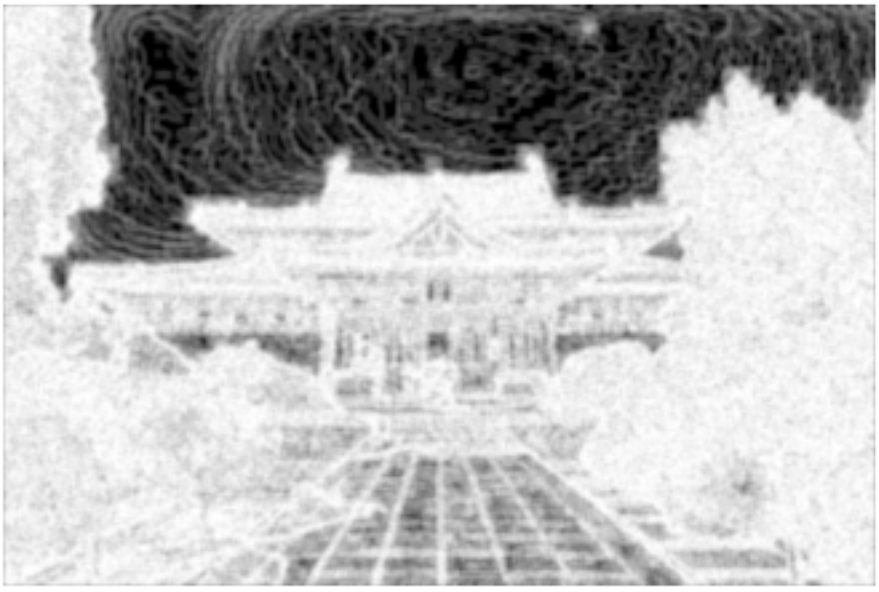
```
boxos = list(mapboxes([1,2,3,10,20,50], np.std, np.array(img)))
```

```
_, axarr = plt.subplots(2, np.ceil(len(boxos)/2).astype('int'))
for i, subimg in enumerate(boxos[:3]):
    axarr[0,i].imshow(subimg)
for i, subimg in enumerate(boxos[3:]):
    axarr[1,i].imshow(subimg)
plt.show()
```



7.2 Intensity entropy

```
plt.imshow(mapbox(2, intensity_entropy, np.array(img)))
plt.axis('off')
plt.savefig('house_entropy_2.png', dpi=600, pad_inches=0, bbox_inches='tight')
```

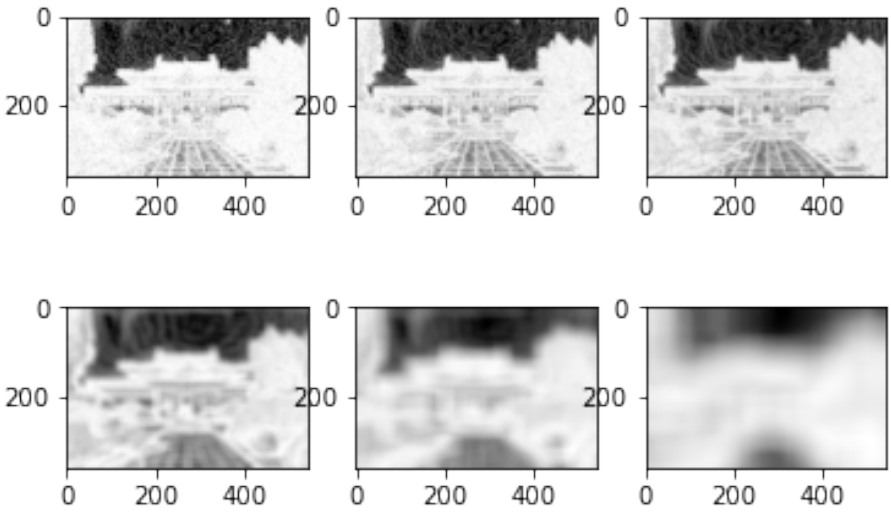


```
plt.imshow(mapbox(20, intensity_entropy, np.array(img)))
plt.axis('off')
plt.savefig('house_entropy_20.png', dpi=600, pad_inches=0, bbox_inches='tight')
```

```
boxSes = list(mapboxes([1,2,3,10,20,50], intensity_entropy, np.array(img)))
```

```
_, axarr = plt.subplots(2, np.ceil(len(boxSes)/2).astype('int'))
for i, subimg in enumerate(boxSes[:3]):
```

```
    axarr[0,i].imshow(subimg)
for i, subimg in enumerate(boxSes[3:]):
    axarr[1,i].imshow(subimg)
plt.show()
```

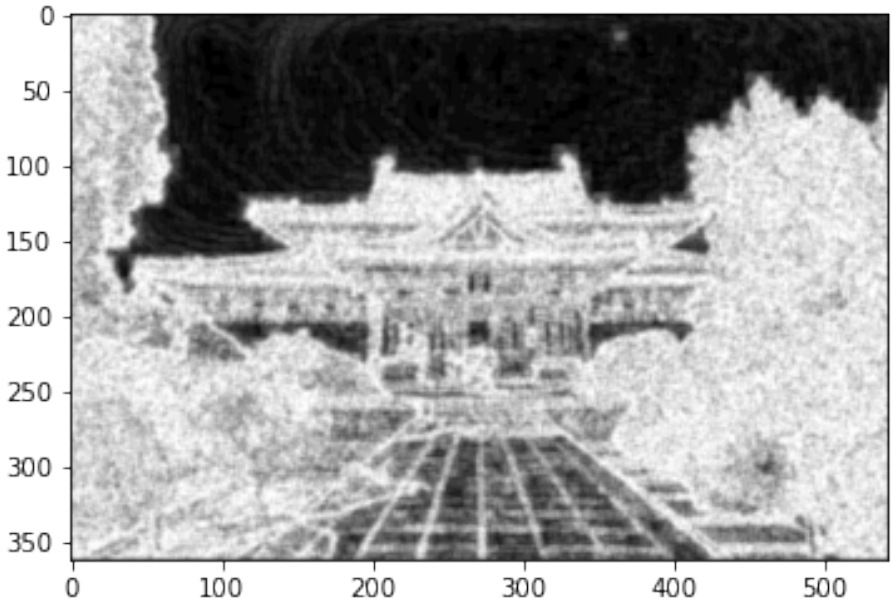


7.3 Replace surprisal with other functions

To what extent do the surprisal-related results depend upon the specific form of the *surprisal* $x \mapsto -\log(x)$ in the expected value of the intensity distribution? We will replace the expected surprisal with the expected f, for different functions f on the empirical probabilities of a pixel taking some intensity.

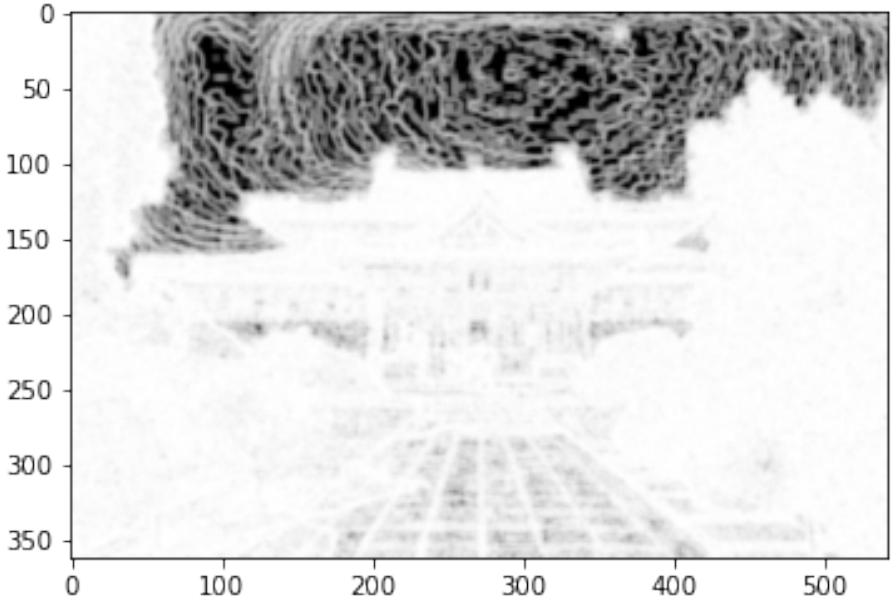
Laurent: $p \mapsto -1 + 1/p$.

```
plt.imshow(mapbox(2, lambda I: intensity_expected(lambda p: -1 + 1/p if p > 0 else 0, I), np.array(img)));
```



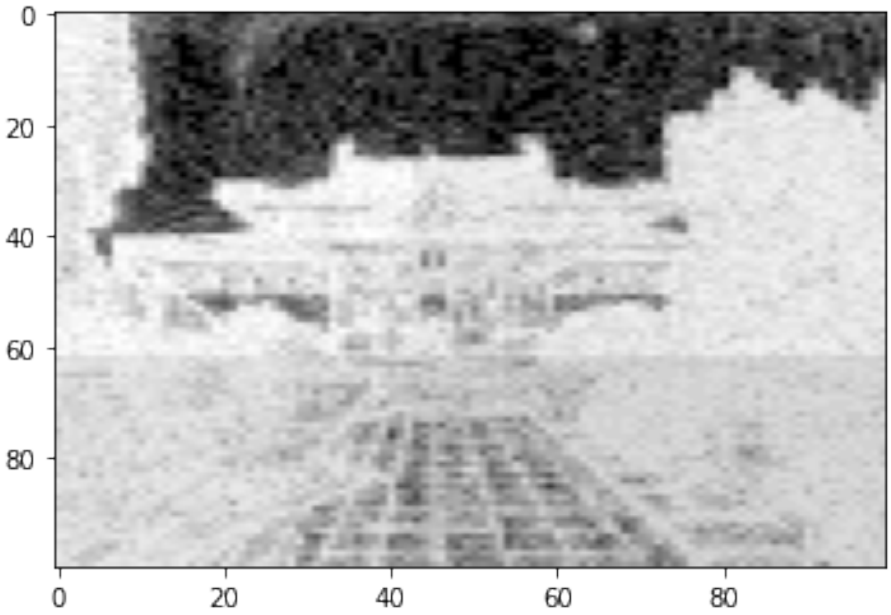
Taylor: $p \mapsto -(1+p)$.

```
plt.imshow(mapbox(2, lambda I: intensity_expected(lambda p: -(1+p), I), np.array(img)));
```

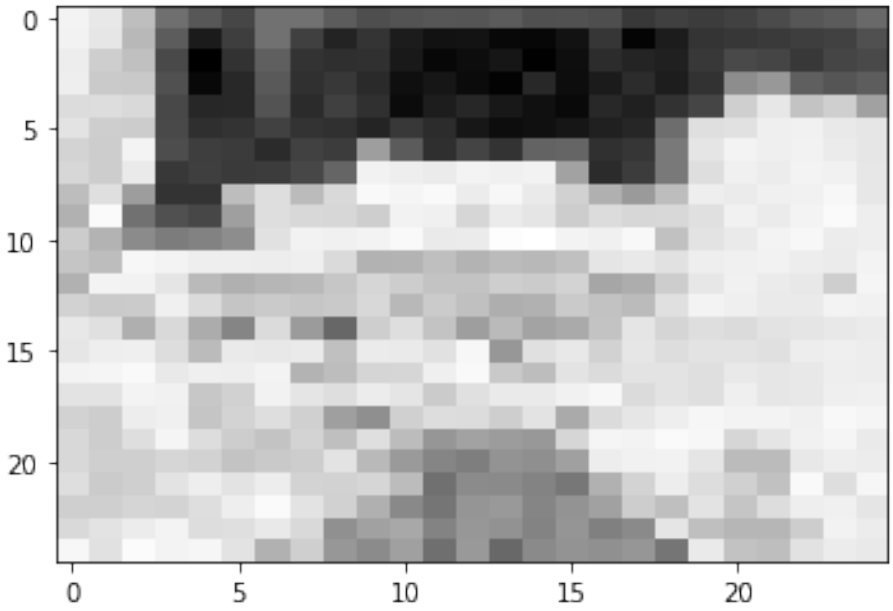


7.4 Intensity entropy on disjoint blocks

```
plt.imshow(mapblocks(100, 100, intensity_entropy, np.array(img)),
           aspect=np.divide(*np.shape(img)));
```




```
plt.imshow(mapblocks(25, 25, intensity_entropy, np.array(img)),
           aspect=np.divide(*np.shape(img)));
```



8 Fractal dimensions

May 29, 2020

The previous results hint at characterizing the growth of the intensity entropy with different discretizations.

Definition 6. The *Rényi entropy of order* $\alpha \geq 0$ of a discrete random variable X with support $\mathfrak{X}$ is

$$H_\alpha(X) = \frac{1}{1-\alpha} \log \sum_{x \in \mathfrak{X}} P(x)^\alpha = \frac{\alpha}{1-\alpha} \log \|P\|_\alpha,$$

where $\|P\|_\alpha$ denotes the α-norm of the vector of probability values. The limit $\alpha \to 1$ reproduces the Shannon entropy.

Definition 7. Given a real random variable X, define a discretized random variable

$$\langle X \rangle_\varepsilon = \frac{\lfloor \varepsilon X \rfloor}{\varepsilon}.$$

Then the *generalized dimension* of X is

$$d_\alpha(X) = \lim_{\varepsilon \to 0} \frac{H_\alpha(\langle X \rangle_\varepsilon)}{\log \varepsilon} = \lim_{\varepsilon \to 0} \frac{\alpha}{1-\alpha} \log \left(\|\langle X \rangle_\varepsilon\|_\alpha - \varepsilon \right).$$

The case $\alpha \to 1$ is the *information dimension* of X. The generalized dimension may be estimated from linear regression of $H_\alpha(\langle X \rangle_\varepsilon)$ with $\log \varepsilon$ as the independent variable.

9 Fractal dimension regression

```
import numpy as np
import numpy.linalg as linalg
import matplotlib.pyplot as plt
from PIL import Image, ImageFilter, ImageOps
from scipy import interpolate
from scipy import integrate
from src.intensity_entropy import *
from src.kernels import *
plt.rcParams['image.cmap'] = 'inferno'
```

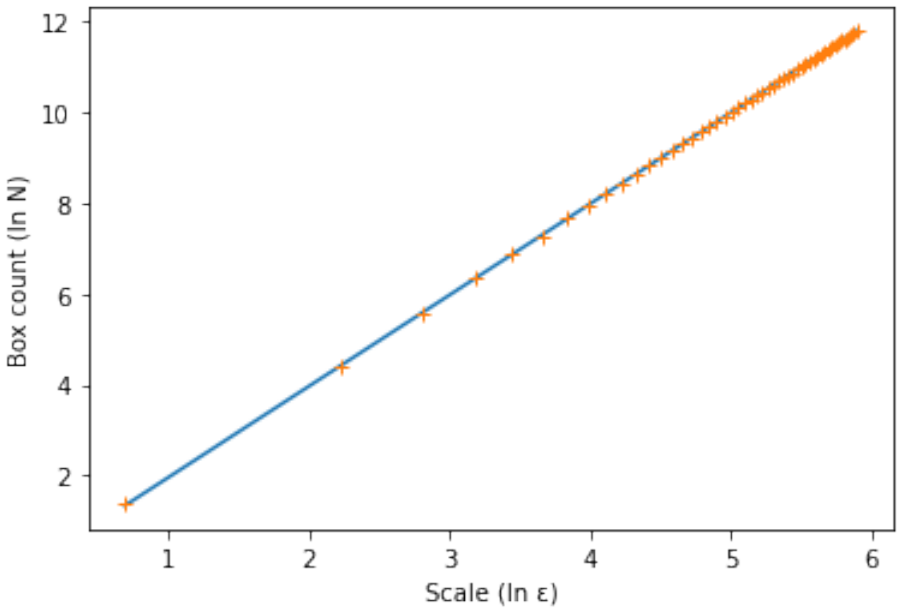
```
img = ImageOps.grayscale(Image.open('test.jpg'))
scale = max(np.shape(img))
data = np.array(img)
img
```

9.1 Box-counting dimension

```
def boxdim(data):
    εs = np.linspace(2, min(np.shape(data)))
    boxes = [np.log(np.sum(mapblocks(
        ε, ε, lambda x: 1 if np.any(x) else 0, data))) for ε in εs]
    logεs = np.log(εs)
    endεs = logεs[[0, -1]]
    dimfit = np.polyfit(np.log(εs), boxes, 1) # [slope, intercept]
    plt.plot(endεs, dimfit[0]*endεs + dimfit[1])
    plt.plot(logεs, boxes, '+')
    plt.xlabel('Scale (ln ε)')
    plt.ylabel('Box count (ln N)')
    return dimfit[0]
```

```
boxdim(data)
```

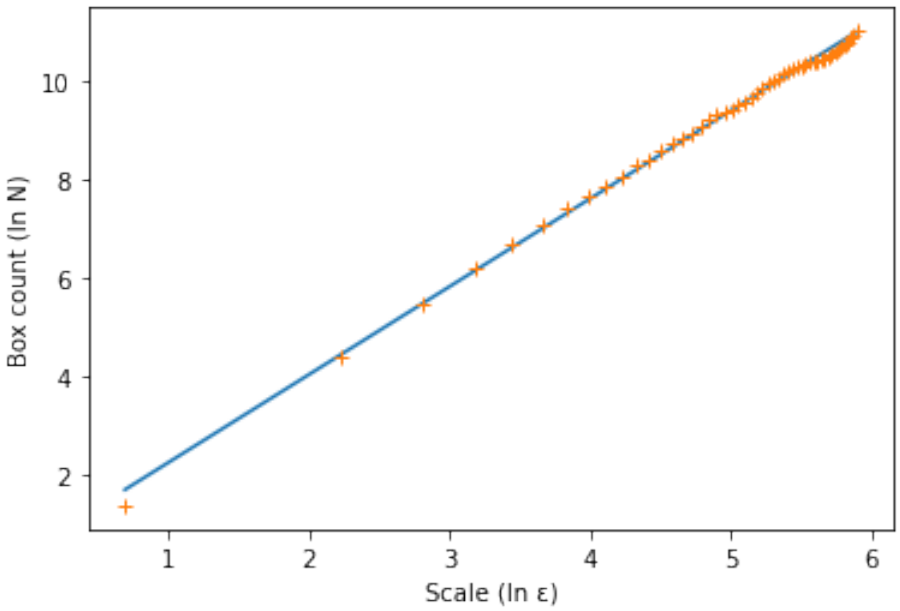
```
2.0087040269581435
```



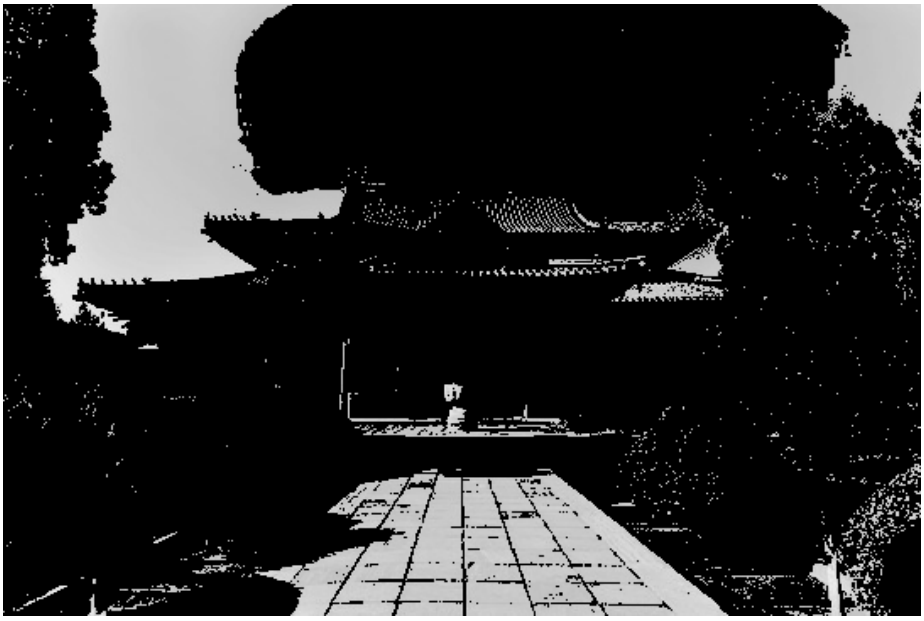

```
sky = data.copy()
sky[sky < 128+32] = 0
Image.fromarray(sky)
```

```
boxdim(sky)
```

1.7877778191348215

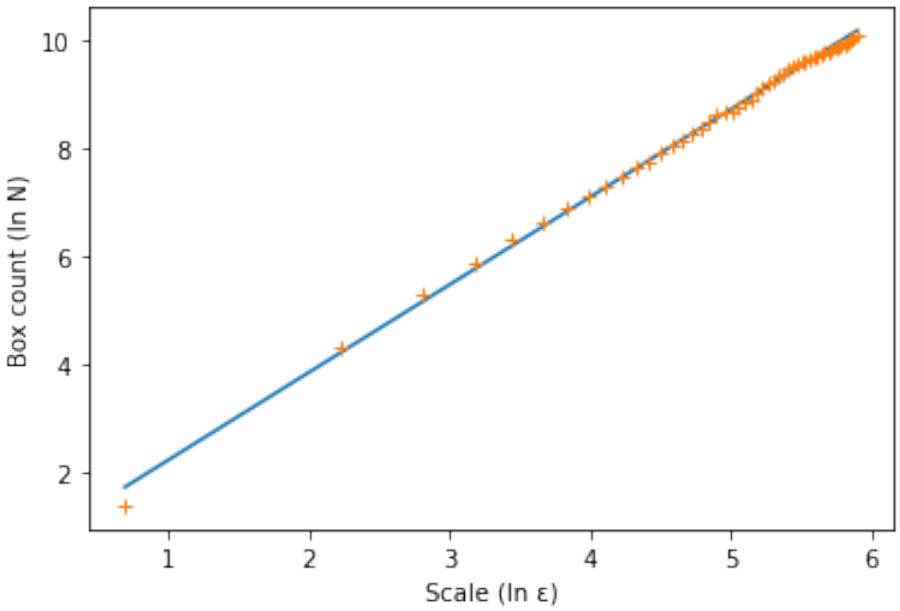



```
nosky = data.copy()
nosky[nosky < 128+64] = 0
Image.fromarray(nosky)
```



```
boxdim(nosky)
```

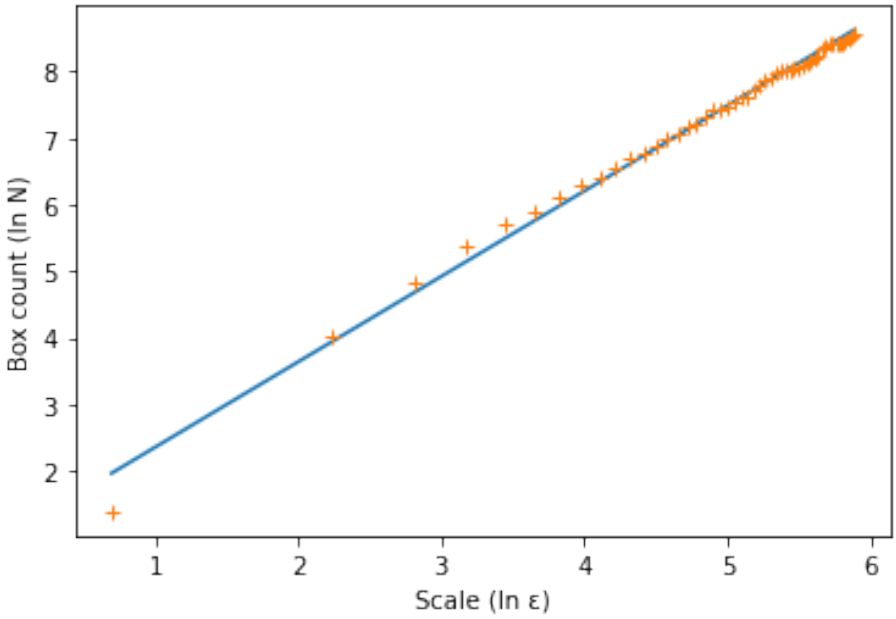
1.6214794967487127




```
dots = data.copy()
dots[nosky < 128+64+16] = 0
Image.fromarray(dots)
```

```
boxdim(dots)
```

```
1.2821025677557252
```



9.2 Information dimension

```
def discretize(f, a, b, ε, N=20):
    return [integrate.simps(f(np.linspace(c - ε/2, c + ε/2, N)), dx=ε / (N - 1))
            for c in np.arange(a + ε/2, b, ε)]
```

```
def infodim(dist, s=1e-5):
    l = len(dist)
    spl = interpolate.splrep(range(l), dist, s=s)
    f = lambda x: interpolate.splev(x, spl)

    εs = l / np.linspace(10, l)
    logεs = -np.log2(εs)
    endεs = logεs[[0, -1]]
    entropies = [shannon_entropy(discretize(f, 0, l, ε)) for ε in εs]
    dimfit, cov = np.polyfit(logεs, entropies, 1, cov='unscaled')

    plt.plot(endεs, dimfit[0]*endεs + dimfit[1])
    plt.plot(logεs, entropies, '+')
    plt.xlabel('Scale (lg ε)')
    plt.ylabel('Shannon entropy (bits)')

    return dimfit[0], cov[0,0]
```

The Gaussian distribution

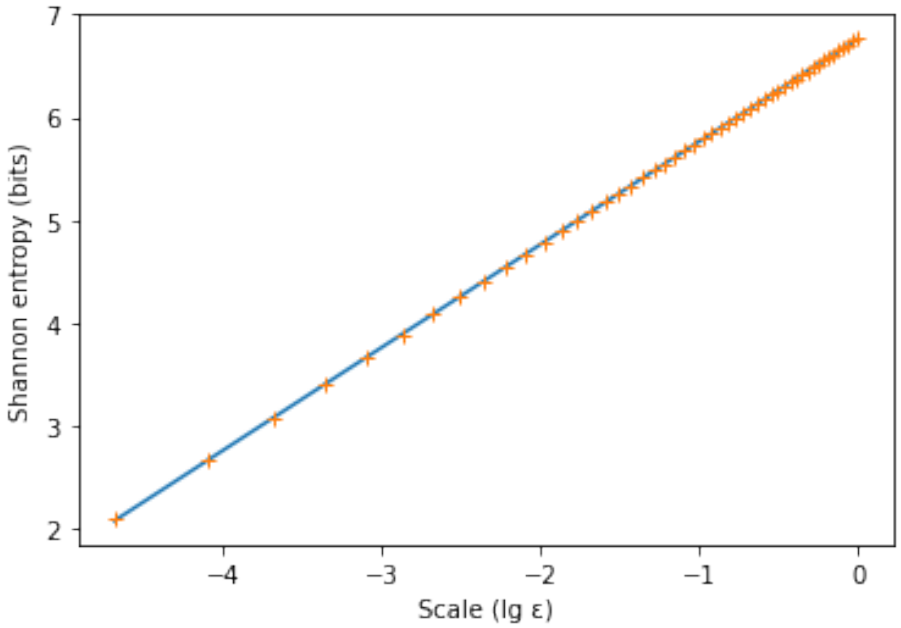
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

is continuous, so its information dimension is 1.

```
def gaussian(μ, σ, x):
    return np.exp(-(x - μ)**2 / (2*σ**2)) / (σ*np.sqrt(2*np.pi))
```

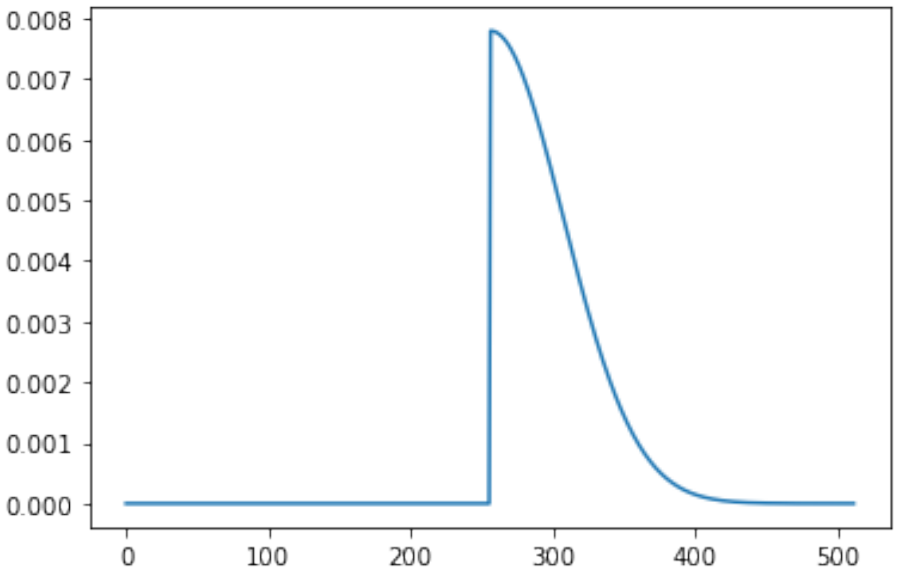
```
infodim((10/256) * gaussian(0, 1, np.linspace(-5, 5, 256)))
```

```
(1.0014184290221988, 0.015707497682893923)
```



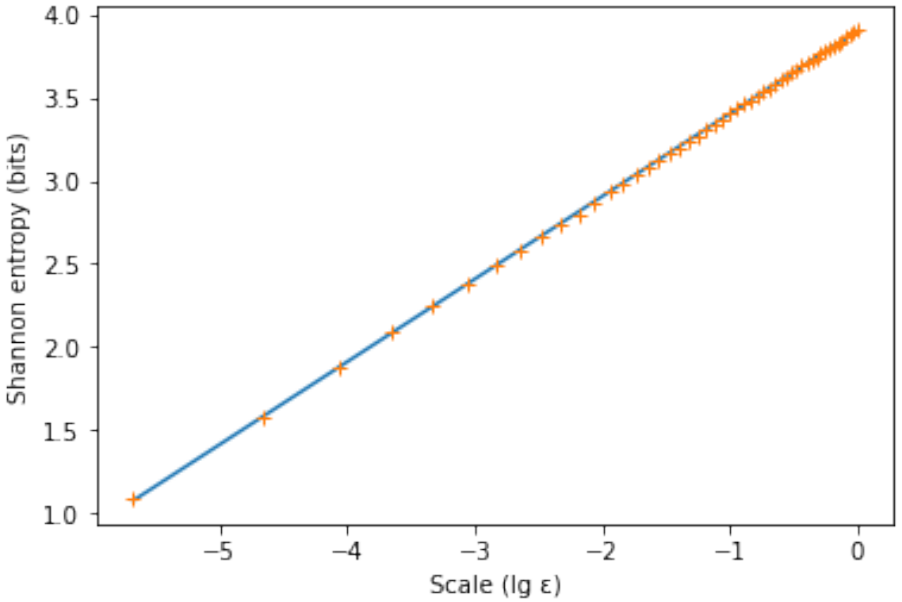
The rectified Gaussian distribution $g(x) = \Theta(x)f(x) + \delta(x)/2$ is half-continuous, so its information dimension is 1/2.

```
dist = np.concatenate([[0]*256, (5/256)*gaussian(0, 1, np.linspace(0, 5, 256))])
plt.plot(dist);
```



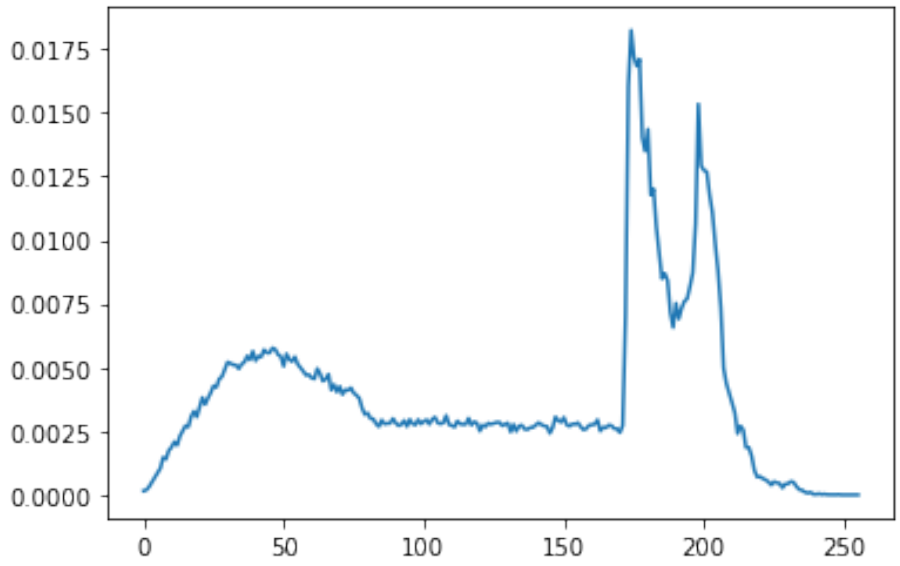

```
infodim(dist)
```

```
(0.4979088715795226, 0.012313889825394398)
```



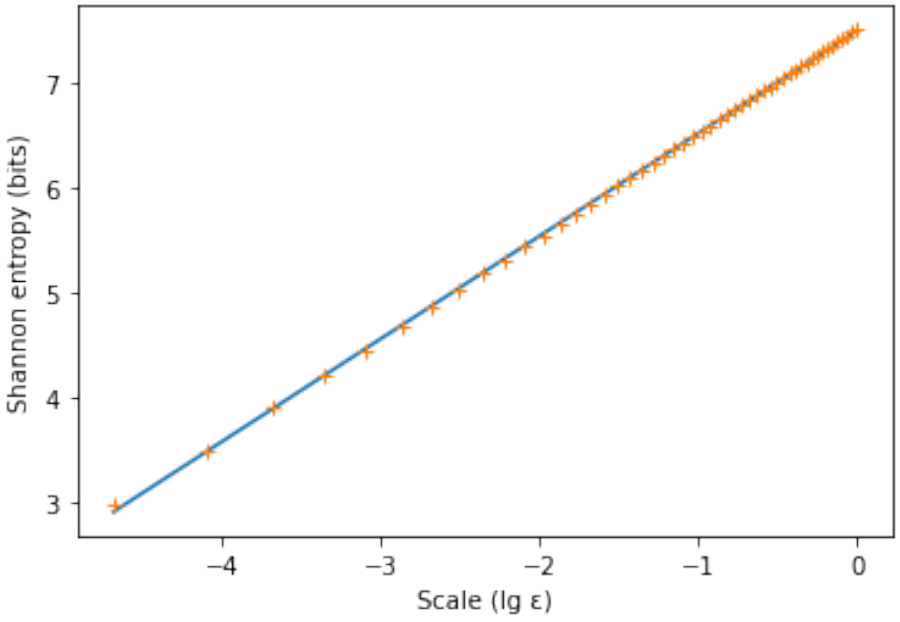
Now that we've validated `infodim`, what does it say about the intensity distribution of an image?

```
dist = intensity_distribution(img)
plt.plot(dist);
```




```
infodim(dist)
```

```
(0.98265843047326, 0.015707497682893923)
```



10 Probability and inference

May 30, 2020

Let's look at a simple inference problem before considering images. This example illustrates how the approach founded on probability theory differs from the naïve statistical approach usually taken by physicists.

Example 2 (Biased coin tosses). Consider tossing a biased coin N times to obtain n heads. What is the probability p' that the next coin toss comes up heads?

The temptation is to claim n/N as the probability, but this is *incorrect* if we want to allow all consistent biases. The problem with this solution is that the most probable bias is assumed to be the true bias.

The probability of getting m heads if a single head has probability p is

$$P(m \mid p) = \binom{N}{m} p^m (1-p)^{N-m}.$$

We have no other information, so we assume that all of the biases are equally likely. This means that $P(p)$ is constant (the uniform prior). The distribution of biases p given the observation of m heads is then

$$P(p \mid m) = \frac{P(m \mid p)P(p)}{P(m)} = \frac{P(m \mid p)P(p)}{\int_0^1 \mathrm{d}\tilde{p}\, P(m \mid \tilde{p})P(\tilde{p})} = \frac{P(m \mid p)}{\int_0^1 \mathrm{d}\tilde{p}\, P(m \mid \tilde{p})}.$$

We compute that

$$P(m) = \binom{N}{m} \int_0^1 \mathrm{d}p\, p^m (1-p)^{N-m} = \binom{N}{m} \frac{m!(N-m)!}{(N+1)!} = \frac{1}{N+1},$$

so the next coin toss is heads with probability

$$\begin{aligned} p' &= \int_0^1 \mathrm{d}p\, P(\text{head} \mid n,\, p) P(p \mid n) = \int_0^1 \mathrm{d}p\, p\, P(p \mid n) \\ &= \int_0^1 \mathrm{d}p\, p(N+1) \binom{N}{n} p^n (1-p)^{N-n} = \frac{n+1}{N+2}. \end{aligned}$$

For $n = 3$ and $N = 10$, $p' = 0.33$. This is a more conservative estimate than $p' = 0.30$ from the most probable bias.

11 Ising images

June 1, 2020

What happens if we apply a model from statistical physics to an image?

12 Ising images

```
import numpy as np
import numpy.linalg as linalg
import matplotlib.pyplot as plt
from PIL import Image, ImageFilter, ImageOps
import imageio
plt.rcParams['image.cmap'] = 'gray'
```

```
from ipywidgets import IntProgress
from IPython.display import display
import time
```

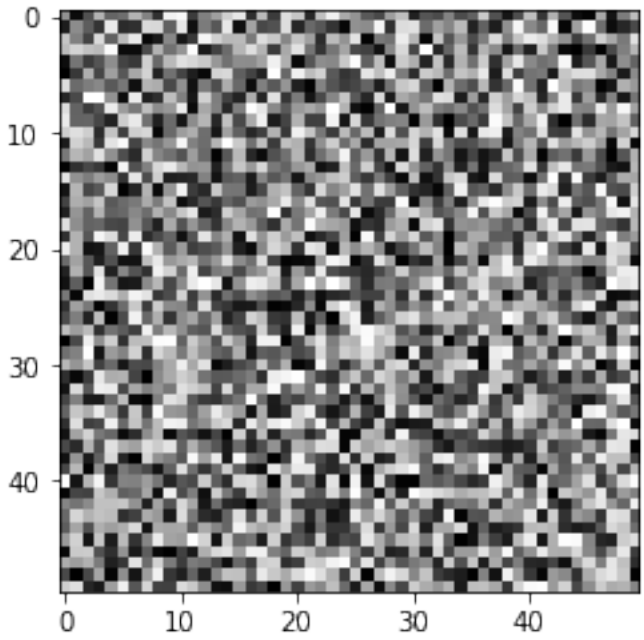
12.1 Standard Ising (on a torus)

In grayscale for fun.

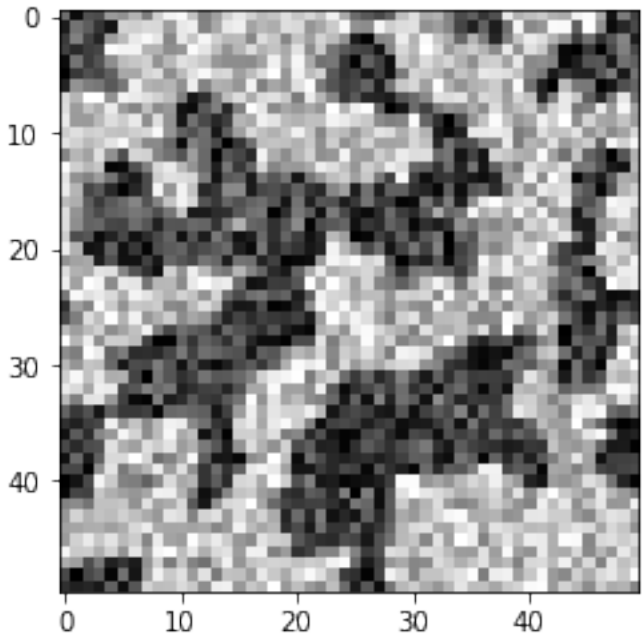
```
def neighbors(a, i, j):
    return np.hstack([a[:,j].take([i-1,i+1], mode='wrap'),
                      a[i,:].take([j-1,j+1], mode='wrap')])
```

```
def energy(img, i, j):
    return -1 + np.sum(np.abs(img[i, j] - neighbors(img, i, j)))

def isingstep(β, img):
    w, h = np.shape(img)
    i = np.random.randint(w)
    j = np.random.randint(h)
    E0 = energy(img, i, j)
    img[i, j] *= -1
    E1 = energy(img, i, j)
    P = np.exp(-β*(E1 - E0)) if E1 > E0 else 1
    if np.random.rand() > P: # Restore old
        img[i, j] *= -1
    return img
```

```
img = 2*np.random.rand(50, 50) - 1
plt.imshow(img);
```




```
n = 100000
for i in range(n):
    isingstep(3 * (np.pi / 2) / np.arctan(n - i), img)
plt.imshow(img);
```



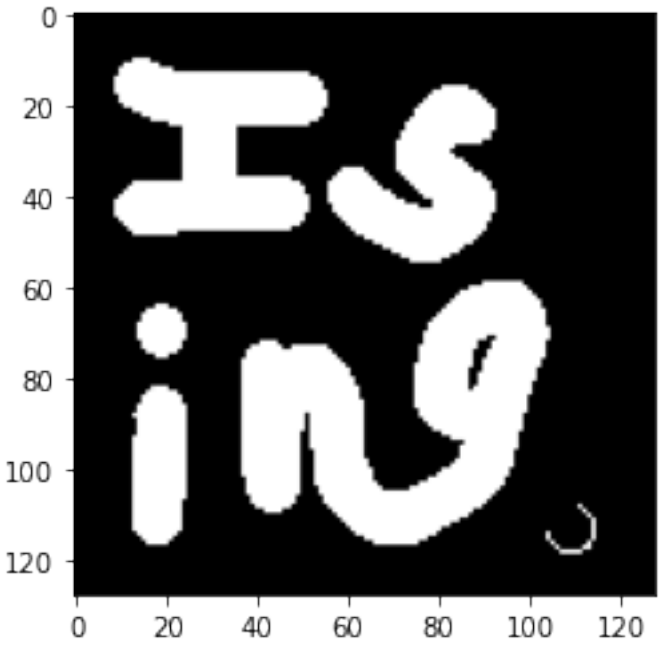
12.2 Image-edge Ising

```
edges = Image.open("ising-edges.png")
edata = np.array(edges) > 128
edges
```

```
def eenergy(img, edges, i, j):
    """Edge-modified Ising energy: 0 on edge."""
    if edges[i, j]:
        return 0
    w, h = np.shape(img)
    c = img[i, j]
    l = img[i-1, j] if i > 0   else img[w-1, j]
    r = img[i+1, j] if i < w-1 else img[0, j]
    t = img[i, j-1] if j > 0   else img[i, h-1]
```

```
        b = img[i, j+1] if j < h-1 else img[i, 0]
        return -img[i, j] * (l + r + t + b)

    def nenergy(img, edges, i, j):
        """Neighbor-modified Ising energy: 0 interactions with edges."""
        if edges[i, j]:
            return 0

        w, h = np.shape(img)
        c = img[i, j]
        l = r = t = b = 0
        if i > 0:
            l = img[i-1, j] if not edges[i-1, j] else 0
        else:
            l = img[w-1, j] if not edges[w-1, j] else 0

        if i < w - 1:
            r = img[i+1, j] if not edges[i+1, j] else 0
        else:
            r = img[0, j] if not edges[0, j] else 0

        if j > 0:
            t = img[i, j-1] if not edges[i, j-1] else 0
        else:
            t = img[i, h-1] if not edges[i, h-1] else 0

        if j < h - 1:
            b = img[i, j+1] if not edges[i, j+1] else 0
        else:
            b = img[i, 0] if not edges[i, 0] else 0

        return -img[i, j] * (l + r + t + b)

    def eisingstep(β, img, edges):
        w, h = np.shape(img)
        i = np.random.randint(w)
        j = np.random.randint(h)
        E0 = nenergy(img, edges, i, j)
        img[i, j] *= -1
        E1 = nenergy(img, edges, i, j)
        P = np.exp(-β*(E1 - E0)) if E1 > E0 else 1
        if np.random.rand() > P: # Restore old
            img[i, j] *= -1
        return img

    def frame(writer, data):
        writer.append_data((255 * ((eimg + 1) / 2)).astype('uint8'))
```

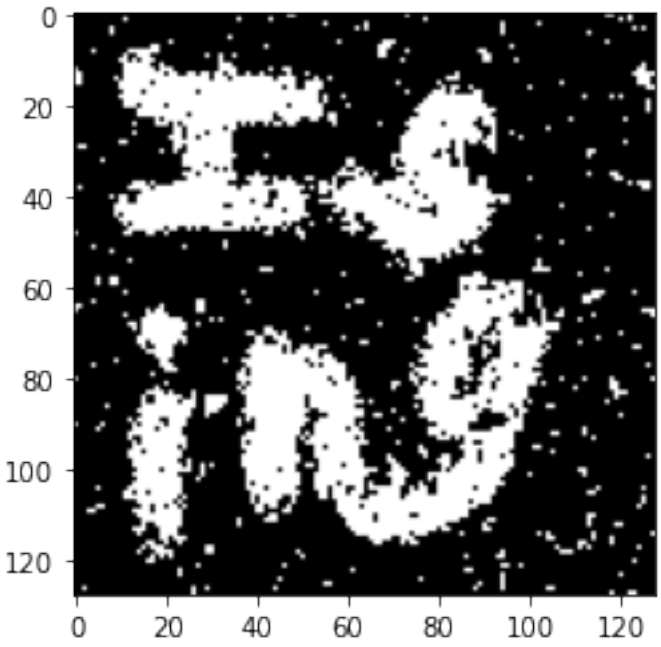
```
    img = Image.open("ising-letters.png")
    eimg = -1 + 2 * (np.array(img) / 255)
    plt.imshow(eimg);
```



movie.gif: Full neighbor Ising.

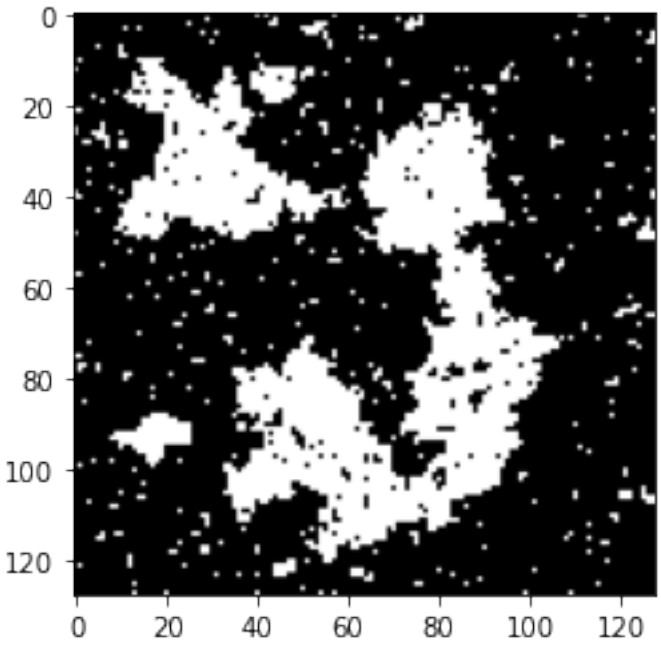
```
n = 1000000
f = IntProgress(min=0, max=1 + (n-1) // 1000) # instantiate the bar
display(f)
with imageio.get_writer('movie.gif', mode='I') as writer:
    frame(writer, eimg)
    for i in range(n):
        eisingstep(0.5 * (np.pi / 2) / np.arctan(n - i), eimg, edata)
        if i % 1000 == 0:
            f.value += 1
            frame(writer, eimg)
plt.imshow(eimg);
```

```
IntProgress(value=0, max=1000)
```



`imovie.gif`: Normal Ising.

```
n = 1000000
img = eimg
with imageio.get_writer('imovie.gif', mode='I') as writer:
    frame(writer, img)
    for i in range(n):
        isingstep(0.5 * (np.pi / 2) / np.arctan(n - i), img)
        if i % 1000 == 0:
            frame(writer, img)
plt.imshow(img);
```



12.3 Image-metric Ising

```
# def takewrap(a, i, j, xs=np.arange(-1, 2), ys=np.arange(-1, 2)):
def takewrap(a, i, j, xs=np.arange(0, 1), ys=np.arange(0, 1)):
    return np.array([x for v in a.take(xs+i, axis=0, mode='wrap')
                       for x in v.take(ys+j, mode='wrap')])
```

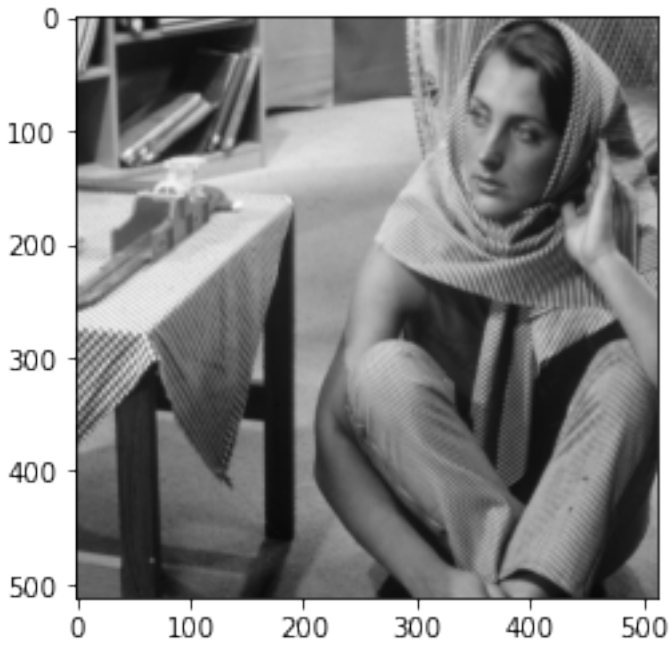
12.3.1 Unrestricted swapping motion

Swapping preserves the intensity distribution.

```
def sienergy(img, init, i, j):
    """Inversion-symmetric image energy"""
    eq = takewrap(img, i, j) == takewrap(init, i, j)
    return -np.abs(np.sum(2*eq - 1))

def ienergy(img, init, i, j):
    """Image energy based on 3x3 block deviation"""
    return np.abs(init[i, j] - img[i, j])

def swisingstep(β, img, edges):
    w, h = np.shape(img)
    i0 = np.random.randint(w)
    i1 = np.random.randint(w)
    j0 = np.random.randint(h)
    j1 = np.random.randint(h)
    E0 = ienergy(img, edges, i0, j0) + ienergy(img, edges, i1, j1)
    img[i0, j0], img[i1, j1] = img[i1, j1], img[i0, j0]
    E1 = ienergy(img, edges, i0, j0) + ienergy(img, edges, i1, j1)
```

```
    P = np.exp(-β*(E1 - E0)) if E1 > E0 else 1
    if np.random.rand() > P: # Restore old
        img[i0, j0], img[i1, j1] = img[i1, j1], img[i0, j0]
    return img

def nnisingstep(β, img, edges):
    w, h = np.shape(img)
    i0 = np.random.randint(w)
    i1 = int((i0 + np.sign(np.random.rand() - 1/2)) % w)
    j0 = np.random.randint(h)
    j1 = int((j0 + np.sign(np.random.rand() - 1/2)) % h)
    E0 = ienergy(img, edges, i0, j0) + ienergy(img, edges, i1, j1)
    img[i0, j0], img[i1, j1] = img[i1, j1], img[i0, j0]
    E1 = ienergy(img, edges, i0, j0) + ienergy(img, edges, i1, j1)
    P = np.exp(-β*(E1 - E0)) if E1 > E0 else 1
    if np.random.rand() > P: # Restore old
        img[i0, j0], img[i1, j1] = img[i1, j1], img[i0, j0]
    return img
```

```
img = Image.open("barbara.png")
eimg = -1 + 2 * (np.array(img) / 255)
initimg = eimg.copy()
plt.imshow(initimg);
```

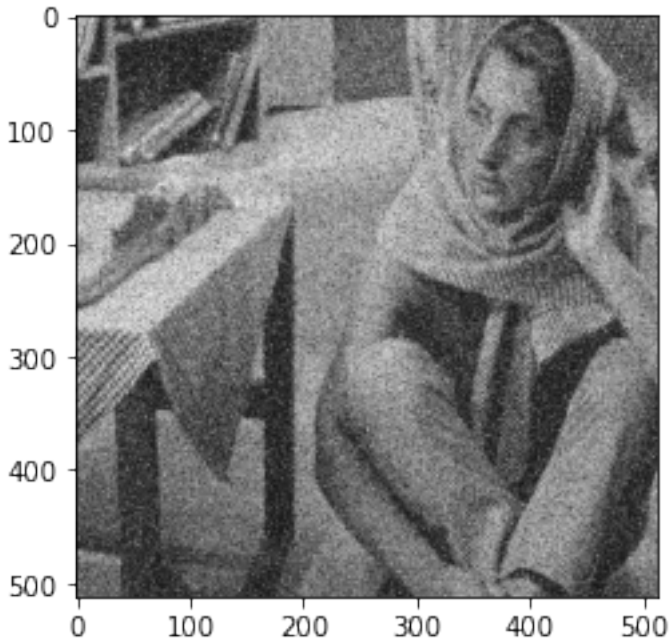


`swmovie.gif`: Image metric Ising (arbitrary swaps with `ienergy`).

```
n = 2000000
f = IntProgress(min=0, max=(1 + (n-1) // 1000)) # instantiate the bar
display(f)
with imageio.get_writer('swmovie.gif', mode='I') as writer:
```

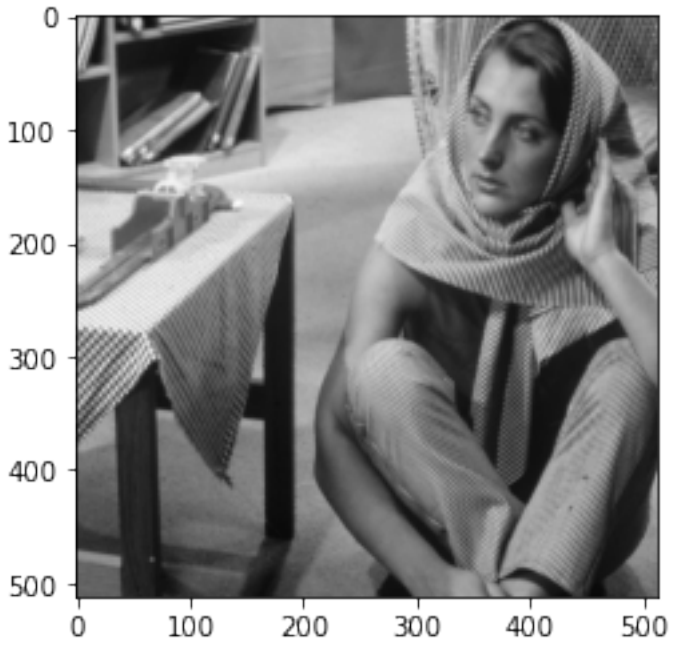
```
    frame(writer, eimg)
    for i in range(n):
        k = i/n
        swisingstep(3, eimg, initimg)
        if i % 1000 == 0:
            f.value += 1
            frame(writer, eimg)
#     for i in range(n):
#         k = i/n
#         swisingstep(4*(1 - k) + 1e-3*k, eimg, initimg)
#         if i % 1000 == 0:
#             f.value += 1
#             frame(writer, eimg)
#     for i in range(n):
#         k = i/n
#         swisingstep(1e-3*(1 - k) + 4*k, eimg, initimg)
#         if i % 1000 == 0:
#             f.value += 1
#             frame(writer, eimg)

plt.imshow(eimg);
```

```
IntProgress(value=0, max=2000)
```



12.3.2 Nearest-neighbor swapping motion

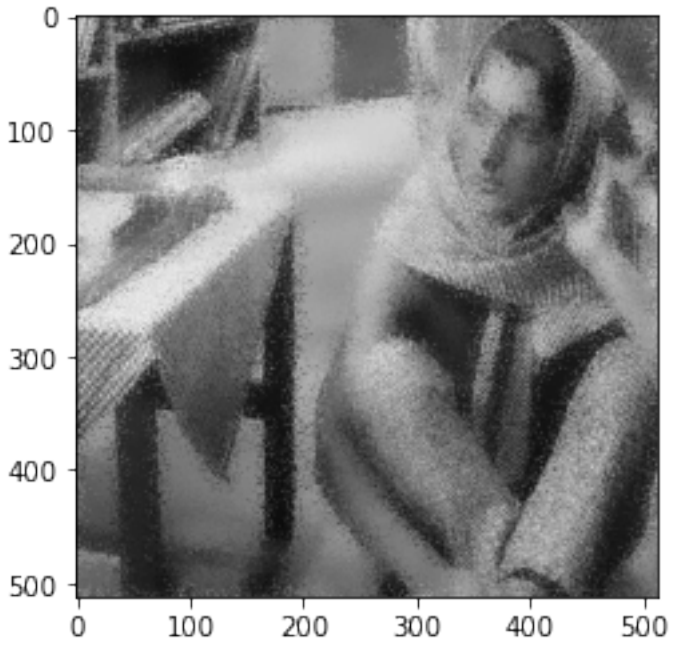
```
img = Image.open("barbara.png")
eimg = -1 + 2 * (np.array(img) / 255)
initimg = eimg.copy()
plt.imshow(initimg);
```



nnmovie.gif: Image metric Ising (neighborly swaps with ienergy).

```
n = 2000000
f = IntProgress(min=0, max=3*(1 + (n-1) // 1000)) # instantiate the bar
display(f)
with imageio.get_writer('nnmovie.gif', mode='I') as writer:
    frame(writer, eimg)
    for i in range(n):
        k = i/n
        nnisingstep(5*(1 - k) + 1e-4*k, eimg, initimg)
        if i % 1000 == 0:
            f.value += 1
            frame(writer, eimg)
    for i in range(n):
        nnisingstep(1e-4, eimg, initimg)
        if i % 1000 == 0:
            f.value += 1
            frame(writer, eimg)
    for i in range(n):
        k = i/n
        nnisingstep(1e-4*(1 - k) + 5*k, eimg, initimg)
        if i % 1000 == 0:
            f.value += 1
            frame(writer, eimg)

plt.imshow(eimg);
```

```
IntProgress(value=0, max=6000)
```



13 Statistical Mechanics of Images

June 3, 2020

Given the qualitative success of the image-metric based Ising images, we consider generalizations.

Definition 8. An *N-element image space* over metric spaces (K, d) and (P, a) is the space $\mathsf{Img} = (P \times K)^N$. A corresponding *image* is an element of Img.

The space P determines the spatial arrangement of the image, and is usually two-dimensional Euclidean space. We usually consider the subset of an image space where the P-coordinates are fixed, in a grid layout. The space K determines the qualities of an image at a point in P. This is usually a color or intensity space, and in practical applications is a machine integer like $128 \in \mathbb{Z}_{256}$.

Definition 9. An *image system* on an image space Img consists of a *ground image* $I_0 \in \mathsf{Img}$ and a *dispersion relation* $E : \mathbb{R} \to \mathbb{R}$. This defines the *energy* of an image I as

$$E(I) = \sum_{(p_0, k_0) \in I_0} \sum_{k \in \{k : (p_0, k) \in I\}} E(d(k_0, k)).$$

Example 3. For usual images in $((\mathbb{Z}_n \times \mathbb{Z}_m) \times \mathbb{Z}_{256})^N$, where $N = nm$ and the positions of I_0 and I coincide (indexed by i and j), we have

$$E(I) = \sum_{i=1}^{n} \sum_{j=1}^{m} E(d(k_0^{ij}, k^{ij})) = \sum_{i=1}^{n} \sum_{j=1}^{m} \varepsilon \left| k_0^{ij} - k^{ij} \right|^1,$$

with typical choices of E and d.

In the binary case ($K = \mathbb{Z}_2$), we have N independent two-level systems.

Example 4 (Grayscale images). Consider a pixel of a ground grayscale image, with integer value $k_0 \in 0, \ldots, K-1$ for even K. There are then

$$2g = 2\begin{cases} k_0, & k_0 < K/2 \\ K - k_0 - 1, & \text{else} \end{cases}$$

energy values that occur twice, and $K - 2g$ energy values that occur once (like $|x|$ on an interval like $[-3, 8]$). Thus the partition function for this single pixel is

$$\begin{aligned} Z_g &= \sum_{k=-g}^{K-g-1} e^{-\beta\varepsilon|k|} = 1 + \sum_{k=1}^{g} e^{-\beta\varepsilon k} + \sum_{k=1}^{K-g-1} e^{-\beta\varepsilon k} \\ &= 1 + \frac{e^{-\beta g\varepsilon}\left(e^{\beta g\varepsilon} - 1\right)}{e^{\beta\varepsilon} - 1} + \frac{e^{-\beta(K-g-1)\varepsilon}\left(e^{\beta(K-g-1)\varepsilon} - 1\right)}{e^{\beta\varepsilon} - 1} \end{aligned}$$

and the partition function for the whole image is

$$Z = \prod_{g=0}^{-1+K/2} Z_g^{NP(g)},$$

where $NP(g)$ is the number of pixels in the ground image with the given g-value. We then see that

$$\ln Z = \sum_{g=0}^{-1+K/2} NP(g) \ln Z_g = N \left\langle \ln Z_g \right\rangle_G,$$

where G is the random variable that takes the value g with probability $P(g)$. It then follows that $\langle E/N \rangle = \left\langle E_g \right\rangle_G$ and $S/N = \left\langle S_g \right\rangle_G$ as usual for extensive variables.

14 Thermodynamic quantities for images from a microscopic model

June 4, 2020

Since we are thinking of images as statistical entities, what is the corresponding microscopic model? Given such a model, what quantities do we consider in thermal equilibrium, and how can we understand different ensembles?

14.1 Quantum filled-site model (FSM)

Definition 10 (FSM). We define a lattice model corresponding to a *ground image* I_0 as follows. Each pixel with value $k_0 \in K \subseteq \mathbb{Z}$ in the image corresponds to a *site*, which is a discrete system with K *levels*. The energy of level k is $E_{k_0}(k) = \varepsilon|k - k_0|^r$.[1] We

[1] If we want to consider colors, then K is a metric space and we replace $k - k_0$ with the metric.

usually have $r = 1$ or 2. We suppose that the levels are filled by fermions that interact according to the Hamiltonian

$$\mathsf{H} = \sum_{k \in K} \sum_{i} V \mathsf{c}_{ik}^\dagger \mathsf{c}_{ik} - \sum_{\ell \in \mathcal{N}_k} \sum_{j \in \mathcal{N}_i} t_\ell \mathsf{c}_{ik}^\dagger \mathsf{c}_{j\ell}.$$

For $K \subseteq \mathbb{Z}$, $\mathcal{N}_k = k + \{-1, 0, 1\}$.

14.2 Observables and thermodynamic state variables

Several observables of the FSM are of interest:

- **Pixel colors**. The occupations n_k of different levels at a *single* site induce a distribution on K. In equilibrium, the mean level

  $$\langle k \rangle \equiv \frac{\sum_{k \in K} k n_k}{\sum_{k \in K} n_k}$$

  should be near k_0, since the energy of a level is symmetric about k_0. As the temperature increases, so will the variance of the mean level. On the flip side, does *varying* k_0 for many pixels quasistatically (changing the ground image) do work on the system? Yes, but is this consistent with what we expect?

- **Color distribution**. The net occupations m_k of different levels across *all* sites induce a distribution on K. In the special case of gray images (so levels are intensity), the entropy of the induced random variable is intensity entropy that we have studied previously. This is distribution is stationary when different levels cannot interact, but is it so at finite temperature?

- **Opacity**. The net occupancy of a *site* could be connected to its opacity. In equilibrium, this should be similar across all sites. Then regions with *no* particles during nonequilibrium processes make sense. The picture of a gas with fluctuating density that emits light comes to mind. When at maximum opacity, the gas in that region cannot be compressed further, and cannot accept more particles. The canonical density properties of a photon gas (like energy density) might be a good reason to choose the particles to be bosons.

- **Number of pixels**. We could vary the total number of pixels different ways. One way is to have a continuous ground image, and choose different grid discretizations. Another way is to have a large ground image grid and vary the zoom level. It would be sensible to combine these sorts of transformations with pixel color transformations, since they include translation and rotation as special cases. This seems most most similar to varying the volume of a gas. Including opacity makes fast adiabatic piston motion volume changes like $V \mapsto 2V$ make sense.

- **Number of particles**. Depending on if we allow interactions between levels, it may be appropriate to consider chemical potentials. Either for all particles, or for each color. Could this be conjugate to opacity?

- **The usual**. Given that quasistatic transformation of the other quantities does work the way we expect, we can consider the usual response variables like heat capacities and compressibilities. There is also the thermodynamic entropy.

15 Progress summary (from beginning)

June 6, 2020 Over the last two weeks, I calculated some metrics on images and did some basic simulations. The most important metric was the intensity entropy, which is used as the "entropy" of an image in the maximum entropy method (MEM) of image reconstruction used by astronomers. This was calculated for a whole image and locally in different regions of an image. On the topic of scaling, fractal dimensions were explored. The box-counting dimension was computed for different images, and the information dimension of the intensity distribution was considered. I also did readings on probability theory and machine learning, since the usual frequentist approach that experimental physicists take is not applicable. Variants of Ising models were simulated for images, which led to the postulation of a microscopic model for varying images (the FSM), which is similar to a Hubbard model. The implications of this approach remain to be explored.

16 Progress summary

June 12, 2020 This week, I investigated different methods for obtaining thermodynamic quantities from simulations. The issue is that quantities like entropy and the Helmholtz free energy depend on global properties of the phase space (the probability or density of a microstate), and thus cannot be constructed as cumulants of microstates during a simulation. A related issue is the improbability of "tunneling" across energy barriers when taking the usual temperature-weighted steps.

These considerations motivate histogram-based methods, like the Wang-Landau algorithm that I implemented. Instead of operating in the canonical ensemble, we take a biased random walk on energies so that the result is a flat histogram of visited energies. The density of states from this process may then be used to compute a canonical partition function. Modifications where we keep a joint density of states with respect to another variable make other ensembles accessible.

Further progress with the Wang-Landau algorithm was slowed by a discrepancy in the steps needed between Wang and Landau's results and mine for the 32 by 32 Ising ferromagnet. Their original paper claims a 0.035 % average error in the density of states after only 700 000 total spin flips.[2] This is far fewer than the spin flips I needed, and another paper that looks at the scaling of the tunneling time (spin flips to go from ground to anti-ground state) might corroborate this.[3] Their tunneling times are all well above 1 720 000 (an eighth of their τ_{exact}) for the same simulation. Success in the Wang-Landau algorithm requires visiting all energies many times, so execution takes several tunneling times. We are still trying to resolve this discrepancy, as well

[2] 10.1103/PhysRevLett.86.2050
[3] 10.1103/PhysRevLett.92.097201

as other vague details from the original paper, like the possible choice of energy bins for continuous systems and unspecified edge behavior for energy intervals.

I have looked at other histogram methods like WHAM, as well as parallel tempering (replica exchange MCMC). Both parallel tempering and the Wang-Landau algorithm are attractive because they are easily parallelizable. I have also come back to considering the MAXENT/MEM image reconstruction technique and the issues of entropy and feature representation again.

17 Description of MaxEnt

June 13, 2020 I describe the basic idea of the MaxEnt reconstruction technique, and implement the most naïve approach. This makes it clear that a better optimization algorithm is needed. The idea here is less to make it work than to get an idea of how it works, and then use a better software package later on if we do end up using MaxEnt.

18 Maximum-entropy reconstruction

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal, misc
```

Given a measurement D of a system I_0, we wish to reconstruct I_0 from D. We take the Bayesian approach and find the I that maximizes $P(I|D) \propto P(D|I)P(I)$, where the likelihood is related to the error in D due to I and the prior is $P(I) = \exp(\lambda S(I))$, where S is some notion of the entropy of an image. If $D = T(I_0)$, then $P(D|I) = \exp(-E(T(I), D))$, for some metric E on D's. From the form of the objective $\ln P(D|I) + \ln(I) = -E(T(I), D) + \lambda S(I)$, we see that the entropy acts as a regularizer.

The astronomers have both I and D as intensity lists $\mathbb{Z}_N \to \mathbb{R}_{\geq 0}$, where

$$S(I) = \sum_i I_i \ln I_i$$

and

$$E(D', D) = \sum_i \exp\left(-\frac{(D'_i - D_i)^2}{2\sigma_i}\right)$$

(with empirical σ_i). The transformation T is convolution with the point spread function of the telescope.

To perform the optimization, we need not only the functions S and E, but also a procedure to modify candidate images.

```
def maxent_objective(D, I, λ): # For minimization
    return D.E(I.transform()) - λ*I.S
```

```
# Greedy for now, but can be something like simulated annealing
def maxent(D, I, λ = 1, N = 1_000_000, ε = 1e-8):
    f0 = np.inf
    f = maxent_objective(D, I, λ)
```

```
    i = 0
    while ε < f0 - f and i < N:
        I.propose()
        fv = maxent_objective(D, I, λ)
        if fv < f: # Greedy
            f0, f = f, fv
            I.accept()
        i += 1
        if i % (N // 20) == 0:
            print("Maxent: {} / {}".format(i, N))

    print("Maxent: i: {} / {}, Δf: {}".format(i, N, f0 - f))
    return I, i, f
```

18.1 Example: 1D Point from Gaussian

```
class Gaussian:
    def __init__(self, μ=0, σ=1):
        self.μ = μ
        self.σ = σ
    def E(self, Gv):
        return (self.μ - Gv.μ)**2 + (self.σ - Gv.σ)**2

class Point:
    def __init__(self, x=0):
        self.x = x
        self.dx = 0
        self.S = self.entropy()
    def propose(self):
        self.dx = np.random.rand() - 0.5
        self.S = self.entropy()
    def accept(self):
        self.x += self.dx
        self.dx = 0 # Idempotence
    def entropy(self):
        return -(self.x - 10)**2 # Opposite from true max
    def transform(self):
        return Gaussian(μ = self.x + self.dx)
```

```
D = Gaussian(0, 1)
I = Point(9)
I0, _, _ = maxent(D, I, λ = 1, N = 1_000_000, ε = 1e-4)
I0.x
```

```
Maxent: i: 1000000 / 1000000, Δf: 0.011299906795997572
```

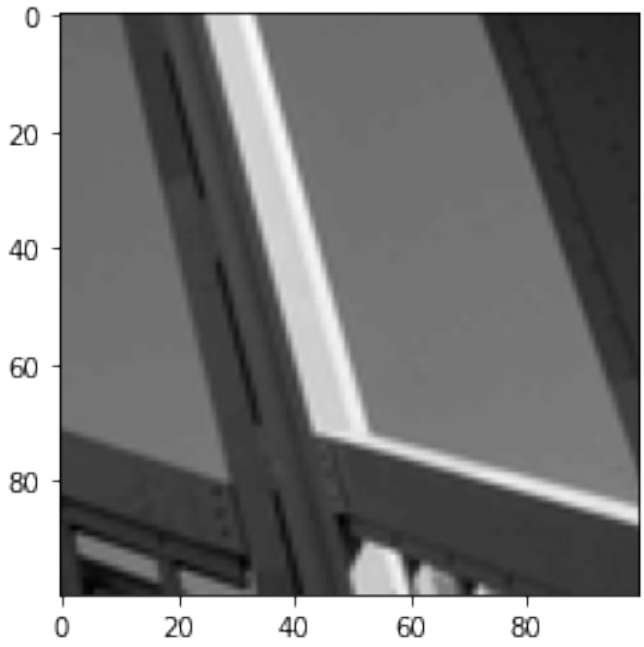
```
4.268883578482481
```

Results are sort of near 5. The optimization is terrible, but you get the idea: the entropy shifts the best point away from zero.

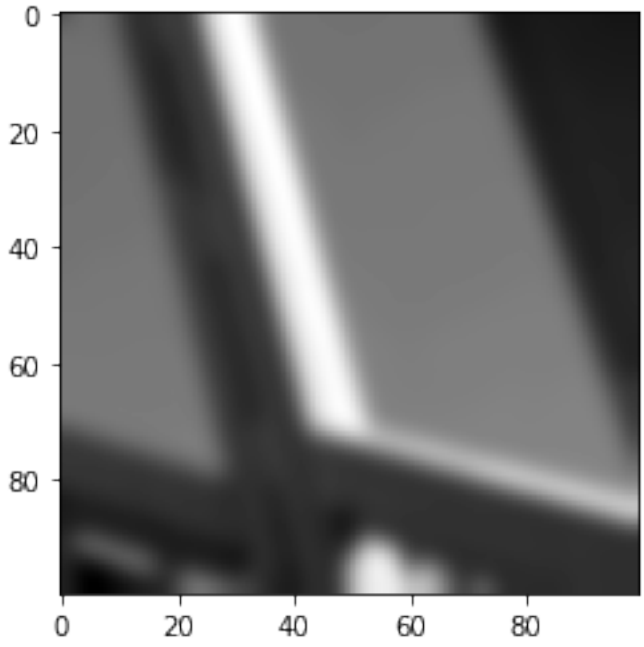
18.2 Example: Image from PSF convolution (measurement)

```
class DImage:
    def __init__(self, I):
        self.I = I
    def E(self, Dv):
        return np.sum((self.I - Dv.I)**2)

class IImage:
    def __init__(self, I):
        self.I = I
        self.w, self.h = np.shape(I)
        self.i, self.j = 0, 0
        self.I0 = 0
        self.Iv = 0
        n = int(np.sqrt(self.w * self.h))
        self.G = signal.windows.gaussian(n // 10, n // 50)
        self.S = self.entropy()
    def propose(self):
        self.i, self.j = np.random.randint(self.w), np.random.randint(self.h)
        I0 = self.I[self.i, self.j]
        self.I0 = I0
        self.Iv = np.random.randint(256)
        self.S = self.entropy()
    def accept(self):
        self.I[self.i, self.j] = self.Iv
    def entropy(self):
        return -np.sum(np.log(self.I + 1)) + self.I0*np.log(self.I0 + 1) - self.Iv*np.log(self.Iv + 1)
    def transform(self):
        Iv = self.I.copy()
        Iv[self.i, self.j] = self.Iv
        return DImage(signal.sepfir2d(Iv, self.G, self.G))
```

```
I = IImage(misc.ascent()[250:350,250:350])
plt.imshow(I.I, cmap='gray');
```




```
plt.imshow(I.transform().I, cmap='gray');
```

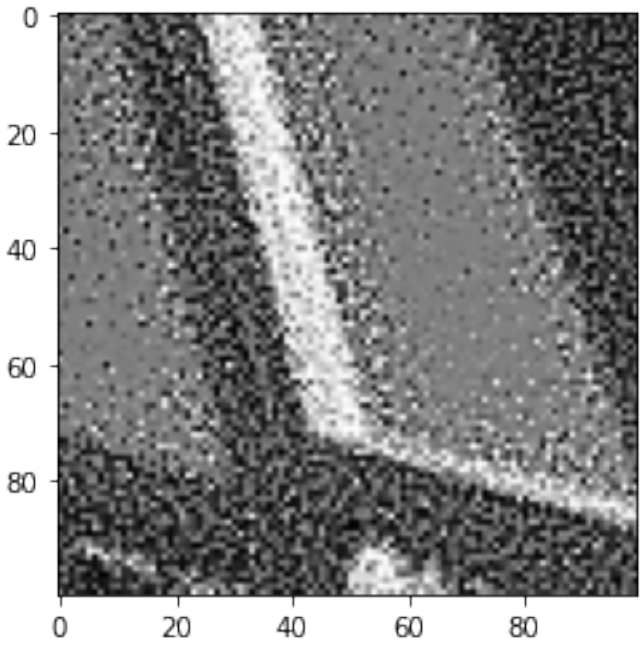



```
I = IImage(misc.ascent()[250:350,250:350])
```

```
Iguess = IImage(128 * np.ones((I.w, I.h), dtype=int))
I0, _, _ = maxent(I.transform(), Iguess, λ = 1e-9, N = 1_000_000, ε = 1e-20) # Just do the max iterations
```

```
Maxent: 50000 / 1000000
Maxent: 100000 / 1000000
Maxent: 150000 / 1000000
Maxent: 200000 / 1000000
Maxent: 250000 / 1000000
Maxent: 300000 / 1000000
Maxent: 350000 / 1000000
Maxent: 400000 / 1000000
Maxent: 450000 / 1000000
Maxent: 500000 / 1000000
Maxent: 550000 / 1000000
Maxent: 600000 / 1000000
Maxent: 650000 / 1000000
Maxent: 700000 / 1000000
Maxent: 750000 / 1000000
Maxent: 800000 / 1000000
Maxent: 850000 / 1000000
Maxent: 900000 / 1000000
Maxent: 950000 / 1000000
Maxent: 1000000 / 1000000
Maxent: i: 1000000 / 1000000, Δf: 4.847227362683043
```

```
plt.imshow(I0.I, cmap='gray');
```



19 “Greedy” painting-like pictures

June 14, 2020

We noticed that swapping neighboring pixels in the previous simulations produced images that looked somewhat like paintings. There are other ways to make images look like paintings, and this is one of them. We randomly draw a shape of fixed size on the image. The color of the shape is the mean color of the image in the rectangle where the shape is drawn. This is done many times, and the size of the shapes gradually decreases. I call this “greedy” because choosing the mean color is the locally optimal color. This method has the advantage of being similar to the process of painting, which is why it is presented instead of the usual “oilify” filters in image processing software. An improvement would try to draw shapes that reflect the shapes of the subject matter, and perhaps add other texture. These kinds of considerations might require more global strategies.

20 Greedy Cubism

Draw an image by greedily drawing cubes.

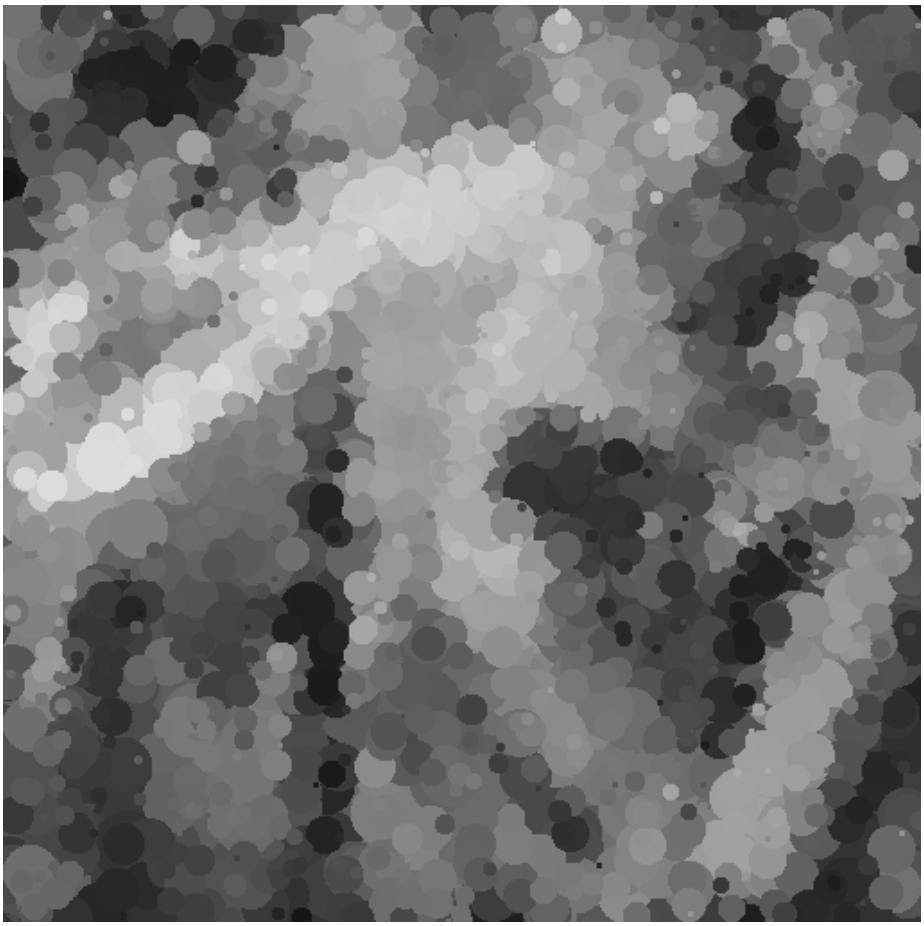
```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image, ImageOps, ImageDraw
from scipy import signal, misc
```

```
img = Image.open("barbara.png")
```

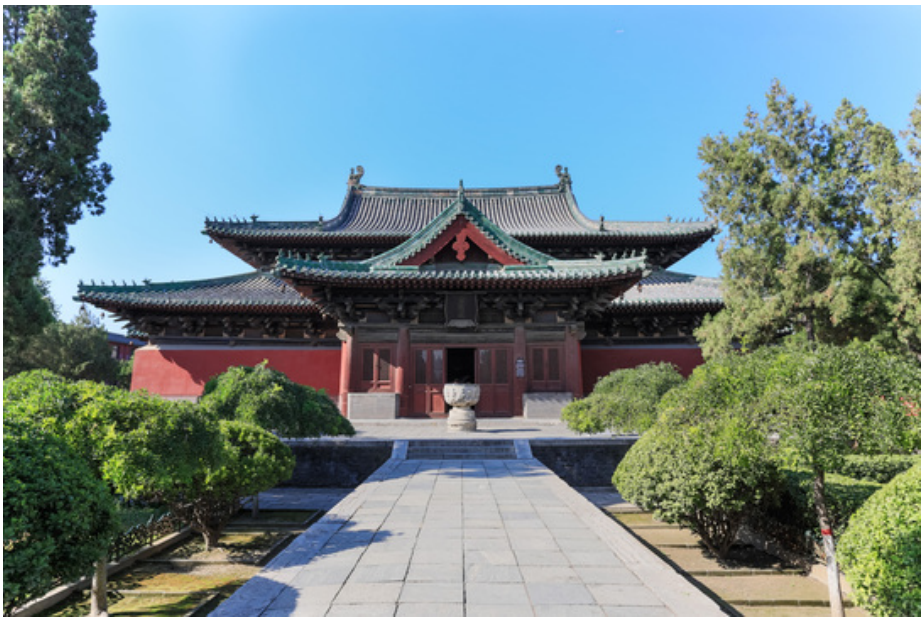
```
def greedy_cubes(img, N):
    xs, ys = img.size
    art = Image.new(img.mode, (xs, ys))
    draw = ImageDraw.Draw(art)
    rmax = int(np.sqrt(xs*ys) / 10)
    r = rmax
    ε = 10
    for i in range(N):
        x = np.random.randint(xs)
        y = np.random.randint(ys)
        [np.mean(c) for c in cimg.split()]
        r = int(rmax * (1 - (i/(N+1))**2)) + 1
        box = [x - r, y - r, x + r, y + r]
        color = tuple(int(np.round(np.mean(c))) for c in img.crop(box=box).split())
        draw.ellipse(box, fill=color)
    return art
```

```
art = greedy_cubes(img, 10000)
```

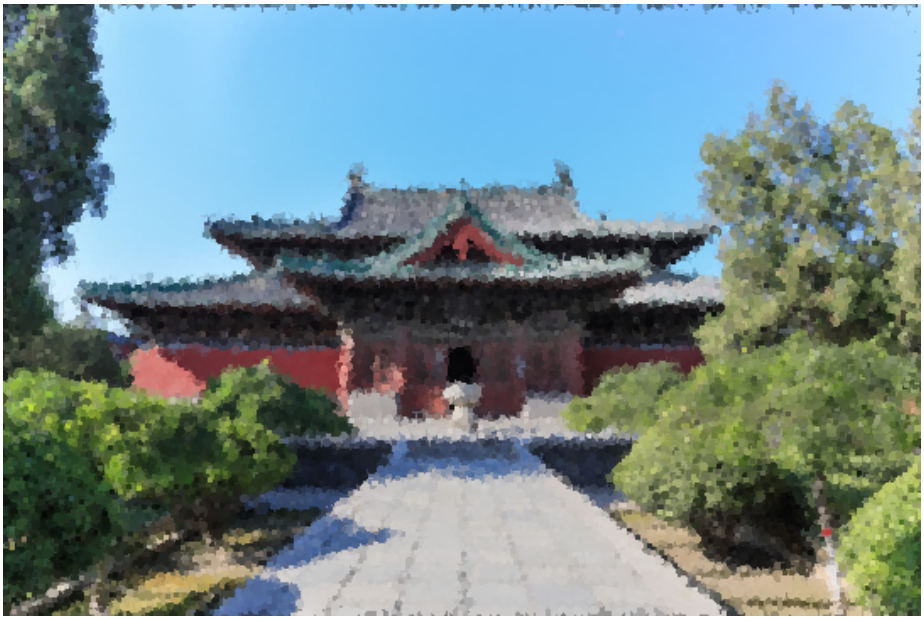
```
art
```



```
cimg = Image.open('test.jpg')
cimg
```



```
art = greedy_cubes(cimg, 1000000)
art
```



21 Exact density of states for gray and BW images

June 16, 2020

In doing the future statistical simulations of images, we would like to know the density of states $g(E)$ in our site-based model. This is the number of ways to choose the N pixel values m_n from $0, \ldots, M$ so that

$$E = \sum_{n=1}^{N} \left| m_n - m_n^* \right|,$$

where $0 \leq E \leq MN$ and m_n^* is the value of the ground image at pixel n. While the solution to follow works for arbitrary values of m_n^*, the results are simplest (no multinomial coefficients) if we consider $m_n^* = 0$ or M or $m_n^* = M/2$. That is, the image is all black or white (BW) or gray. Then we have

$$E = \sum_{n=1}^{N} g' m_n$$

with $m_n = 0, \ldots, M$ and $g' = 1$ (BW) or $m_n = 0, \ldots, M/2$ and $g' = 2$ (gray).[4] In that case, we'll consider BW for concreteness.

We encode the energy as an exponent, so that $g(E)$ is the coefficient of x^E in the polynomial

$$p(x) \equiv \left(x^0 + x^1 + \cdots + x^M \right)^N,$$

since each inner sum represents different assignments of a pixel value and the products will produce terms that represent all different combinations of values. We then expand to find

$$\begin{aligned} p(x) &= \left(\frac{1 - x^{M+1}}{1 - x} \right)^N \\ &= \sum_{k=0}^{N} \sum_{j=0}^{\infty} (-1)^k \binom{N}{k} x^{(M+1)k} \cdot (-1)^j \binom{-N}{j} x^j \\ &= \sum_{k=0}^{N} \sum_{j=0}^{\infty} (-1)^k \binom{N}{k} \binom{N+j-1}{j} x^{(M+1)k+j}. \end{aligned}$$

The value of j in the inner sum for x to have power E is $j = E - k(M+1)$, so the coefficient of x^E in $p(x)$ is

$$g(E) = \sum_{k=0}^{N} (-1)^k \binom{N}{k} \binom{N + E - k(M+1) - 1}{E - k(M+1)}.$$

(Due to the stucture of the binomial coefficients, we may let the summation extend over all $k \in \mathbb{Z}$ for further manipulation.)

[4] This gives $m = 0$ degeneracy 2, which is not strictly correct but matters little.

22 Comparison to Wang and Landau's results

June 18, 2020

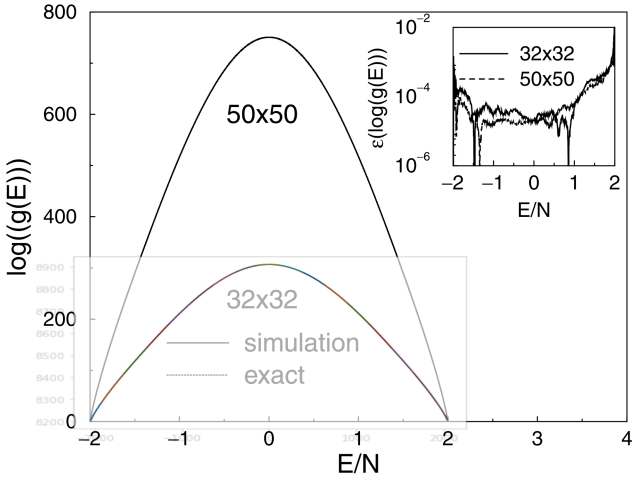


Figure 1: Our results agree with Wang and Landau's results, though it took more iterations than they reported to reproduce.

23 Progress summary

June 19, 2020 This week, I further prepared to study the thermodynamics of our image systems. I derived the exact density of states for the all-black image, and the same technique (generating functions) could be used with computer enumeration of integer partitions to obtain the density of states for an arbitrary image system. I also spent more time improving my implementation of the Wang-Landau algorithm. It is now able to simulate arbitrary energy bins and divide different energy intervals across multiple CPU cores and combine them back together. The python code was made type-stable so that functions and the simulation state can be JITed by a LLVM-based compiler (Numba). These two improvements increased the speed of the simulations by more than a factor of 10. With some more tweaks and code to automate the preparation of parallel kernels and manage results, we should be able to easily perform these simulations for many parameter values with high statistics.

24 Simulations for canonical ensemble averages

June 22, 2020 We implement some simulation software for the Wang-Landau algorithm.

24.1 Organized parallel simulations

We need some utility functions that

- Manage passing simulation parameters to parallel processes
- Run the parallel processes with correct input and distinct random states
- Save parameters and results to files automatically (without overwriting previous results)
- Log results while running simulations
- Join parallel results back together

Since the simulation systems (Numba `jitclass` objects) are not picklable (Python's serialization used to communicate between processes by `multiprocessing`), the relevant parameters to reconstruct a system are passed between processes. We require that all systems be accessible from the `systems` module, so that we may pass and save the class name as a way of accessing the specification of the system.

```
import numpy as np
from multiprocessing import Pool
from scipy.signal import windows
import sys
import time
import os, struct # for `urandom`
import pprint # for parameters
import tempfile
import h5py, hickle
```

```
if 'src' not in sys.path: sys.path.append('src')
import systems
```

```
def make_params(system):
    return {system.__class__.__name__:
            {k: v for k, v in zip(system.state_names(), system.state())}}

def make_system(system_params, system_prep = lambda x:x):
    return system_prep([getattr(systems, cls)(**state)
                        for cls, state in system_params.items()][0])

def make_psystems(params, psystem_prep): #:: params -> (system -> [system]) -> [params]
    log = params.get('log', False)
    if log:
        print('Finding parallel bin systems ... ', end='', flush=True)
    psystems = psystem_prep(make_system(params['system']), **params['parallel'])
    if log:
        print('done.')
    return [(make_params(s), *r) for s, *r in psystems]

def urandom_reseed():
    """Reseeds numpy's RNG from `urandom` and returns the seed."""
    seed = struct.unpack('I', os.urandom(4))[0]
    np.random.seed(seed)
    return seed

def worker(simulation, psystem, params):
    log = params.get('log', False)
```

```
    urandom_reseed()
    psystem_params, *args = psystem
    system = make_system(psystem_params)
    if log:
        print('(', end='', flush=True)
    # Individual simulation output is too much when running parallel simulations.
    params['simulation'].update({'log': False})
    results = simulation(system, *args, **params['simulation'])
    if log:
        print(')', end='', flush=True)
    return results

def show_params(params):
    print('Run parameters')
    print('--------------')
    pprint.pp(params, sort_dicts=False)
    print()

def save_results(results, params, log=False, prefix='simulation-', dir='data'):
    with tempfile.NamedTemporaryFile( # Note: dir shadows dir()
        mode='wb', prefix=prefix, suffix='.h5', dir=dir, delete=False) as f:
        with h5py.File(f, 'w') as hkl:
            if log:
                print('Writing results ... ', end='', flush=True)
            hickle.dump({
                'parameters': params,
                'results': results
            }, hkl)
            relpath = os.path.relpath(f.name)
            if log:
                print('done: ', relpath)
    return relpath

def run(params, simulation, system_prep,
        psystem_prep = lambda x:x, result_wrapper = lambda x:x, **kwargs):
    params.update(kwargs)
    parallel = 'parallel' in params
    log = params.get('log', False)
    if log:
        show_params(params)

    if parallel:
        psystems = make_psystems(params, psystem_prep)
    else:
        psystem = make_system(params['system'], system_prep)

    if log:
        if parallel:
            print('Running || ', end='', flush=True)
        else:
            print('Running ...')
        start_time = time.time()

    if parallel:
        with Pool() as pool:
            results = pool.starmap(worker, ((simulation, args, params) for args in psystems))
        results = [result_wrapper(r) for r in results]
```

```
    else:
        results = result_wrapper(simulation(*psystem, **params['simulation'], **kwargs))

    if log:
        seconds = int(time.time() - start_time)
        if parallel:
            print(' || done in', seconds, 'seconds.')
        else:
            print('... done in', seconds, 'seconds.')

    # Save single-shot results in a singleton list so that we can analyze parallel and
    # single results the same way.
    rdict = {'results': results if parallel else [results]}
    save_params = params.pop('save', False)
    if save_params:
        relpath = save_results(results, params, log, **save_params)
        rdict.update({'file': relpath})

    return rdict
```

We can choose overlapping bins for the parallel processes to negate boundary effects.

```
def extend_bin(bins, i, k = 0.05):
    if len(bins) ≤ 2: # There is only one bin
        return bins
    k = max(0, min(1, k))
    return (bins[i] - (k*(bins[i] - bins[i-1]) if 0 < i else 0),
            bins[i+1] + (k*(bins[i+2] - bins[i+1]) if i < len(bins) - 2 else 0))
```

Often parallel results are the value of a real function on some grid or list of bins. Given that many of these pieces may overlap, we must combine them back together into a full solution. This requires first transforming the results so that they are comparable, and then performing the combination. The most common case is repetition of the same real-valued experiment. No transformation is required, and we simply average all the results. Even better, we may assign the values within each piece a varying credence from 0 to 1 and perform weighted sums.

We must join results that are only known up to an additive constant. We must then assign an offset to each solution, where one of the offsets may be taken to be zero without loss of generality. The algorithm below implements the offsets that minimize the weighted mean-squared error of the overlap regions

$$\varepsilon = \sum_{i<j,\,k} ((y_{ik} + c_i) - (y_{jk} + c_j))^2 w_{ik} w_{jk},$$

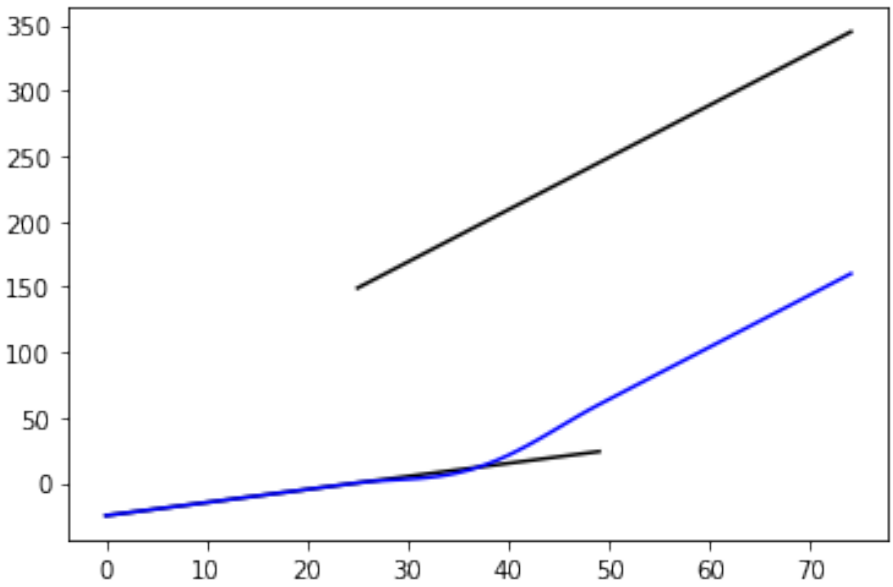
where i and j index different (overlapping) results, k indexes the full grid, and w_{ik} is the nonnegative weight assigned to index k of result i. Since we usually have results defined on a subset of the whole grid, we assign weights to the subset and set the other weights to be zero, so that the missing values y_{ik} outside the known subset are irrelevant.

```
def join_results(xs, ys, wf = windows.hann):
    xf = np.array(sorted(set().union(*xs)))
```

```
    xi = [np.intersect1d(xf, x, return_indices=True)[1] for x in xs]

    n, m = len(xf), len(xs)
    ws = np.zeros((m, n))
    wc = np.zeros((m, n))
    for i in range(m):
        l = len(xs[i])
        ws[i, xi[i]] = wf(l)
        wc[i, xi[i]] = np.ones(l)
    unweighted = np.sum(wc, 0) <= 1

    Δys = np.zeros(m)
    for i in range(m):
        Σc = Σw = 0
        for j in range(i):
            a = Δys[j] * np.ones(n)
            a[xi[j]] += ys[j]
            a[xi[i]] -= ys[i]
            w = ws[i,:] * ws[j,:]
            Σc += np.dot(a, w)
            Σw += np.sum(w)
        Δys[i] = Σc / Σw if i > 0 else 0

    yf = np.zeros(n)
    for i in range(m):
        w = ws[i, xi[i]]
        # The weights are meaningful only as relative weights at overlap points.
        # We must avoid division by zero at no-overlap points with weight zero.
        # Note that overlap points with all weights zero will be an issue, as
        # the weights in that situation are meaningless.
        w[(w == 0) & unweighted[xi[i]]] = 1
        yf[xi[i]] += (ys[i] + Δys[i]) * w
        ws[i, xi[i]] = w
    Σws = np.sum(ws, 0)
    yf /= Σws

    return xf, yf, Δys
```

Demonstration of joining overlapping results.

```
if __name__ == '__main__':
    from matplotlib import pyplot as plt

    testn = 50
    xs = [np.arange(testn), np.arange(testn // 2, testn + testn // 2)]
    ys = [np.arange(testn) - testn // 2, 4*np.arange(testn // 2, testn + testn // 2) + (testn - 1)]
    axs, ays, _ = join_results(xs, ys)

    for x, y in zip(xs, ys):
        plt.plot(x, y, 'black')
    plt.plot(axs, ays, 'blue');
```



24.2 Systems for organized simulation

```
import sys
if 'src' not in sys.path: sys.path.append('src')
```

List of imports for all relevant systems.

```
from statistical_image import *
from ising_model import *
```

24.2.1 Specification

Extensions

- Encode this specification and the requirements of simulations as abstract base classes?
- Consider changing to `numba.typed.Dict` in the future if the API is guaranteed to be stable.

A system for simulation is a Numba `jitclass` that implements `state`, `state_names`, and `copy` functions. Given an instance `s` of a system class `System`, these functions should satisfy

```
id_systems = [
    s,
    s.copy(), # deep
    System(*s.state()),
    System(**{k: v for k, v in zip(s.state_names(), s.state())})
]
1 == len({t.state() for t in id_systems}) == len({t.state_names() for t in id_systems})
```

By default, we have

```
class System: # ...
    def copy(self):
        return self.__class__(*self.state())
```

In addition, different simulations may require more methods to be implemented.

24.2.2 Wang-Landau

A Wang-Landau simulation requires a `System` to have the variables `E`, `Ev`, and `sweep_steps`, and to implement the methods `energy_bins`, `energy`, `propose`, and `accept`.

24.2.3 System: The 2D Ising model

```
import numpy as np
from numba.experimental import jitclass
from numba import int64
integer = int64
```

```
__all__ = ['IsingModel']
@jitclass([
    ('spins',  integer[:,:]),
    ('L',  integer),
    ('sweep_steps', integer),
    ('E',  integer),
    ('Ev', integer),
    ('dE', integer),
    ('i',  integer),
    ('j',  integer)
])
class IsingModel:
    def __init__(self, spins):
        shape = np.shape(spins)
        if len(shape) != 2 or shape[0] != shape[1]:
            raise ValueError('IsingModel spin array is not a square.')
        self.spins = spins
        self.L = shape[0]
        self.sweep_steps = shape[0]**2
        self.E = self.energy()
        self.Ev = self.E
        self.dE = 0
        self.i = 0
        self.j = 0
    def state(self):
        return (self.spins.copy(),)
    def state_names(self):
        return ('spins',)
    def copy(self):
        return IsingModel(*self.state())
    def energy_bins(self):
        Ex = 2 * self.L**2
        ΔE = 4
        Es = np.arange(-Ex, Ex + ΔE + 1, ΔE)
        # Penultimate indices are not attainable energies
        return np.delete(Es, [1, -3])
```

```
    def neighbors(self, i, j):
        return np.array([
            self.spins[i-1, j],
            self.spins[(i+1) % self.L, j],
            self.spins[i, j-1],
            self.spins[i, (j+1) % self.L],
        ])
    def energy(self):
        E = 0
        for i in range(self.L):
            for j in range(self.L):
                E -= np.sum(self.spins[i, j] * self.neighbors(i, j))
        return E // 2
    def propose(self):
        i, j = np.random.randint(self.L), np.random.randint(self.L)
        self.i, self.j = i, j
        dE = 2 * np.sum(self.spins[i, j] * self.neighbors(i, j))
        self.dE = dE
        self.Ev = self.E + dE
    def accept(self):
        self.spins[self.i, self.j] *= -1
        self.E = self.Ev
```

24.2.4 System: Statistical Image

```
import numpy as np
from scipy import special
from numba.experimental import jitclass
from numba import int64
integer = int64
```

```
__all__ = ['StatisticalImage']
@jitclass([
    ('I0', integer[:]),
    ('I',  integer[:]),
    ('N',  integer),
    ('M',  integer),
    ('sweep_steps', integer),
    ('E',  integer),
    ('Ev', integer),
    ('dE', integer),
    ('dx', integer),
    ('i',  integer)
])
class StatisticalImage:
    def __init__(self, I0, I, M):
        if len(I0) != len(I):
            raise ValueError('Ground image I0 and current image I should have the same length.')
        if M < 0:
            raise ValueError('Maximum site value must be nonnegative.')
        self.I0 = I0
        self.I  = I
        self.N  = len(I0)
        self.M  = M
        self.sweep_steps = len(I0)
        self.E  = self.energy()
        self.Ev = self.E
```

```
        self.dE = 0
        self.dx = 0
        self.i  = 0
    def state(self):
        return self.I0.copy(), self.I.copy(), self.M
    def state_names(self):
        return 'I0', 'I', 'M'
    def copy(self):
        return StatisticalImage(*self.state())
    def energy_bins(self):
        E0 = 0
        Ef = np.sum(np.maximum(self.I0, self.M - self.I0))
        ΔE = 1
        return np.arange(E0, Ef + ΔE + 1, ΔE)
    def energy(self):
        return np.sum(np.abs(self.I - self.I0))
    def propose(self):
        i = np.random.randint(self.N)
        self.i = i
        x0 = self.I0[i]
        x = self.I[i]
        r = np.random.randint(2)
        if x == 0:
            dx = r
        elif x == self.M:
            dx = -r
        else:
            dx = 2*r - 1
        dE = np.abs(dx) if x0 == x else (dx if x0 < x else -dx)
        self.dx = dx
        self.dE = dE
        self.Ev = self.E + dE
    def accept(self):
        self.I[self.i] += self.dx
        self.E = self.Ev
```

24.2.5 Exact density of states

We only compute to halfway since g is symmetric and the other half's large numbers cause numerical instability.

```
def reflect(a, center=True):
    if center:
        return np.hstack([a[:-1], a[-1], a[-2::-1]])
    else:
        return np.hstack([a, a[::-1]])
```

```
def bw_g(E, N, M, exact=True):
    return sum((-1)**k * special.comb(N, k, exact=exact) * special.comb(E + N - 1 - k*(M + 1), E - k*(M +
    ↪  1), exact=exact)
        for k in range(int(E / M) + 1))
def exact_bw_gs(N, M):
    Es = np.arange(N*M + 1)
    gs = np.vectorize(bw_g)(np.arange(1 + N*M // 2), N, M, exact=False)
    return Es, reflect(gs, len(Es) % 2 == 1)
```

24.3 The Wang-Landau algorithm (density of states)

We determine thermodynamic quantities from the partition function by obtaining the density of states from a simulation.

```
from numba import njit
import numpy as np
```

```
import sys
if 'src' not in sys.path: sys.path.append('src')
import simulation as sim
```

Utility functions for the simulation.

```
@njit(inline='always')
def bisect_right(a, x, lo=0, hi=None):
    if lo < 0:
        raise ValueError('lo must be non-negative')
    if hi is None:
        hi = len(a)
    while lo < hi:
        mid = (lo + hi) // 2
        if x < a[mid]:
            hi = mid
        else:
            lo = mid + 1
    return lo

@njit
def binindex(a, x):
    return bisect_right(a, x, lo=0, hi=len(a) - 1) - 1
```

```
@njit
def flat(H, ε = 0.2):
    """Determines if a histogram is approximately flat to within ε of the mean height."""
    return not np.any(H < (1 - ε) * np.mean(H)) and np.all(H ≠ 0)
```

24.3.1 Algorithm

A Wang-Landau algorithm, with quantities as logarithms and with monte-carlo steps proportional to $f^{-1/2}$ (a "Zhou-Bhat schedule").

We use energy bins encoded by numbers E_i for $i \in [0, N]$, so that there are N bins. The energies E covered by bin i satisfy $E_i \leq E < E_{i+1}$. For the bounded discrete systems that we are considering, we must choose E_N to be an arbitrary number above the maximum energy.

```
def system_prep(system):
    return system, system.energy_bins()
```

```
@njit
def simulation(system, Es,
               max_sweeps = 1_000_000,
               flat_sweeps = 1,
               eps = 1e-8,
               logf0 = 1,
```

```
                flatness = 0.2,
                log = False
               ):
    """
    Run a Wang-Landau simulation on system with energy bins Es to determine
    the system density of states g(E).

    Args:
        system: The system to perform the simulation on (see systems module).
        Es: The energy bins of the system to access. May be a subset of all bins.
        max_sweeps: The scale for the maximum number of MC sweeps per f-iteration.
            The actual maximum iterations may be fewer, but approaches max_sweeps
            exponentially as the algorithm executes.
        flat_sweeps: The number of sweeps between checks for histogram flatness.
            In AJP [10.1119/1.1707017], Landau et. al. use 10_000 sweeps.
        eps: The desired tolerance in f. Wang and Landau [WL] use 1e-8 in the original
            paper [10.1103/PhysRevLett.86.2050].
        logf0: The initial value of ln(f). WL set to 1.
        flatness: The desired flatness of the histogram. WL set to 0.2 (80% flatness).
        log: Whether or not to print results of each f-iteration.

    Returns:
        A tuple of results with entries:
        Es: The energy bins the algorithm was passed.
        S: The logarithm of the density of states (microcanonical entropy).
        H: The histogram from the last f-iteration.
        converged: True if each f-iteration took fewer than the maximum sweeps.

    Raises:
        ValueError: One of the parameters was invalid.
    """
    if (max_sweeps ≤ 0
        or flat_sweeps ≤ 0
        or eps ≤ 1e-16
        or not (0 < logf0 ≤ 1)
        or not (0 ≤ flatness < 1)):
        raise ValueError('Invalid Wang-Landau parameter.')

    # Initial values
    M = max_sweeps * system.sweep_steps
    flat_iters = flat_sweeps * system.sweep_steps
    logf = 2 * logf0 # Compensate for first loop iteration
    logftol = np.log(1 + eps)
    converged = True
    steps = 0

    E0 = Es[0]
    Ef = Es[-1]
    N = len(Es) - 1
    S = np.zeros(N) # Set all initial g's to 1
    H = np.zeros(N, dtype=np.int32)
    i = binindex(Es, system.E)

    if log:
        fiter = 0
        print("Wang-Landau START")
        print("fiter\t steps\t\t max steps")
```

```
        print("-----\t -----\t\t ---------")

    while logftol < logf:
        H[:] = 0
        logf /= 2
        iters = 0
        niters = int((M + 1) * np.exp(-logf / 2))
        if log:
            fiter += 1
        while (iters % flat_iters != 0 or not flat(H, flatness)) and iters < niters:
            system.propose()
            Ev = system.Ev
            j = binindex(Es, Ev)
            if E0 ≤ Ev < Ef and (
                S[j] < S[i] or np.random.rand() ≤ np.exp(S[i] - S[j])):
                system.accept()
                i = j
            H[i] += 1
            S[i] += logf
            iters += 1
        steps += iters
        if niters ≤ iters:
            converged = False
        if log:
            print(fiter, "\t", iters, "\t", niters)

    if log:
        print("Done: ", steps, " total MC iterations;",
              "converged." if converged else "not converged.")
    return Es, S, H, steps, converged

def wrap_results(results):
    return {k: v for k, v in zip(('Es', 'S', 'H', 'steps', 'converged'), results)}
```

24.3.2 Parallel decomposition

```
@njit
def find_bin_systems(system, Es, Ebins, sweeps = 1_000_000, method = 'wl'):
    """
    Find systems with energies in the bins given by `Es` by stepping `sys`.

    Args:
        system: The initial system to search from. This is usually a ground state.
        Es: The energies of the system.
        Ebins: The energy bins to find systems for.
        sweeps: The maximum number of MC sweeps to try.
        method: The string name of the search method to try.
            'wl': Wang-Landau steps where we prefer energies we have not visited
            'increasing': Only accept increases in energy. This only works for
                steps that are not trapped by local maxima of energy.

    Returns:
        A list of independent systems with energies in Ebins.

    Raises:
        ValueError: The method argument was invalid.
        RuntimeError: Bin systems could not be found after N steps.
```

```
    """
    if method == 'wl':
        S = np.zeros(len(Es), dtype=np.int32)
    systems = [None] * (len(Ebins) - 1)
    n = 0
    N = sweeps * system.sweep_steps
    l = len(Ebins) - 1
    systems = [system] * l
    empty = np.repeat(True, l)
    i = binindex(Es, system.E)
    while np.any(empty) and n < N:
        for s in range(l):
            if empty[s] and Ebins[s] ≤ system.E < Ebins[s + 1]:
                systems[s] = system.copy()
                empty[s] = False

        system.propose()
        j = binindex(Es, system.Ev)
        if method == 'wl':
            if S[j] < S[i]:
                i = j
                system.accept()
            S[i] += 1
        elif method == 'increasing':
            if system.E < system.Ev:
                system.accept()
        else:
            raise ValueError('Invalid method argument for finding bin systems.')
        n += 1

    if N ≤ n:
        raise RuntimeError('Could not find bin systems (hit step limit).')
    return systems
```

```
def psystem_prep(system, bins = 8, overlap = 0.1, sweeps = 1_000_000, method = 'wl', **kwargs):
    Es = system.energy_bins() # Intrinsic to the system
    Ebins = np.linspace(Es[0], Es[-1], bins + 1) # For parallel subsystems
    systems = find_bin_systems(system, Es, Ebins, sweeps, method)
    binEs = [(lambda E0, Ef: Es[(E0 ≤ Es) & (Es ≤ Ef)])(*sim.extend_bin(Ebins, i, overlap))
             for i in range(len(Ebins) - 1)]
    return zip(systems, binEs)
```

```
def run(params, **kwargs):
    return sim.run(params, simulation, system_prep, psystem_prep, wrap_results, **kwargs)
```

```
def join_results(results, *args, **kwargs):
    return sim.join_results(*zip(*[(r['Es'][:-1], r['S']) for r in results]), *args, **kwargs)
```

24.4 Thermal calculations on images

```
import numpy as np
from scipy import interpolate, special
```

```
import sys
if 'src' not in sys.path: sys.path.append('src')
import wanglandau as wl
```

24.4.1 Parallel Simulation

```
N = 16
M = 2**5 - 1
I0 = np.zeros(N, dtype=int)
system_params = {
    'StatisticalImage': {
        'I0': I0,
        'I': I0.copy(),
        'M': M
    }
}
```

```
# L = 16
# system_params = {
#     'IsingModel': {
#         'spins': np.ones((L, L), dtype=int)
#     }
# }
```

```
params = {
    'system': system_params,
    'simulation': {
        'max_sweeps': 500_000_000,
        'flat_sweeps': 10_000,
        'eps': 1e-8,
        'logf0': 1,
        'flatness': 0.1
    },
    'parallel': {
        'bins': 8,
        'overlap': 0.25,
        'sweeps': 1_000_000
    },
    'save': {
        'prefix': 'simulation-',
        'dir': 'data'
    }
}
```

```
params.pop('parallel', None) # Single run
wlresults = wl.run(params, log=True)
```

```
Run parameters
--------------
{'system': {'StatisticalImage': {'I0': array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
                                 'I': array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
                                 'M': 31}},
 'simulation': {'max_sweeps': 500000000,
                'flat_sweeps': 10000,
                'eps': 1e-08,
                'logf0': 1,
                'flatness': 0.1},
 'save': {'prefix': 'simulation-', 'dir': 'data'},
 'log': True}
```

```
Running ...
Wang-Landau START
```

iter	steps	max steps
1	1120000	4852245278
2	480000	6230406265
3	800000	7059975221
4	640000	7515304503
5	960000	7753865876
6	1600000	7875971497
7	1760000	7937743507
8	2240000	7968810956
9	2080000	7984390249
10	3520000	7992191314
11	4800000	7996094704
12	9600000	7998047114
13	10880000	7999023498
14	17760000	7999511734
15	27200000	7999755864
16	35040000	7999877931
17	21280000	7999938966
18	67200000	7999969483
19	32480000	7999984742
20	142400000	7999992371
21	104800000	7999996186
22	24000000	7999998093
23	288480000	7999999047
24	29280000	7999999524
25	38560000	7999999762
26	41600000	7999999881
27	97120000	7999999941
28	28960000	7999999971

```
Done:  1036640000  total MC iterations; converged.
... done in 116 seconds.
Writing results ... done:  data/simulation-gozi5xqv.h5
```

```
[r['converged'] for r in wlresults['results']]
```

```
[True]
```

24.4.2 Results

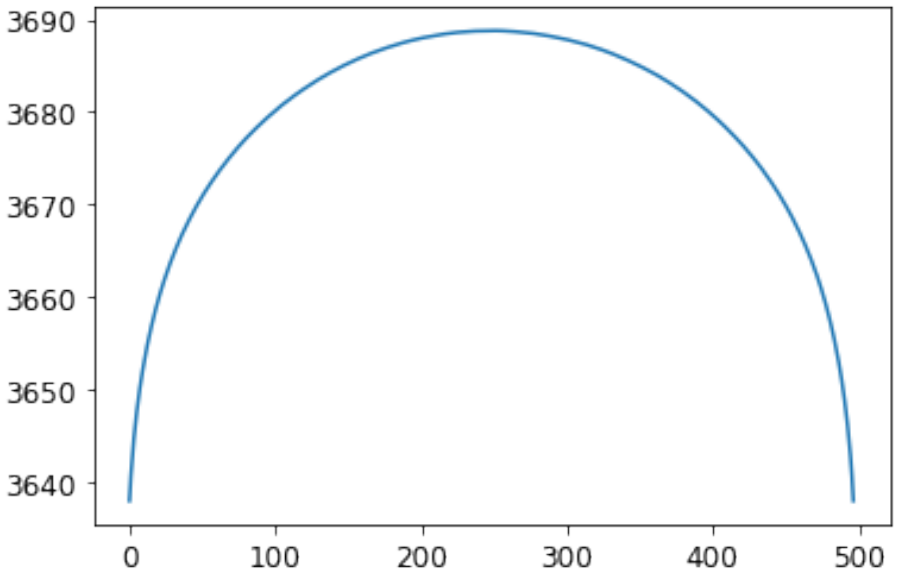
```
import matplotlib.pyplot as plt
plt.rcParams['font.size'] = 12
```

```
import h5py, hickle
with h5py.File('data/simulation-gozi5xqv.h5', 'r') as f:
```

```
    wlresults = hickle.load(f)
system_params = wlresults['parameters']['system']
```

```
wlEs, S, ΔS = wl.join_results([wlresults['results']])
```

```
for i, r in enumerate([wlresults['results']]):
    plt.plot(r['Es'][:-1], r['S'] + ΔS[i])
```




```
N, M = len(system_params['StatisticalImage']['I0']), system_params['StatisticalImage']['M']
```

Fit a spline to interpolate and optionally clean up noise, giving WL g's up to a normalization constant.

```
gspl = interpolate.splrep(wlEs, S, s=0*np.sqrt(2))
wlgs = np.exp(interpolate.splev(wlEs, gspl) - min(S))
```

24.4.3 Exact density of states

We only compute to halfway since g is symmetric and the other half's large numbers cause numerical instability.

```
def reflect(a, center=True):
    if center:
        return np.hstack([a[:-1], a[-1], a[-2::-1]])
    else:
        return np.hstack([a, a[::-1]])
```

The exact density of states for uniform values. This covers the all gray and all black/white cases. Everything else (normal images) are somewhere between. The gray is a slight approximation: the ground level is not degenerate, but we say it has degeneracy 2 like all the other sites. For the numbers of sites and values we are using, this is insignificant.

```
def bw_g(E, N, M, exact=True):
    return sum((-1)**k * special.comb(N, k, exact=exact) * special.comb(E + N - 1 - k*(M + 1), E - k*(M +
    ↪  1), exact=exact)
        for k in range(int(E / M) + 1))
def exact_bw_gs(N, M):
    Es = np.arange(N*M + 1)
    gs = np.vectorize(bw_g)(np.arange(1 + N*M // 2), N, M, exact=False)
    return Es, reflect(gs, len(Es) % 2 == 1)
```

```
def gray_g(E, N, M, exact=True):
    return 2 * bw_g(E, N, M, exact=exact)
def exact_gray_gs(N, M):
    Es = np.arange(N*M + 1)
    gs = np.vectorize(gray_g)(np.arange(1 + N*M // 2), N, M, exact=False)
    return Es, reflect(gs, len(Es) % 2 == 1)
```

Expected results for black/white and gray.

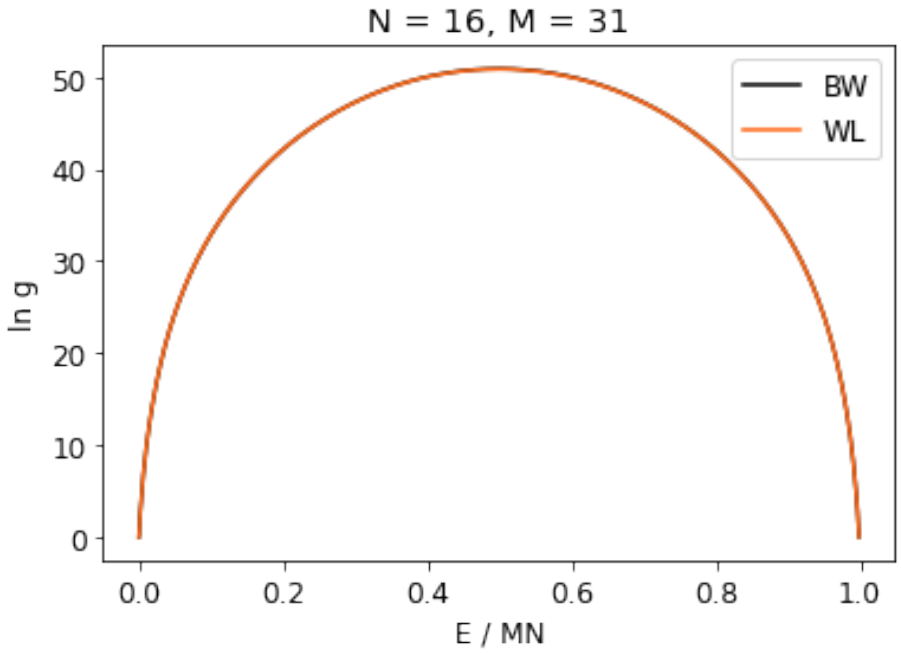
```
bw_Es, bw_gs = exact_bw_gs(N=N, M=M)
gray_Es, gray_gs = exact_gray_gs(N=N, M=-1 + (M + 1) // 2)
```

Choose what to compare to.

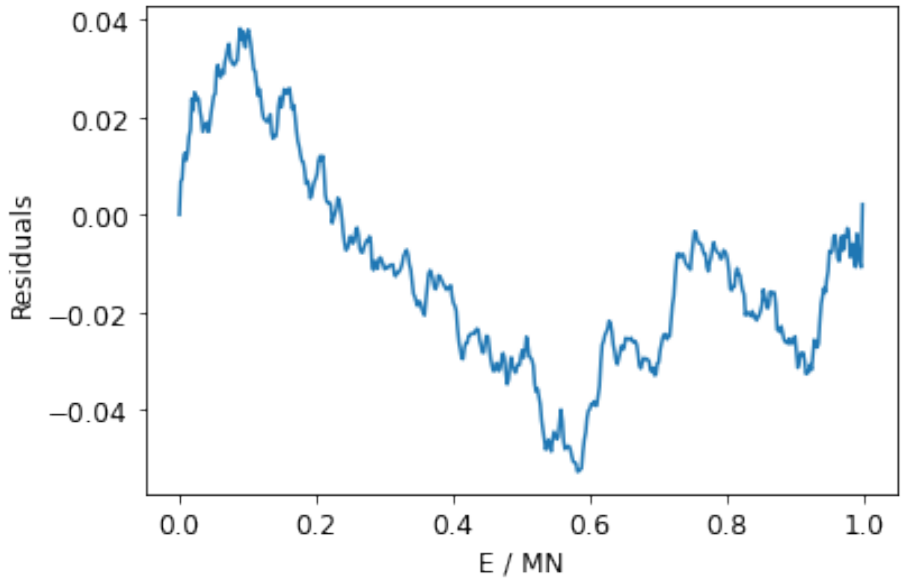
```
Es, gs = bw_Es, bw_gs
```

Presumably all of the densities of states for different images fall in the region between the all-gray and all-black/white curves.

```
plt.plot(bw_Es / len(bw_Es), np.log(bw_gs), 'black', label='BW')
plt.plot(wlEs / len(wlEs), S - min(S), '#ff6716', label='WL')
plt.xlabel('E / MN')
plt.ylabel('ln g')
plt.title('N = {}, M = {}'.format(N, M))
plt.legend()
plt.savefig('wanglandau-bw.png', dpi=600)
```




```
# plt.plot(wlEs / len(wlEs), np.abs(wlgs - bw_gs) / bw_gs)
# plt.ylabel('Relative error')
plt.plot(wlEs / len(wlEs), S - np.log(bw_gs) - min(S))
plt.ylabel('Residuals')
plt.xlabel('E / MN');
```



```
print('End of job.')
```

```
End of job.
```

24.5 Calculating canonical ensemble averages

```
import numpy as np
```

```
class Ensemble:
    def __init__(self, Es, lngs, name = 'Canonical ensemble', λ = None):
        self.Es = Es
        self.lngs = lngs
        self.name = name
        # Choose to scale the exponent of Z to be a convenient size.
        # This can be improved if needed by taking into account typical β*Es.
        self.λ = max(lngs) if λ is None else λ
    def Zλ(self, β):
        return np.sum(np.exp(-(np.outer(β, self.Es) - self.lngs + self.λ)), 1)
    def Z(self, β):
        return np.exp(self.λ) * self.Zλ(β)
    def p(self, β):
        return np.exp(-(β * self.Es - self.lngs + self.λ)) / self.Zλ(β)
    def average(self, f, β):
        return np.sum(f(self) * np.exp(-(np.outer(β, self.Es) - self.lngs + self.λ)), 1) / self.Zλ(β)
    def energy(self, β):
        return self.average(lambda ens: ens.Es, β)
    def energy2(self, β):
        return self.average(lambda ens: ens.Es**2, β)
    def heat_capacity(self, β):
        return self.energy2(β) - self.energy(β)**2
    def free_energy(self, β):
        return -np.log(self.Z(β)) / β
    def entropy(self, β):
        return β * self.energy(β) + np.log(self.Z(β))
```

24.6 Simulation error of Wang-Landau results for black Statistical Images

```
import numpy as np
from scipy import interpolate, special
import os, h5py, hickle
import matplotlib.pyplot as plt
import pprint
plt.rcParams['font.size'] = 12
```

```
import sys
if 'src' not in sys.path: sys.path.append('src')
import wanglandau as wl
from statistical_image import exact_bw_gs
import canonical_ensemble as canonical
```

24.6.1 The setup

```
datadir = 'data/black-images'
paths = [os.path.join(datadir, f) for f in os.listdir(datadir)]
len(paths)
```

```
1024
```

```
with h5py.File(paths[0], 'r') as f:
    result = hickle.load(f)
    imp = result['parameters']['system']['StatisticalImage']
    N = len(imp['I0'])
    M = imp['M']
    Es = result['results']['Es'][:-1]
```

```
pprint.pprint(result['parameters'])
```

```
{'log': True,
 'simulation': {'eps': 1e-08,
                'flat_sweeps': 10000,
                'flatness': 0.2,
                'logf0': 1,
                'max_sweeps': 100000000},
 'system': {'StatisticalImage': {'I': array([10,  7,  1, 10,  6,  0,  0,  0, 28,  0,  7,  2,  1,  1,  1, 11]),
                                 'I0': array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
                                 'M': 31}}}
```

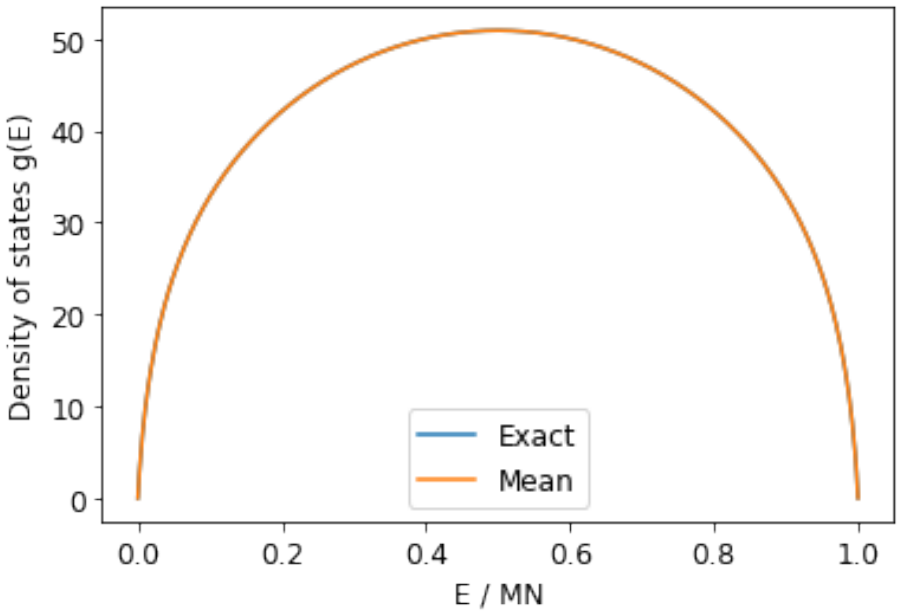
```
def file_lngs(path):
    with h5py.File(path, 'r') as f:
        result = hickle.load(f)
        S = result['results']['S']
        # Shift for computing exponentials
        S -= min(S)
        # Set according to the correct total number of states ((M+1)**N)
        S += N*np.log(M+1) - np.log(np.sum(np.exp(S)))
        # Set according to leftmost value
#         S -= S[0]
        return S
```

24.6.2 Error in the log density of states

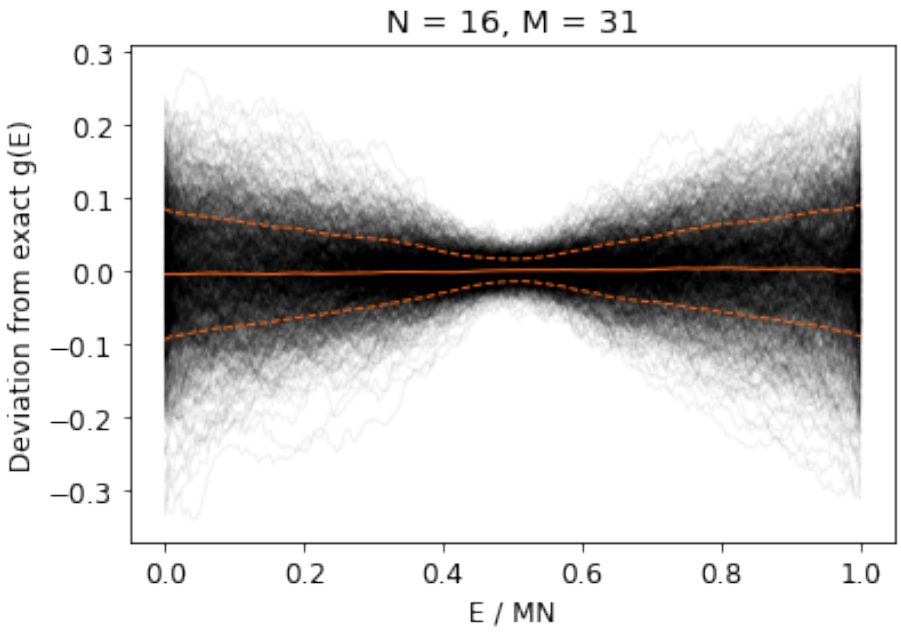
```
xEs, xgs = exact_bw_gs(N, M)
xlng = np.log(xgs)
```

```
mean_lng = np.zeros(len(Es))
std_lng = np.zeros(len(Es))
for lng in map(file_lngs, paths):
    mean_lng += lng
mean_lng /= len(paths)
for lng in map(file_lngs, paths):
    std_lng += (mean_lng - lng)**2
std_lng = np.sqrt(std_lng / (len(paths) - 1))
```

```
plt.plot(xEs / (M*N), np.log(xgs), label='Exact')
plt.plot(Es / (M*N), mean_lng, label='Mean')
plt.xlabel('E / MN')
plt.ylabel('Density of states g(E)')
plt.legend();
```

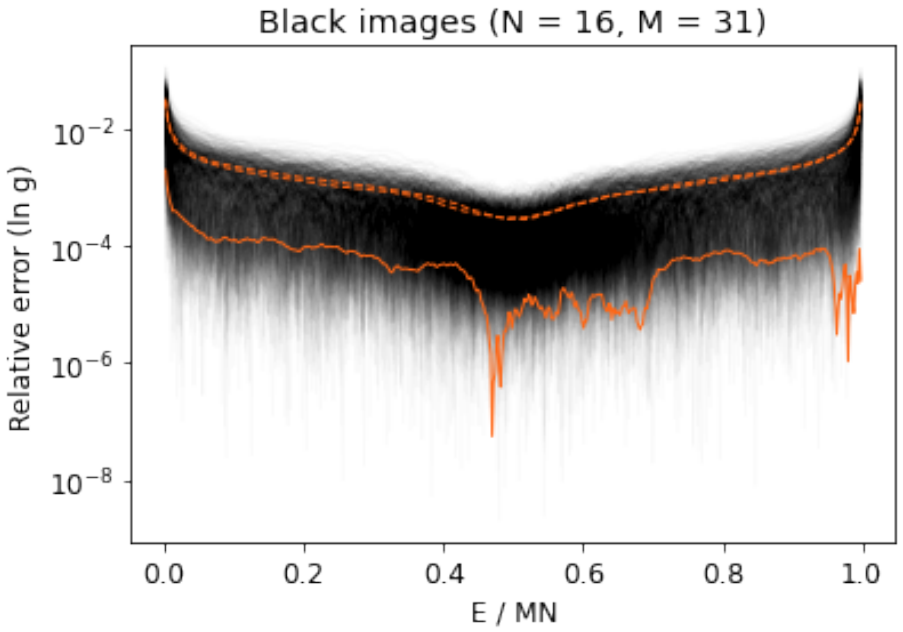



```
for lng in map(file_lngs, paths):
    plt.plot(Es / (M*N), lng - xlng, 'black', alpha=0.05, linewidth=1)
plt.plot(Es / (M*N), mean_lng - xlng, '#ff6716', linewidth=1)
plt.plot(Es / (M*N), (mean_lng - std_lng) - xlng, '#ff6716', linestyle='dashed', linewidth=1)
plt.plot(Es / (M*N), (mean_lng + std_lng) - xlng, '#ff6716', linestyle='dashed', linewidth=1)
plt.title('N = {}, M = {}'.format(N, M))
plt.xlabel('E / MN')
plt.ylabel('Deviation from exact g(E)')
plt.savefig('wanglandau-bw-deviation.png', dpi=600);
```




```
def relative_error(sim, exact):
    if exact == 0.0:
        return np.inf
    else:
        return np.abs(sim - exact) / exact
def relerror(sim, exact = xlng):
    return np.vectorize(relative_error)(sim, exact)
def log_relerror(sim, exact = xlng):
    return np.log10(relerror(sim, exact))
```

```
for lng in map(file_lngs, paths):
    plt.plot(Es / (M*N), relerror(lng), 'black', alpha=0.02, linewidth=1)
plt.plot(Es / (M*N), relerror(mean_lng), '#ff6716', linewidth=1)
plt.plot(Es / (M*N), relerror(mean_lng - std_lng), '#ff6716', linestyle='dashed', linewidth=1)
plt.plot(Es / (M*N), relerror(mean_lng + std_lng), '#ff6716', linestyle='dashed', linewidth=1)
plt.title('Black images (N = {}, M = {})'.format(N, M))
plt.xlabel('E / MN')
plt.ylabel('Relative error (ln g)')
plt.yscale('log')
plt.savefig('wanglandau-bw-relerror.png', dpi=600);
```

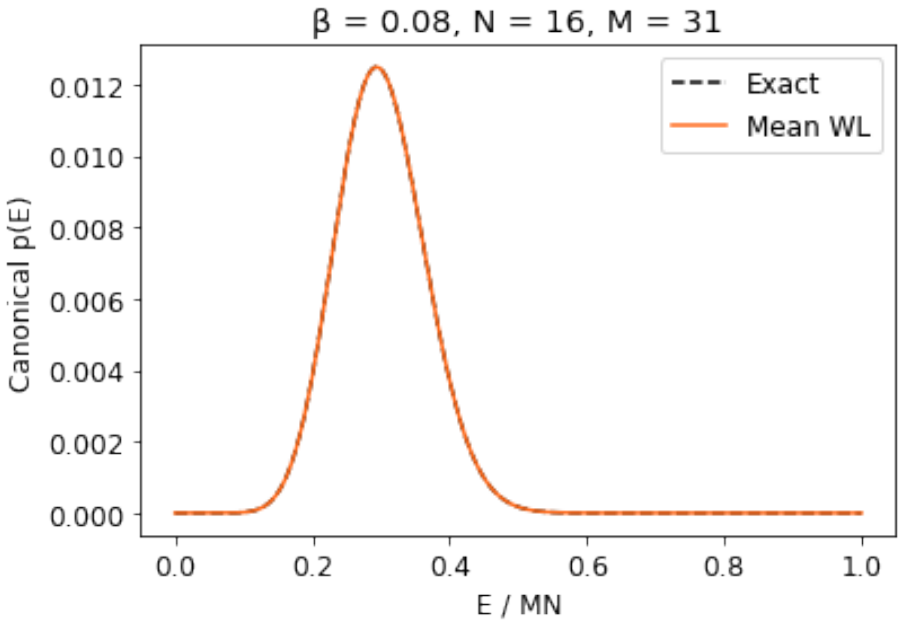


24.6.3 Error in canonical ensemble variables

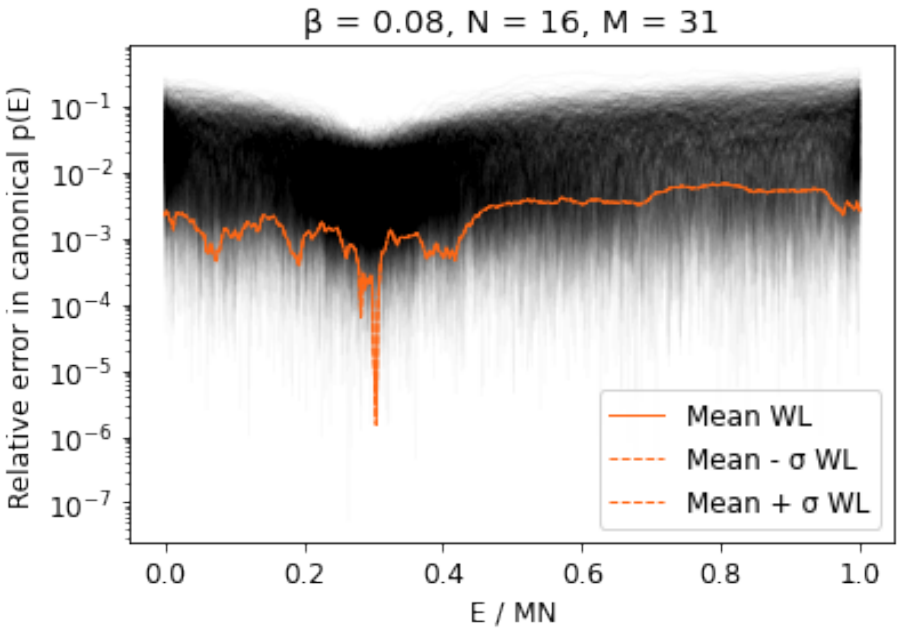
```
βs = np.exp(np.linspace(-7, 4, 500))
exact_ens = canonical.Ensemble(Es, xlng, 'Exact')
mean_ens = canonical.Ensemble(Es, mean_lng, 'Mean WL')
mσ_ens = canonical.Ensemble(Es, mean_lng, 'Mean - σ WL')
pσ_ens = canonical.Ensemble(Es, mean_lng, 'Mean + σ WL')
```

The canonical distribution for fixed β.

```
βc = 8e-2
plt.plot(Es / (M*N), exact_ens.p(βc), 'black', label=exact_ens.name, linestyle='dashed')
plt.plot(Es / (M*N), mean_ens.p(βc), '#ff6716', label=mean_ens.name)
plt.title('β = {}, N = {}, M = {}'.format(βc, N, M))
plt.xlabel("E / MN")
plt.ylabel("Canonical p(E)")
plt.legend();
```

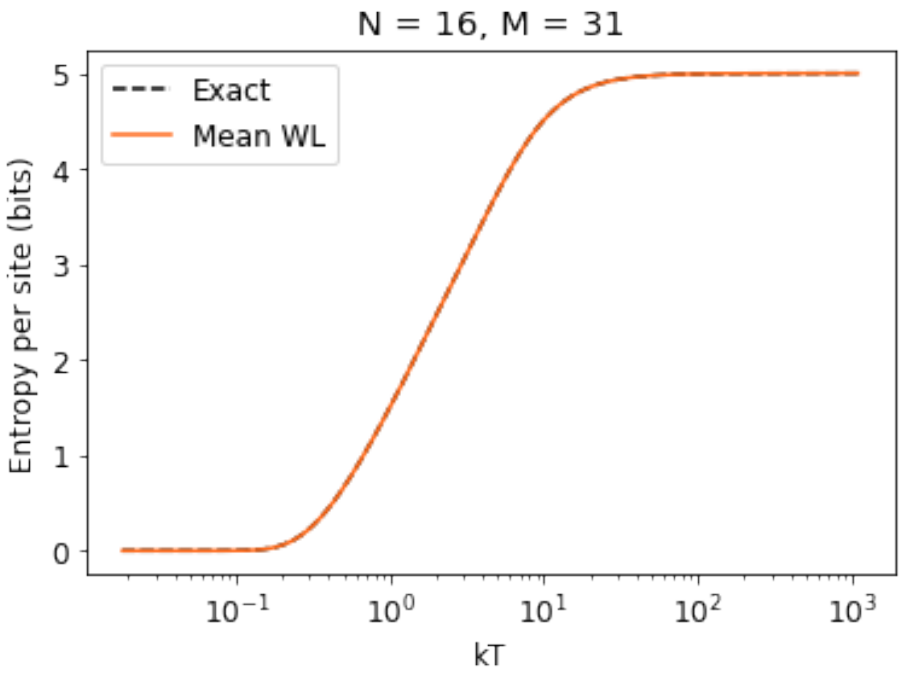



```
for lng in map(file_lngs, paths):
    ens = canonical.Ensemble(Es, lng)
    plt.plot(Es / (M*N), relerror(ens.p(βc), exact_ens.p(βc)),
             'black', alpha=0.02, linewidth=1)
plt.plot(Es / (M*N), relerror(mean_ens.p(βc), exact_ens.p(βc)),
             '#ff6716', linewidth=1, label=mean_ens.name)
plt.plot(Es / (M*N), relerror(mσ_ens.p(βc), exact_ens.p(βc)),
             '#ff6716', linewidth=1, linestyle='dashed', label=mσ_ens.name)
plt.plot(Es / (M*N), relerror(pσ_ens.p(βc), exact_ens.p(βc)),
             '#ff6716', linewidth=1, linestyle='dashed', label=pσ_ens.name)
plt.title('β = {}, N = {}, M = {}'.format(βc, N, M))
plt.xlabel("E / MN")
plt.ylabel("Relative error in canonical p(E)")
plt.yscale('log')
plt.legend();
```



Entropy

```
plt.plot(1 / βs, exact_ens.entropy(βs) / (N*np.log(2)), 'black', label=exact_ens.name, linestyle='dashed')
plt.plot(1 / βs, mean_ens.entropy(βs) / (N*np.log(2)), '#ff6716', label=mean_ens.name)
plt.xlabel("kT")
plt.xscale('log')
plt.ylabel("Entropy per site (bits)")
plt.title('N = {}, M = {}'.format(N, M))
plt.legend()
plt.savefig('wanglandau-bw-S.png', dpi=600)
```




```
exact_ens.entropy(0) / N
```

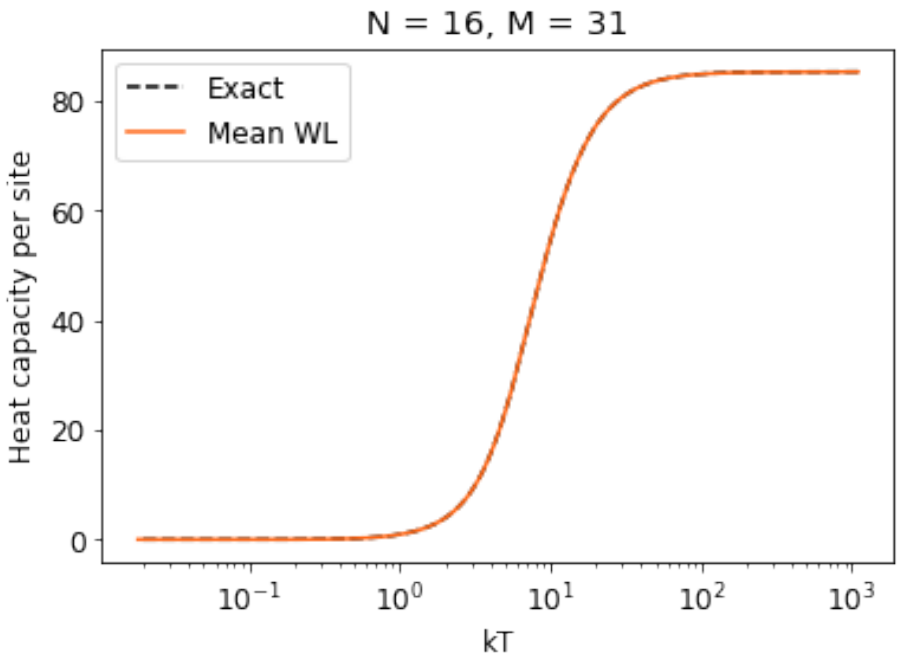
```
array([3.4657359])
```

```
exact_ens.entropy(0) / (N*np.log(2))
```

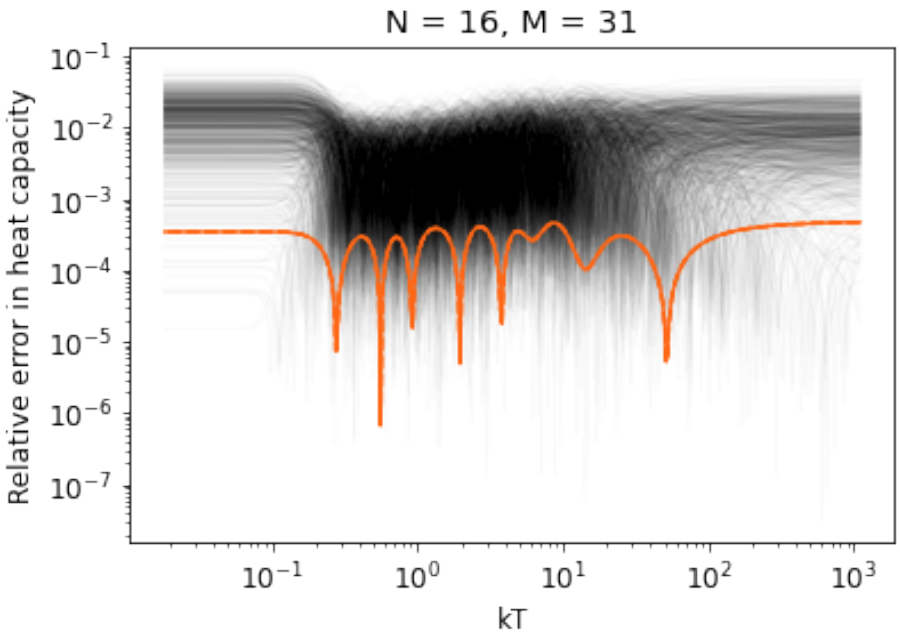
```
array([5.])
```

The relative error in the heat capacity provides a stringent test of the results.

```
plt.plot(1 / βs, exact_ens.heat_capacity(βs) / N, 'black', label=exact_ens.name, linestyle='dashed')
plt.plot(1 / βs, mean_ens.heat_capacity(βs) / N, '#ff6716', label=mean_ens.name)
plt.xlabel("kT")
plt.xscale('log')
plt.ylabel("Heat capacity per site")
plt.title('N = {}, M = {}'.format(N, M))
plt.legend()
plt.savefig('wanglandau-bw-C.png', dpi=600)
```




```
for lng in map(file_lngs, paths):
    ens = canonical.Ensemble(Es, lng)
    plt.plot(1 / βs, relerror(ens.heat_capacity(βs), exact_ens.heat_capacity(βs)),
             'black', alpha=0.02, linewidth=1)
plt.plot(1 / βs, relerror(mean_ens.heat_capacity(βs), exact_ens.heat_capacity(βs)),
             '#ff6716', label=mean_ens.name)
plt.plot(1 / βs, relerror(mσ_ens.heat_capacity(βs), exact_ens.heat_capacity(βs)),
             '#ff6716', linestyle='dashed', label=mσ_ens.name)
plt.plot(1 / βs, relerror(pσ_ens.heat_capacity(βs), exact_ens.heat_capacity(βs)),
             '#ff6716', linestyle='dashed', label=pσ_ens.name)
plt.xlabel('kT')
plt.xscale('log')
plt.ylabel('Relative error in heat capacity')
plt.yscale('log')
plt.title('N = {}, M = {}'.format(N, M))
plt.savefig('wanglandau-bw-C-relerror.png', dpi=600)
```



25 Progress summary

June 26, 2020

This week, I improved my implementation of the Wang-Landau algorithm and verified its correctness. I completed the convenience functions from before, which allowed me to manage and combine thousands of simulations to characterize the error in the density of states. For the same simulation parameters, my relative error on the order of 10^{-4} in the density of states for image systems is comparable to that of Landau ($10^{-3.5}$) for an Ising ferromagnet.[5] The level of the microcanonical entropy is now adjusted so that the sum over the density of states is the total number of states. This prioritizes moderate temperature correctness, but we may also set the level according to a ground state, which increases correctness at low temperatures. I also calculated the corresponding errors in thermodynamic quantities like heat capacity, which gave consistent results.

Another batch of simulations was performed for random grayscale images. The resulting densities of states were spread out over energies past half the maximum energy for a black image, as expected. The energy, heat capacity, and entropy all varied little across random images for temperatures which induce energy fluctuations at the scale of one gray level, but showed significant variance at intermediate temperatures. This reflects how the energy landscape near the ground image is the same for all gray images, but varies as the temperature becomes high enough to reach the bounds of allowed gray values.

[5] DOI: 10.1119/1.1707017

Now that the Wang-Landau algorithm has proven itself to be a useful tool for density estimation, we aim to use it and other methods to quantify the relative information content of different aspects of vision for the second half of the project.

26 Progress summary

July 3, 2020 Not much progress this week, since I took it mostly off to rest, prepare for the PGRE a bit, and celebrate the fourth of July.

27 Entropy of coordinate systems

July 5, 2020 What is the entropy of a continuous space, and what role does the choice of coordinates play? We would like to capture the sense in which it takes much less information to specify a point from a normal distribution with variance σ^2 on $\mathbb{R}^3$ than it does to specify an arbitrary point. Consider a ball of radius R about the origin. If $R \ll \sigma$, then both distributions are about the same, and should not differ much in required information, but for $R \gg \sigma$, the divergence of the uniform distribution from the normal distribution should grow without bound.

Spherical coordinates are given by the usual map $\Phi : [0, R] \times [0, \pi] \times [0, 2\pi] \to \mathbb{R}^3$ with Jacobian $r^2 \sin\theta$. A probability distribution on $\mathbb{R}^3$ may be pulled back to particular coordinates. The uniform distribution on the ball $B(0, R)$ pulls back on Cartesian coordinates to

$$p_u(x, y, z) = \frac{3}{4\pi R^3},$$

but on spherical coordinates pulls back to

$$p_u(r, \theta, \varphi) = \frac{3r^2 \sin\theta}{4\pi R^3}.$$

The spherical coordinates form independent random variables with marginal distributions

$$p_u(r) = \frac{3r^2}{R^3}, \quad p_u(\theta) = \frac{\sin\theta}{2}, \quad \text{and} \quad p_u(\varphi) = \frac{1}{2\pi},$$

whereas the Cartesian coordinates form dependent random variables with conditional distributions

$$p_u(x_i \mid x_{j\neq i}) = \frac{1}{2}\left(R^2 - \sum_{j\neq i} x_j^2\right)^{-1/2}.$$

Similarly, the truncated normal distribution with unit variance on the ball pulls back in spherical coordinates to

$$p(r, \theta, \varphi) = \left[\operatorname{erf}\left(\frac{R}{\sqrt{2}}\right) - R\sqrt{\frac{2}{\pi}} e^{-R^2/2}\right]^{-1} \frac{e^{-r^2/2}}{\sqrt{8\pi^3}}\, r^2 \sin\theta.$$

While $p(\theta) = p_u(\theta)$ and $p(\varphi) = p_u(\varphi)$, the marginal distribution of the radius is

$$p(r) = \left[\operatorname{erf}\left(\frac{R}{\sqrt{2}}\right) - R\sqrt{\frac{2}{\pi}}e^{-R^2/2}\right]^{-1} \frac{e^{-r^2/2}}{\sqrt{2\pi}}\, 2r^2.$$

The KL divergence from the uniform to the truncated normal distribution over the ball is

$$D_{KL}(p \parallel p_u) = \int_{r=0}^{R}\int_{\theta=0}^{\pi}\int_{\varphi=0}^{2\pi} p \ln \frac{p}{p_u}\, \mathrm{d}r\, \mathrm{d}\theta\, \mathrm{d}\varphi\,,$$

independent of using spherical coordinates (Fig. 2). But since the spherical coordinates form independent random variables for both distributions, this separates as

$$\begin{aligned} &D_{KL}(p(r,\, \theta,\, \varphi) \parallel p_u(r,\, \theta,\, \varphi)) \\ &= D_{KL}(p(r) \parallel p_u(r)) + D_{KL}(p(\theta) \parallel p_u(\theta)) + D_{KL}(p(\varphi) \parallel p_u(\varphi)). \end{aligned}$$

As only the radial distribution is different,

$$D_{KL}(p(r,\, \theta,\, \varphi) \parallel p_u(r,\, \theta,\, \varphi)) = D_{KL}(p(r) \parallel p_u(r)),$$

as expected from the symmetries.

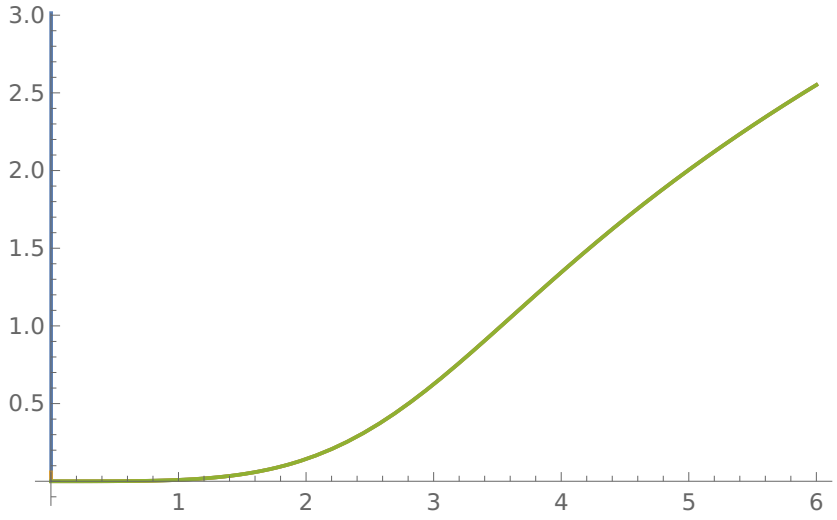


Figure 2: The KL divergence from uniform to truncated normal as a function of ball radius.

How can we quantify the relative information between coordinates $(x_1, \ldots, x_m)$? I suggest

$$I(x_i) = \frac{D_{KL}(p(x_i) \parallel p_u(x_i))}{D_{KL}(p(x_1, \ldots, x_m) \parallel p_u(x_1, \ldots, x_m))}$$

as a measure of the information in the coordinate x_i given the distribution p. This is relative to the uniform distribution p_u where all coordinates are considered equally informative, but a different reference distribution may be chosen for unbounded parameters. This quantity is invariant under changes in scale and has $\sum_i I(x_i) = 1$.

A simple example is a uniform distribution on a rectangle. The idea is that to determine a point in a square within the rectangle, it helps more to know the coordinate of the longer side. To encode this in probability distributions, we let $p_u(x, y) = (ab)^{-1}[0 \le x \le a, 0 \le y \le b]$ and $p(x, y) = c^{-2}[0 \le x \le c, 0 \le y \le c]$ with $c \le a \le b$. Then

$$I(x) = \frac{D_{KL}(p(x) \parallel p_u(x))}{D_{KL}(p(x, y) \parallel p_u(x, y))} = \frac{\ln(a/c)}{\ln(a/c) + \ln(b/c)},$$

with $I(y)$ defined similarly. As $c \to 0$, $I \to 1/2$, as the dimensions of the rectangle become irrelevant. As $c \to a$, $I(x) \to 0$ and $I(y) \to 1$, as you only need to restrict y for a point to be in the square. If $a = b$, then there is never any difference and $I(x) = I(y) = 1/2$ independent of c. For $a \ll b$, $I(x)$ is nearly zero and $I(y)$ is nearly one over most of the range of c, though both still return to $1/2$ as $c \to 0$.

In light of the previous calculation in spherical coordinates, we see that $I(r) = 1$ and $I(\theta) = I(\varphi) = 0$, since we need only know the radius to know if a point is probable.

28 Progress summary

July 10, 2020 This week, I performed a second literature search to determine our next steps. We are now going to focus on the statistics of natural images and how they may be generated.

29 Natural images

July 15, 2020 We consider some of the statistics mentioned in Ruderman's statistics of natural images, and investigate his suggestion that natural images may be able to be reconstructed by somehow inverting the reduction of an image to variances of image blocks.[6]

We later consider heuristic approaches using fractal noise, which we construct from combining Perlin noise at different scales. This is standard practice for procedurally generating terrain in computer graphics. From the perspective of Kolmogorov complexity, the methods used in computer graphics and video games may be taken as upper bounds for optimal programs that generate natural images.

30 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
```

[6] Doi: 10.1088/0954-898X_5_4_006.

```
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)
```

30.1 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)
```

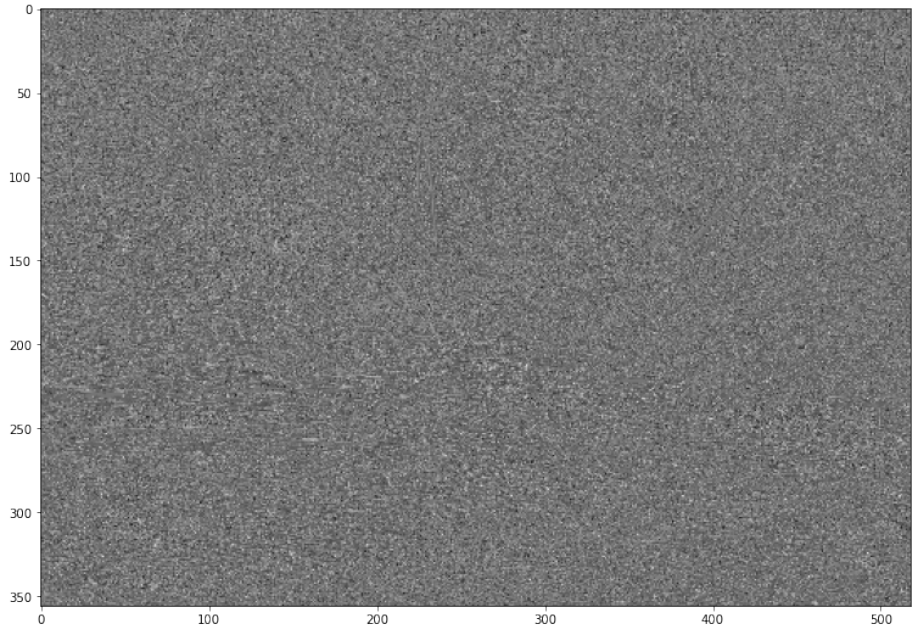
```
np.array(np.shape(img)) // 5
```

```
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)
```

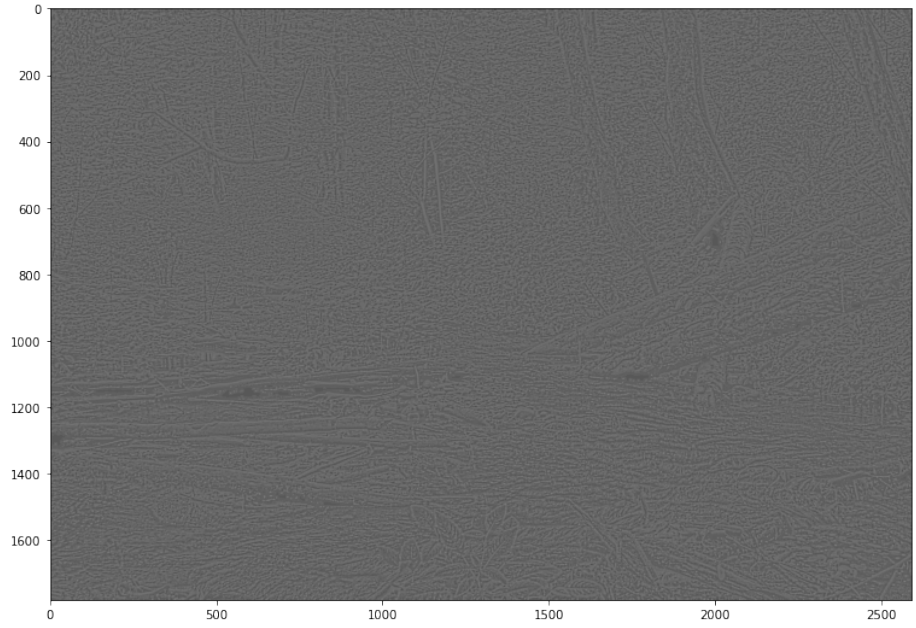
```
rdata = iterate(renorm_blocks(5), 1)(img)
plt.imshow(rdata);
```




```
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)
```

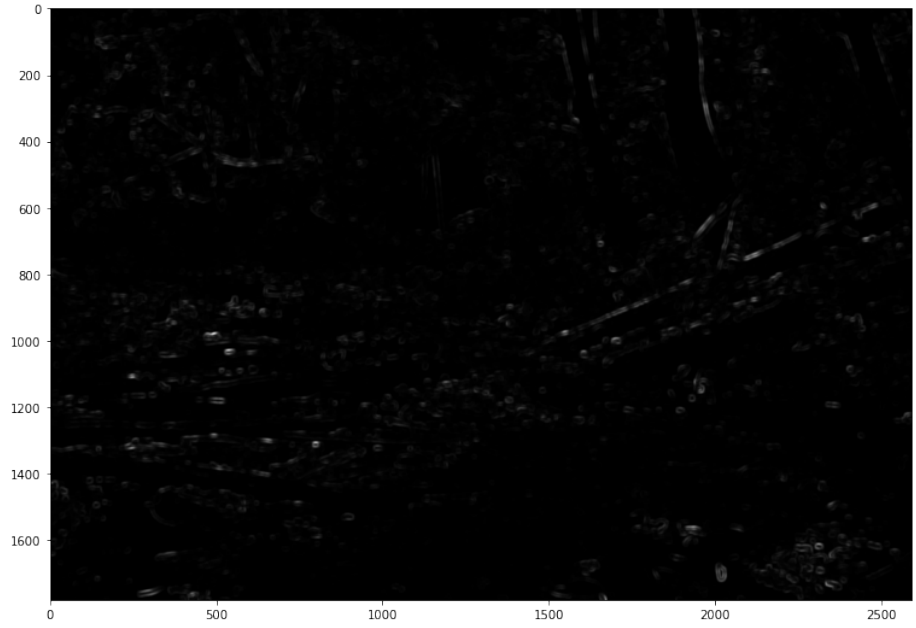
```
plt.imshow(l1data);
```



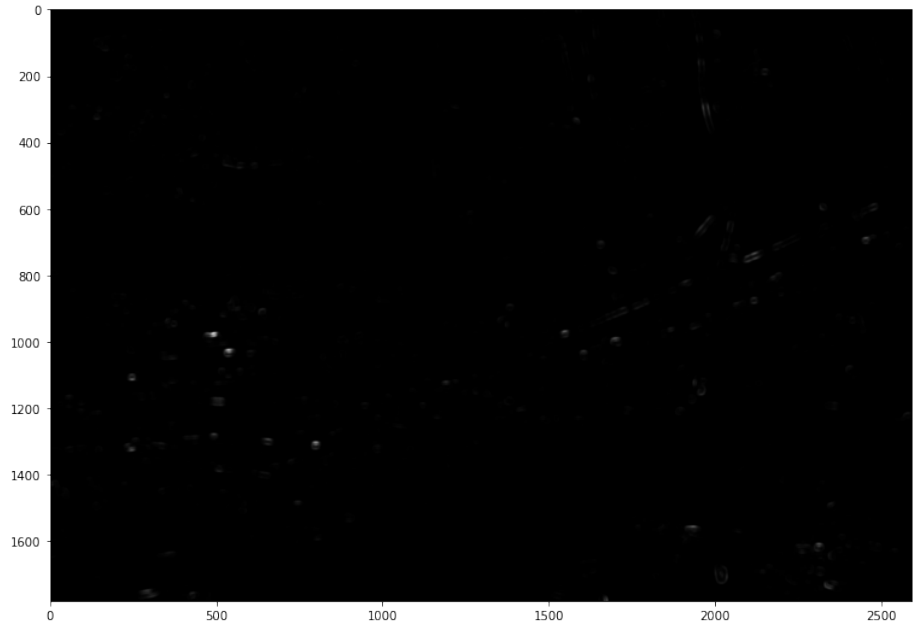

```
v1data = iterate(lambda x: mapbox(5, np.var, x), 1)(data)
v2data = iterate(lambda x: mapbox(5, np.var, x), 1)(v1data)
```

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


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



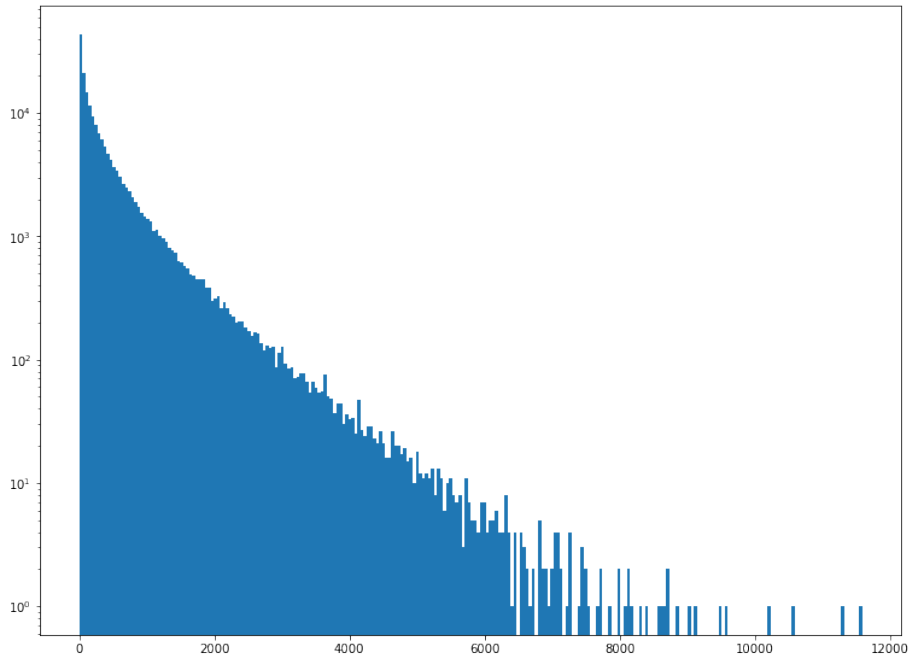
```
plt.imshow(v3data);
```



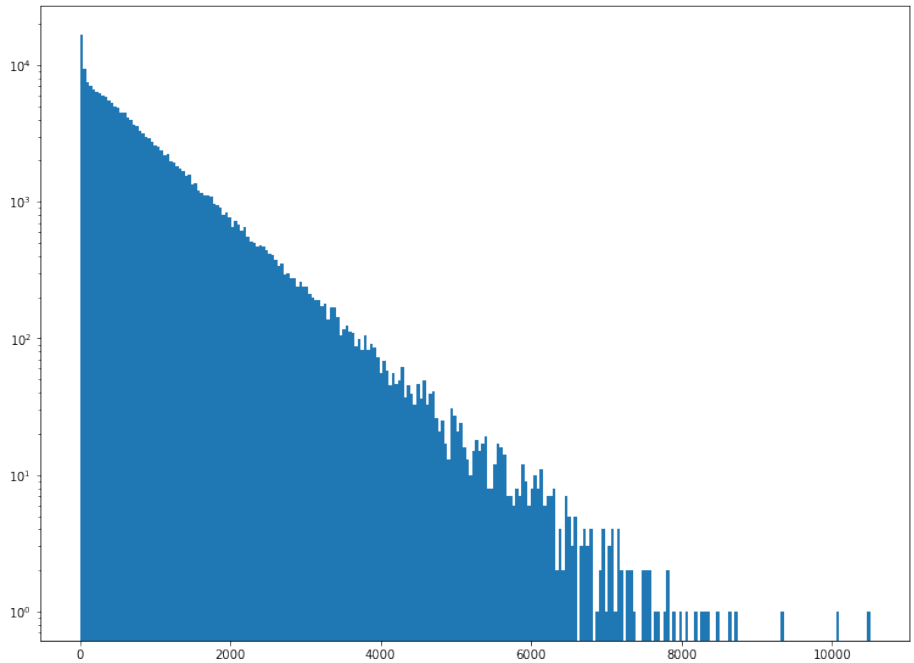

```
sep_vblocks = renorm_blocks(5, np.var)(data)
plt.imshow(sep_vblocks);
```

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

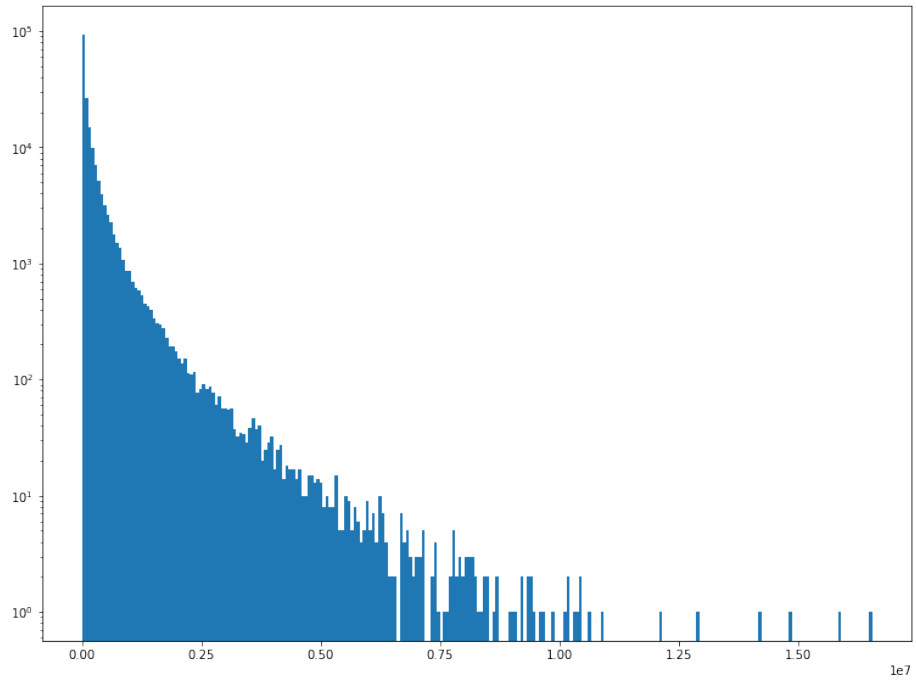

```
plt.hist(sep_vblocks.flat, 256)
plt.yscale('log');
```




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



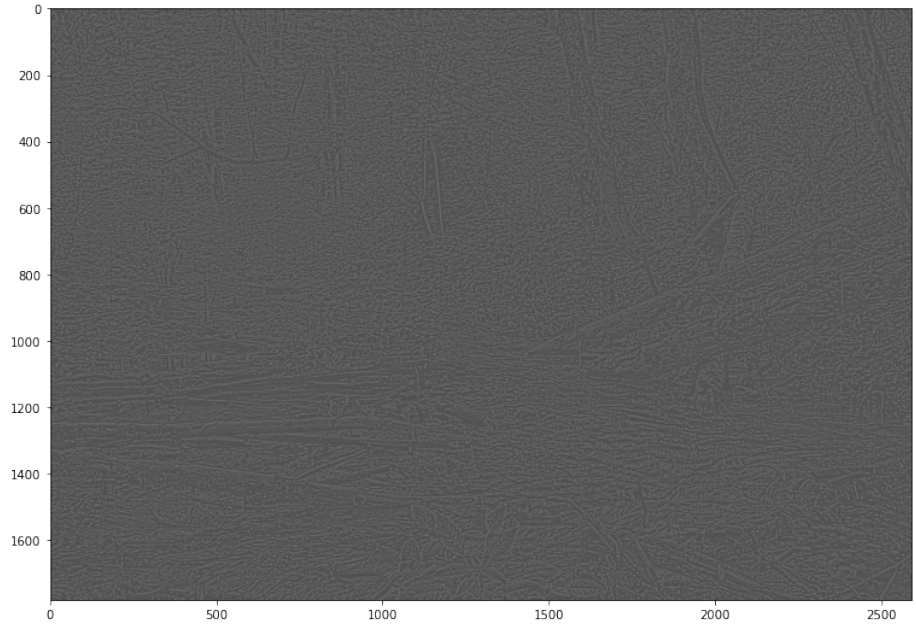
```
mean_v2blocks = renorm_blocks(5, np.mean)(v2data)
plt.hist(mean_v2blocks.flat, 256)
plt.yscale('log');
```



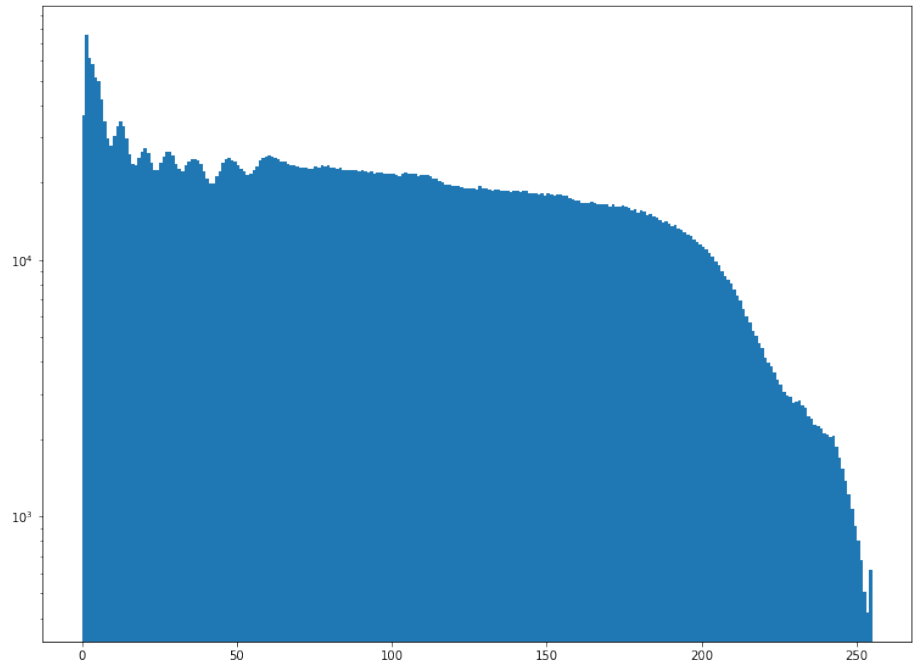

```
vdata = iterate(lambda x: mapbox(5, np.var, x), 1)(ldata)
plt.imshow(vdata);
```


```
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))
```

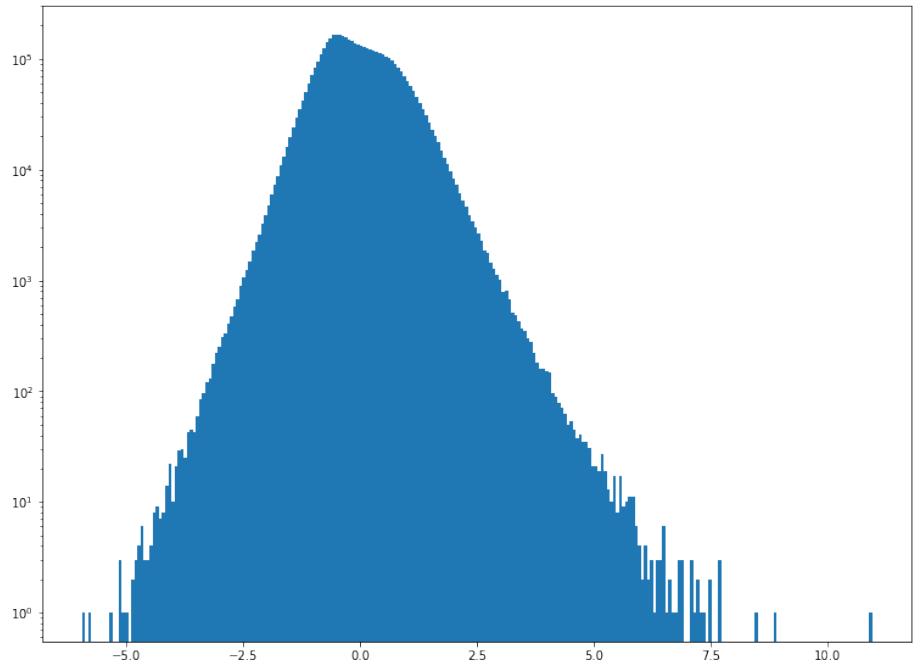
```
plt.imshow(g1data);
```



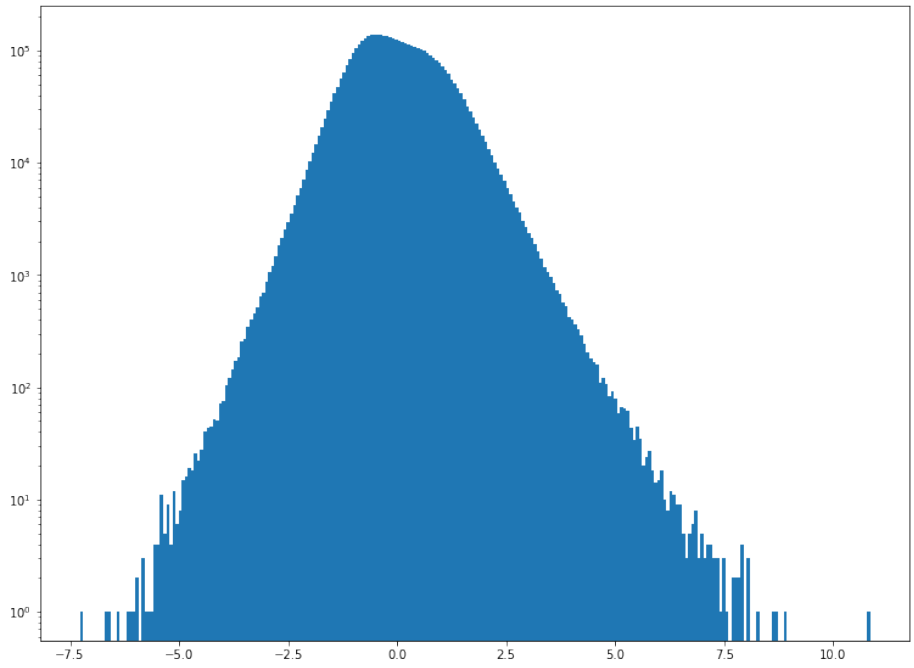

```
plt.hist(np.array(img).flat, range(256))
plt.yscale('log');
```



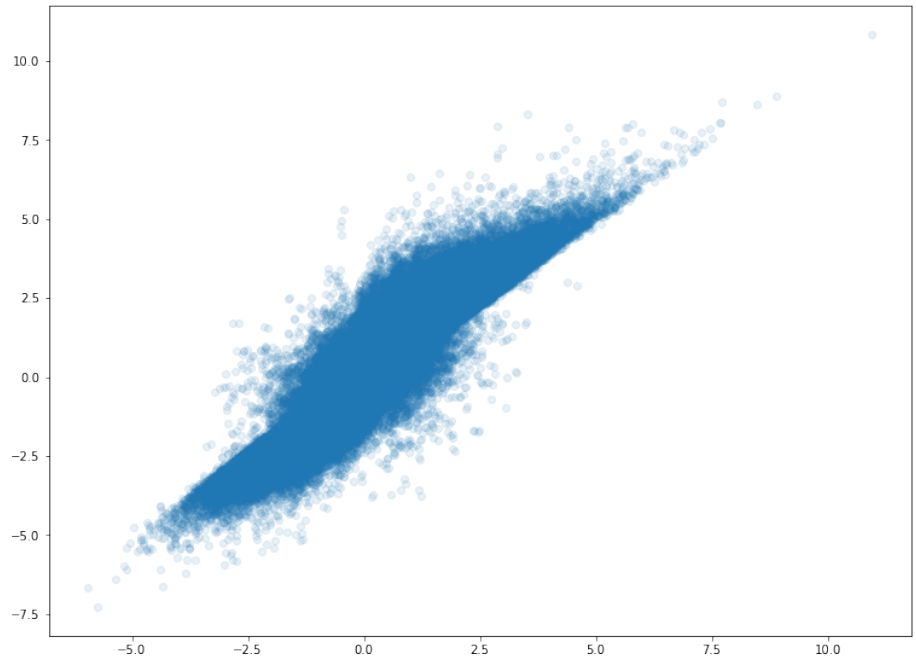

```
plt.hist(g1data.flat, 256)
plt.yscale('log');
```



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

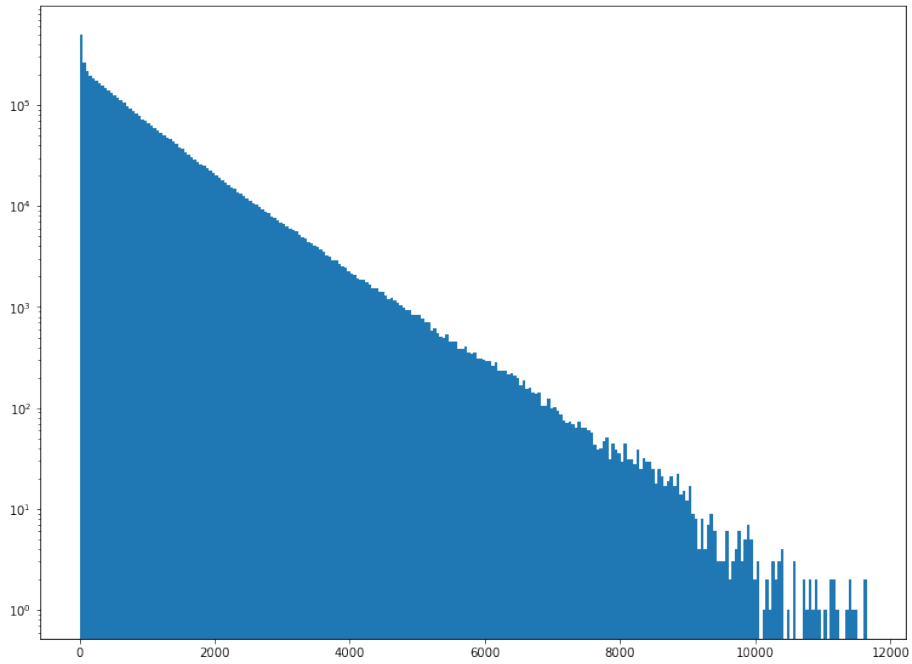



```
plt.scatter(g1data.flat, g2data.flat, alpha=0.1);
```

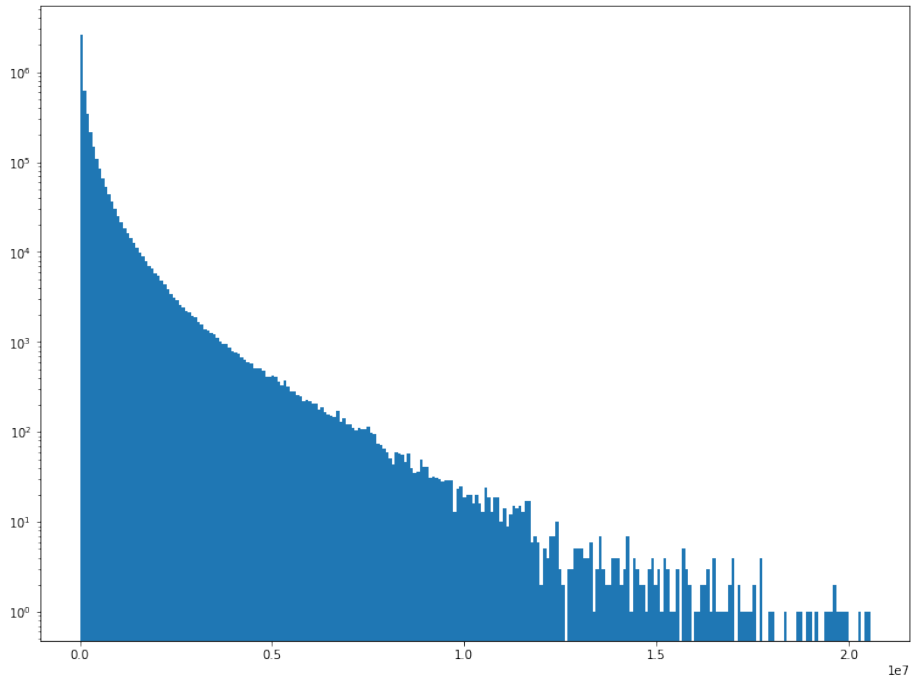



```
plt.imshow(vdata);
```

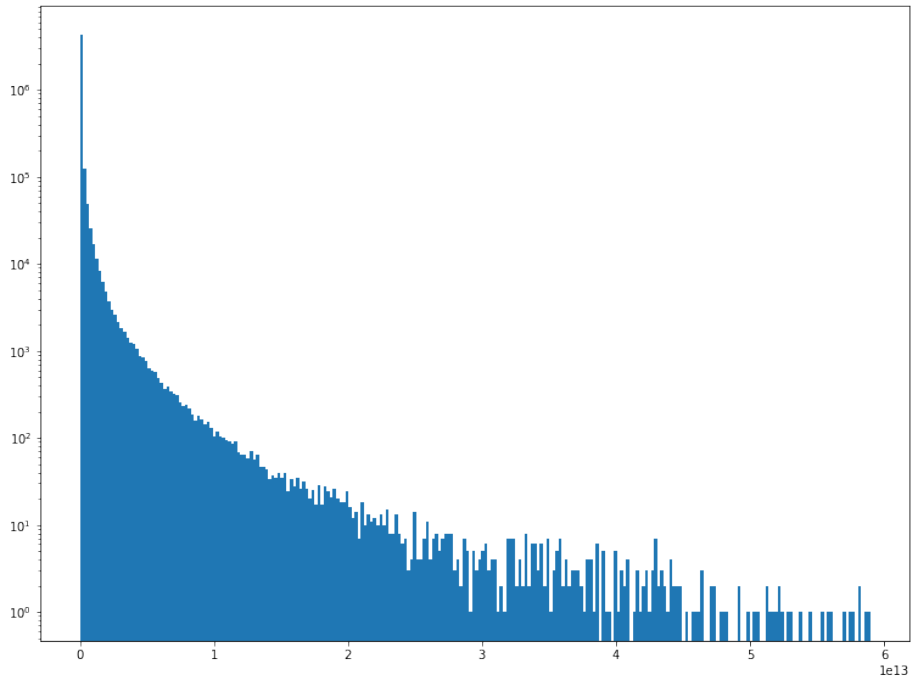

```
plt.hist(v1data.flat, 256)
plt.yscale('log');
```



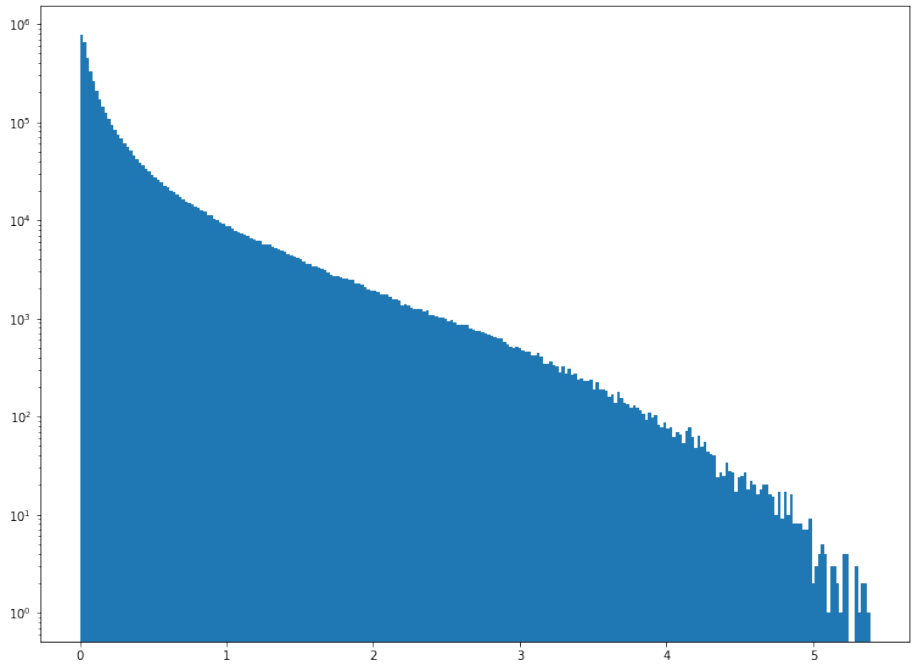

```
plt.hist(v2data.flat, 256)
plt.yscale('log');
```



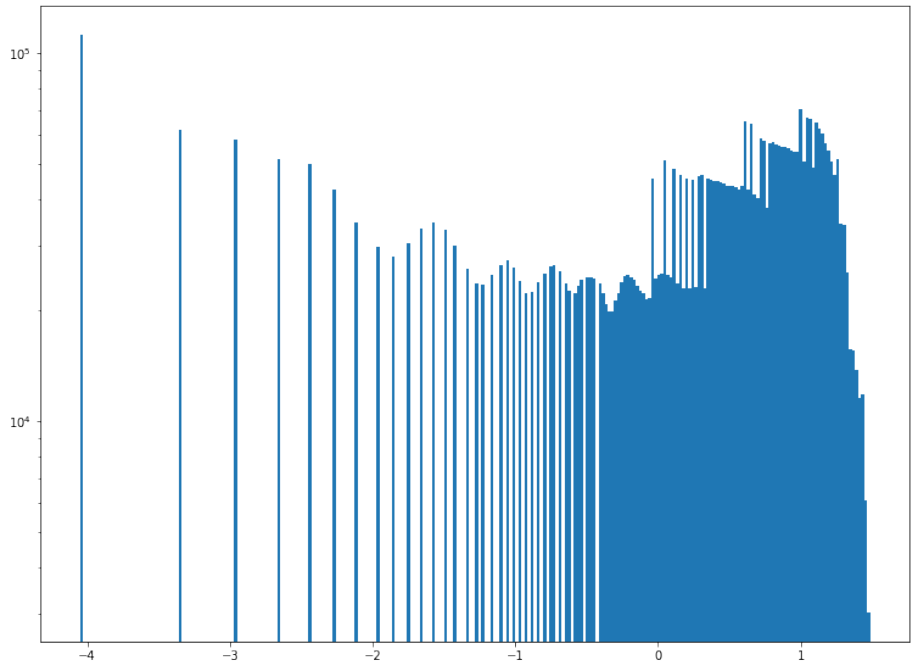
```
plt.hist(v3data.flat, 256)
plt.yscale('log');
```



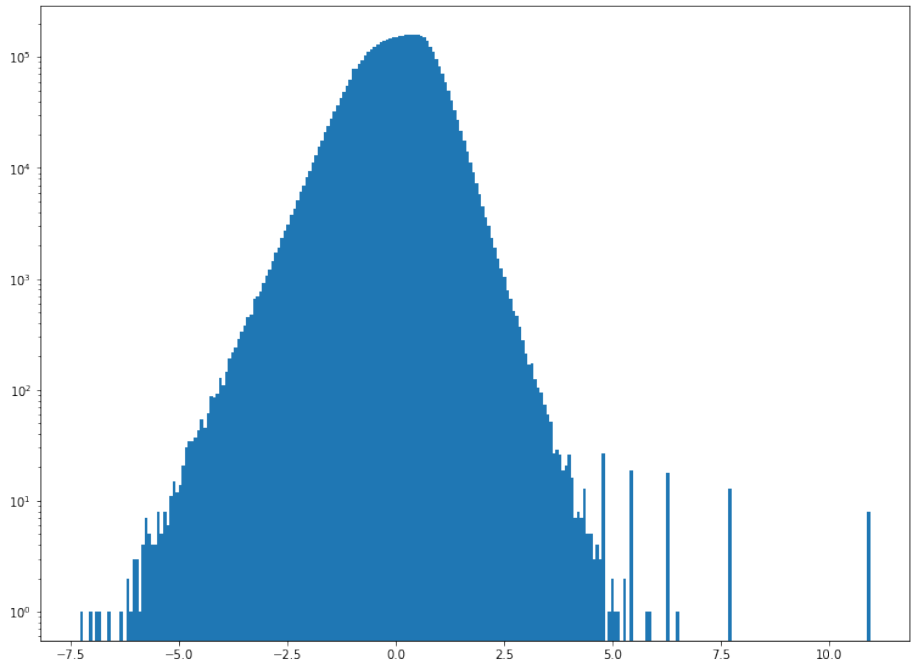

```
plt.hist(vdata.flat, 256)
plt.yscale('log');
```



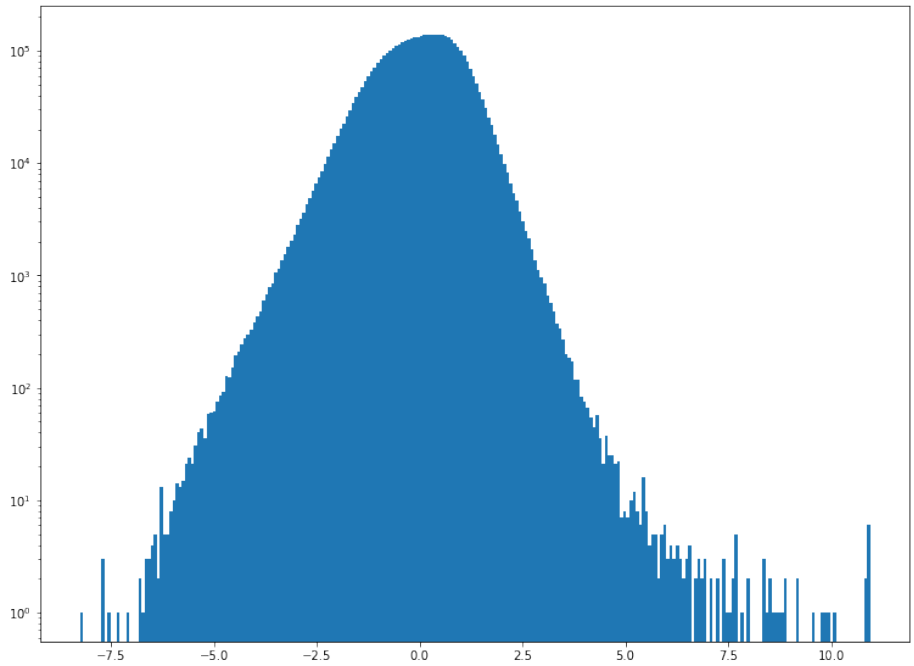

```
plt.hist(ldata.flat, 256)
plt.yscale('log');
```



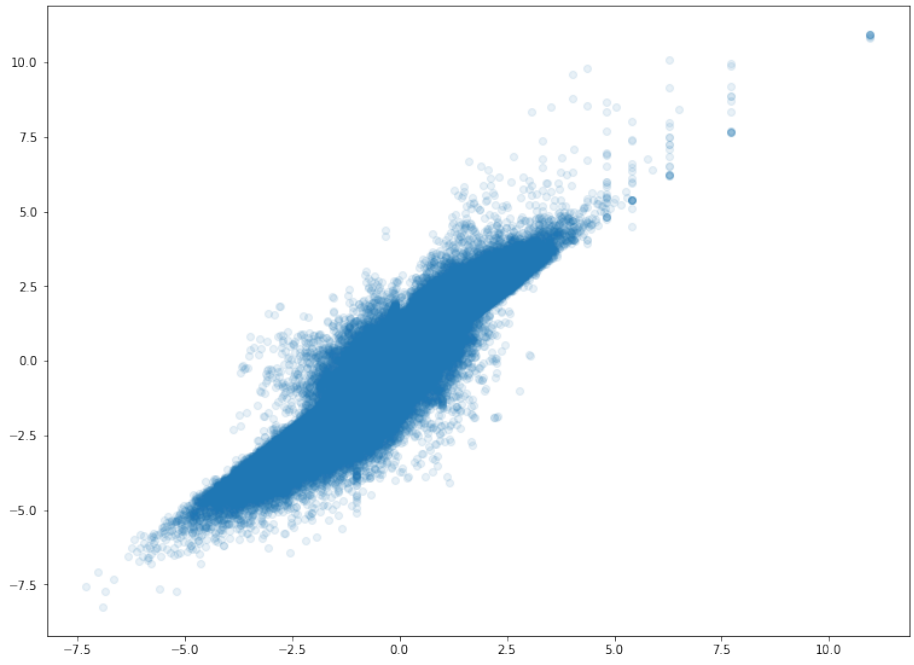
```
plt.hist(l1data.flat, 256)
plt.yscale('log');
```




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




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



30.2 Fractal textures

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

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

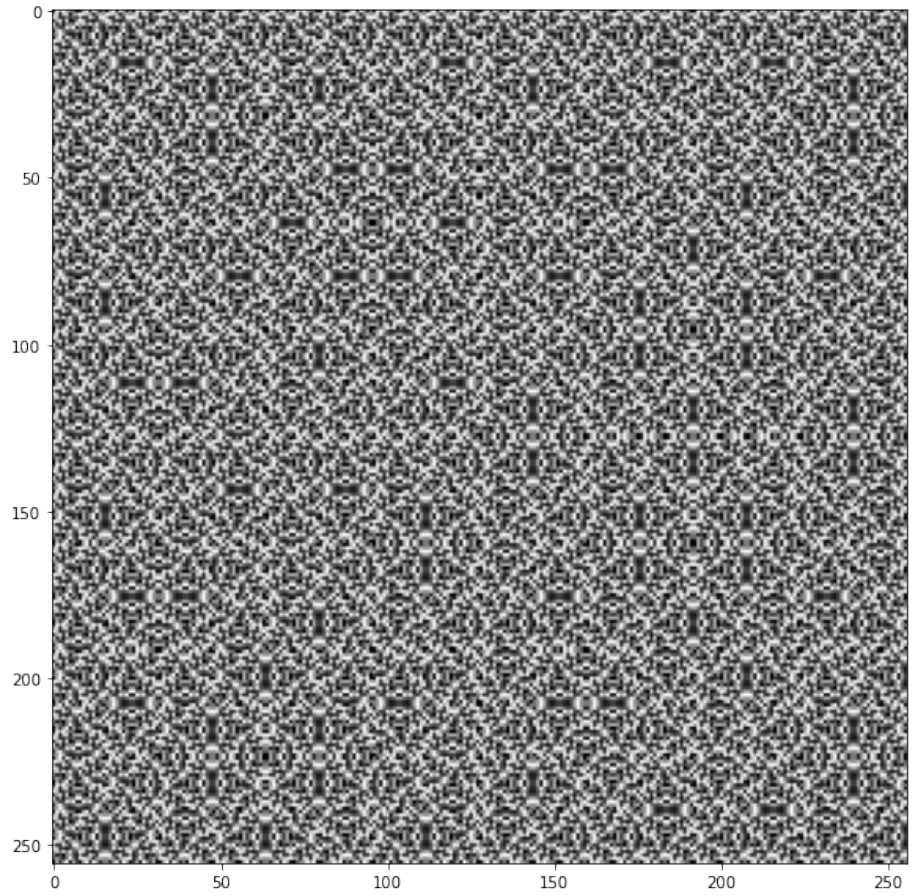
```
np.block([[rep2(b) for b in a] for a in x])
```

```
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)
```

▪

30.3 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.

30.3.1 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
```

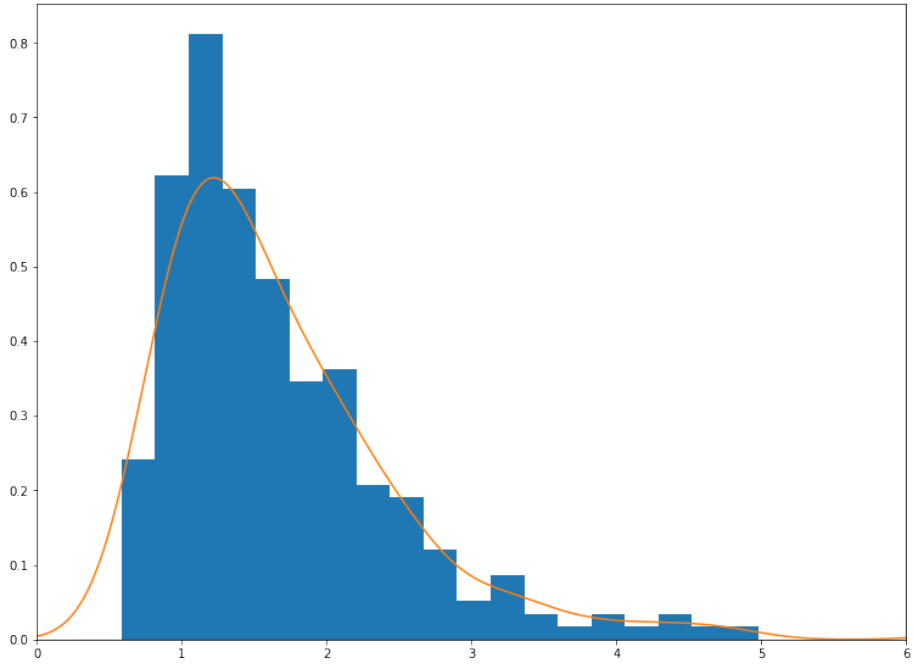
As an example, we sample from a Laplacian distribution with variance 2, but request sets with variance 1. The result is that the variances of the sets are imperfectly nudged away from 2 and towards 1.

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

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

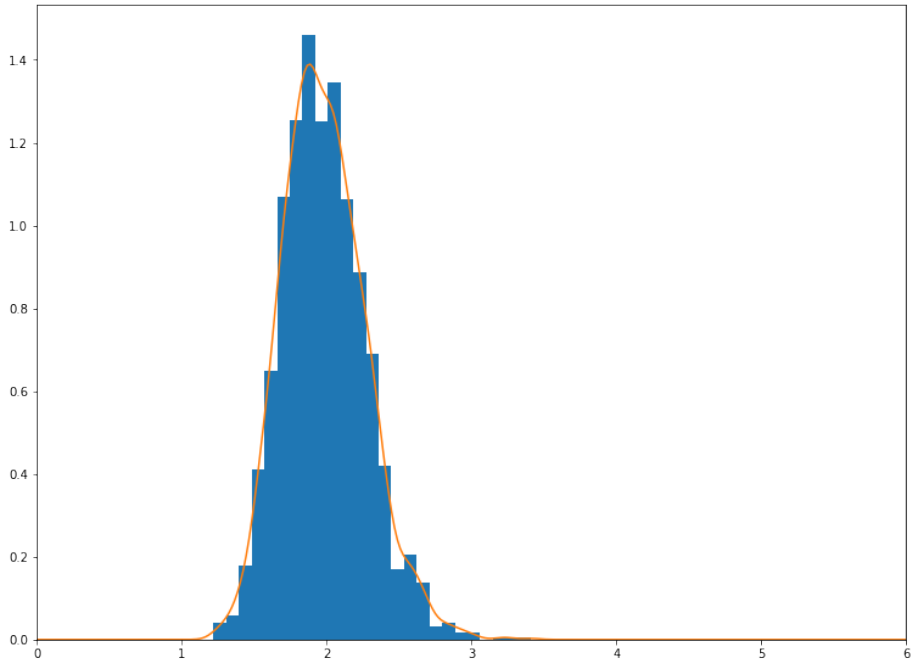
```
1.8887564926323521
```

```
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()
```

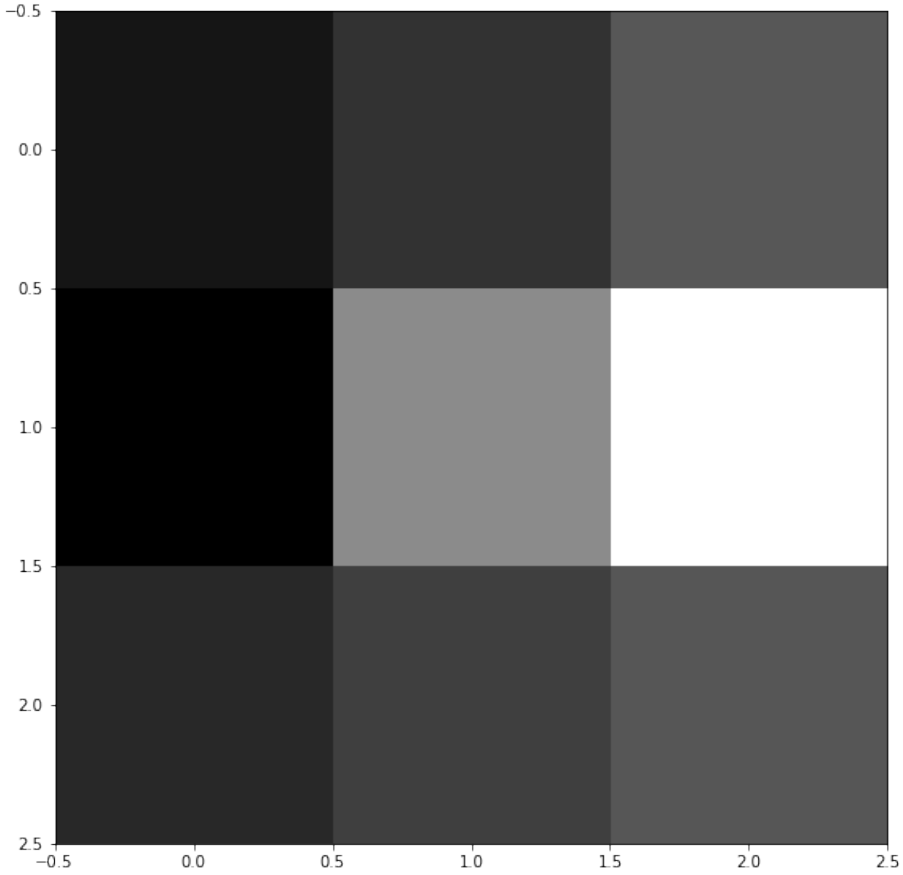


Now what are the distributions of the proposed natural scale-invariant variance images (as in Ruderman's statistics of natural images, doi: 10.1088/0954-898X_5_4_006)? First, we will try an exponential distribution of variances.

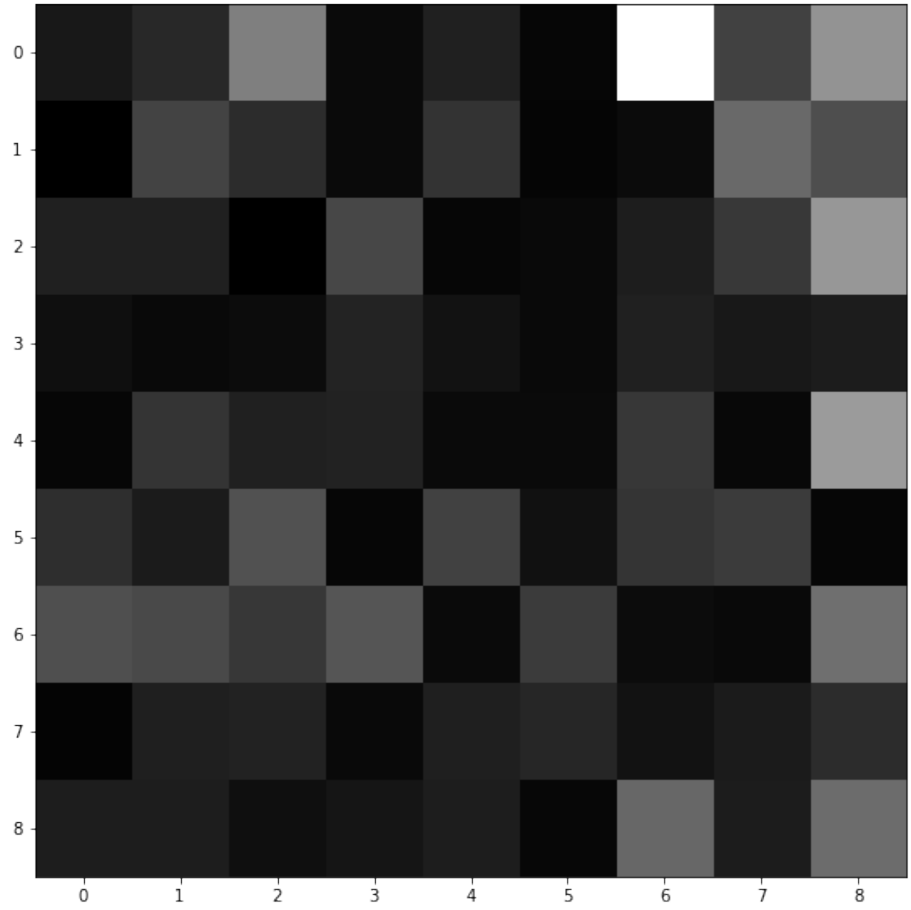
```
n0 = 3
m0 = 3
initial_vars = stats.expon.rvs(size=n0*n0)
inr_vars = inverse_nreduce(initial_vars, m0*m0, np.var, lambda n: (1 / 3) * stats.expon.rvs(size=n))
```

```
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)));
```

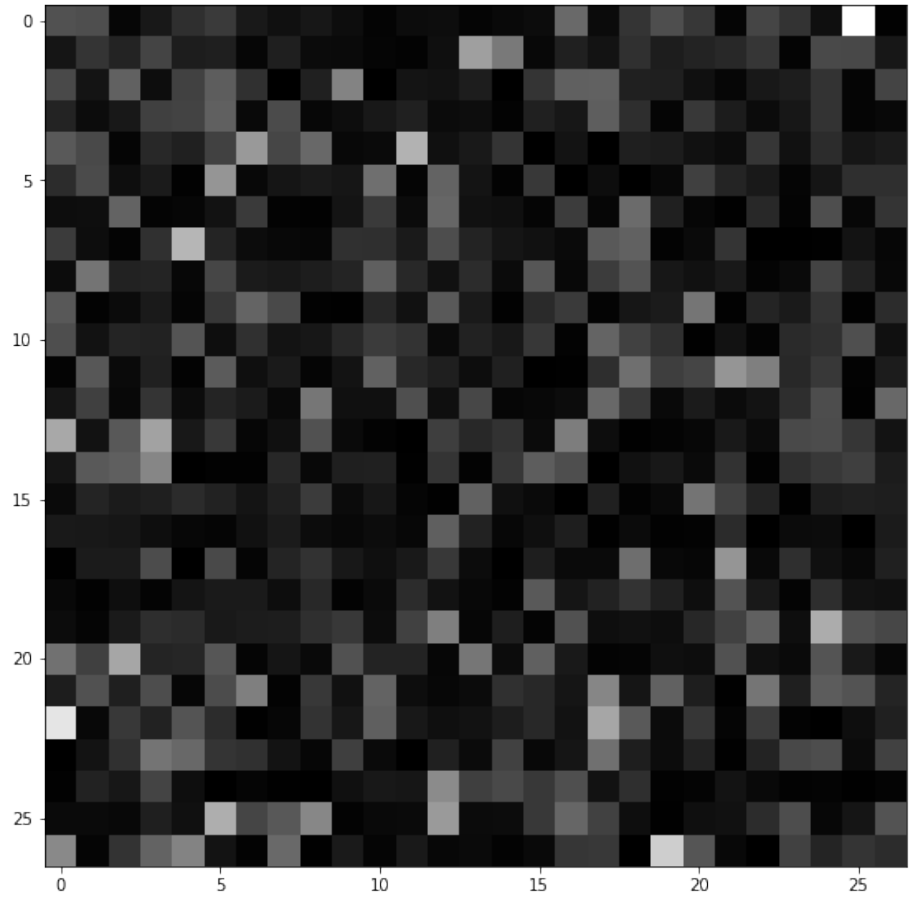



```
plt.imshow(inr_mat);
```



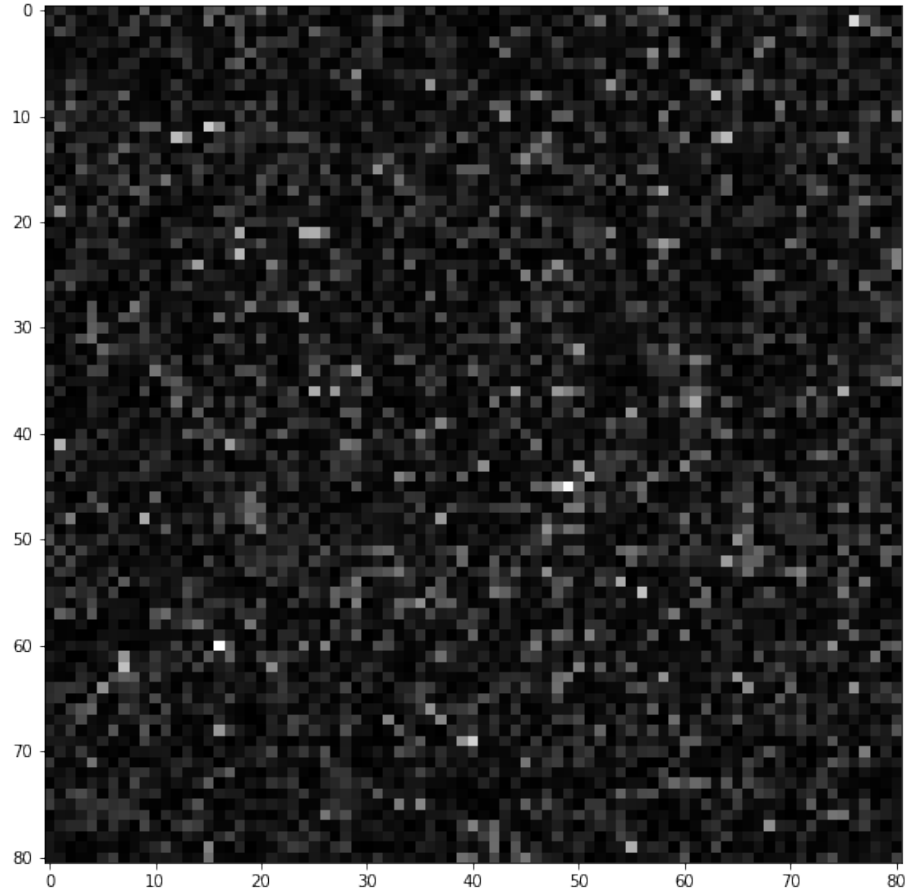

```
inr2_vars = inverse_nreduce(new_vars, m0*m0, np.var, lambda n: (1 / 3) * stats.expon.rvs(size=n))
n1 = int(np.sqrt(np.shape(inr2_vars)[0]))
inr2_mat = np.concatenate([np.concatenate(r, axis=1) for r in np.reshape(inr2_vars, (n1, n1, m0, m0))],
↪  axis=0)
new2_vars = inr2_mat.flat
```

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



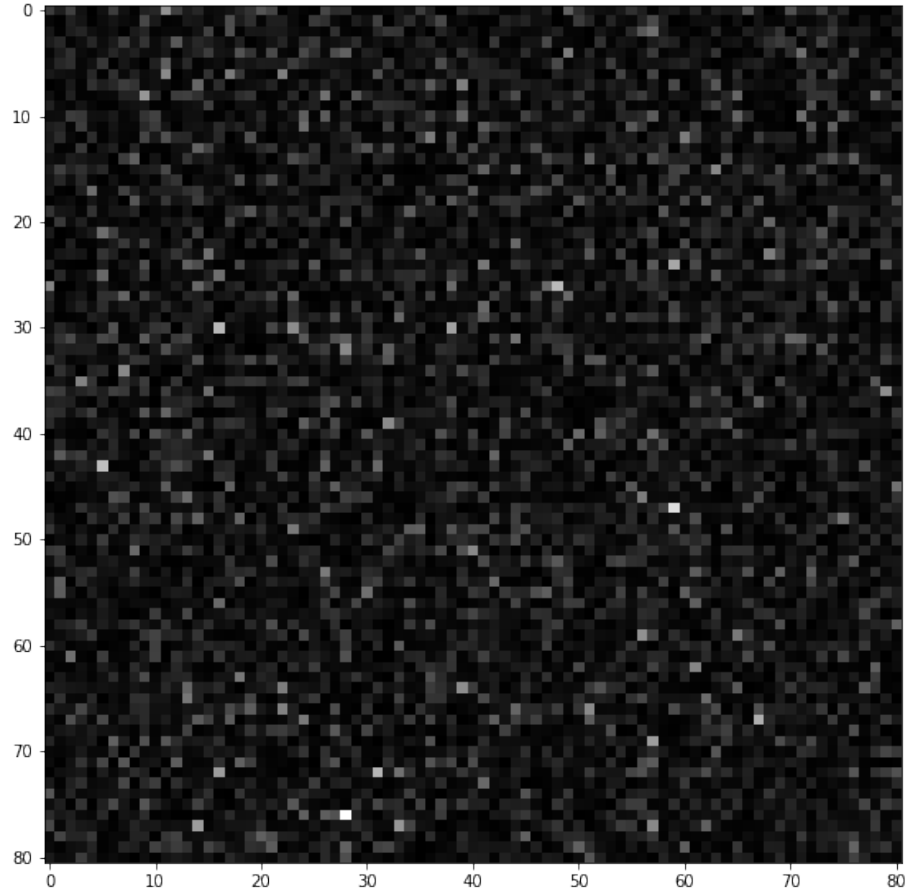

```
inr3_vars = inverse_nreduce(new2_vars, m0*m0, np.var, lambda n: (1 / 3) * stats.expon.rvs(size=n))
n2 = int(np.sqrt(np.shape(inr3_vars)[0]))
inr3_mat = np.concatenate([np.concatenate(r, axis=1) for r in np.reshape(inr3_vars, (n2, n2, m0, m0))],
↪  axis=0)
```

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

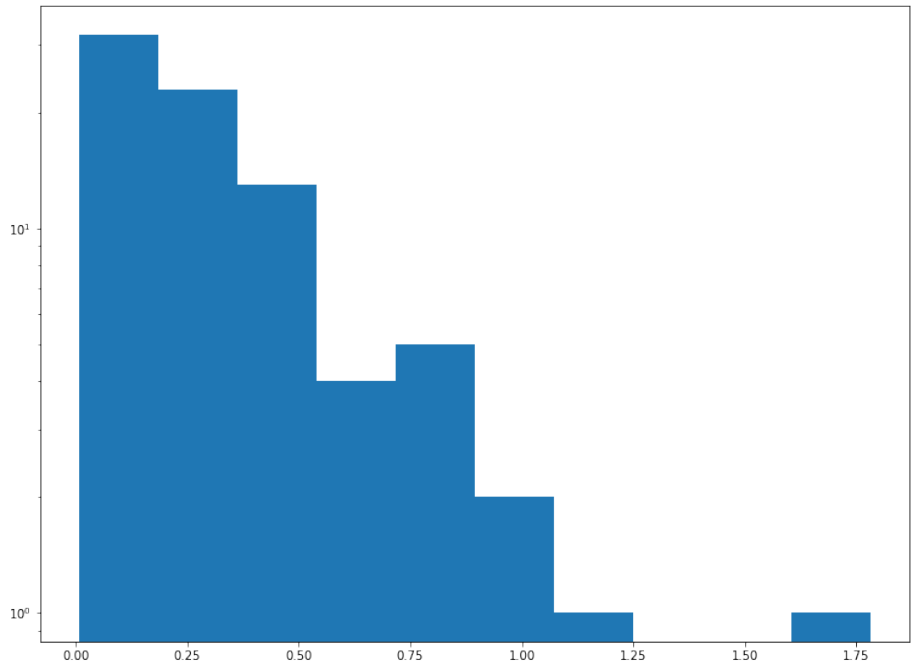



```
control_mat = np.reshape(stats.expon.rvs(size=np.size(inr3_mat)), np.shape(inr3_mat))
```

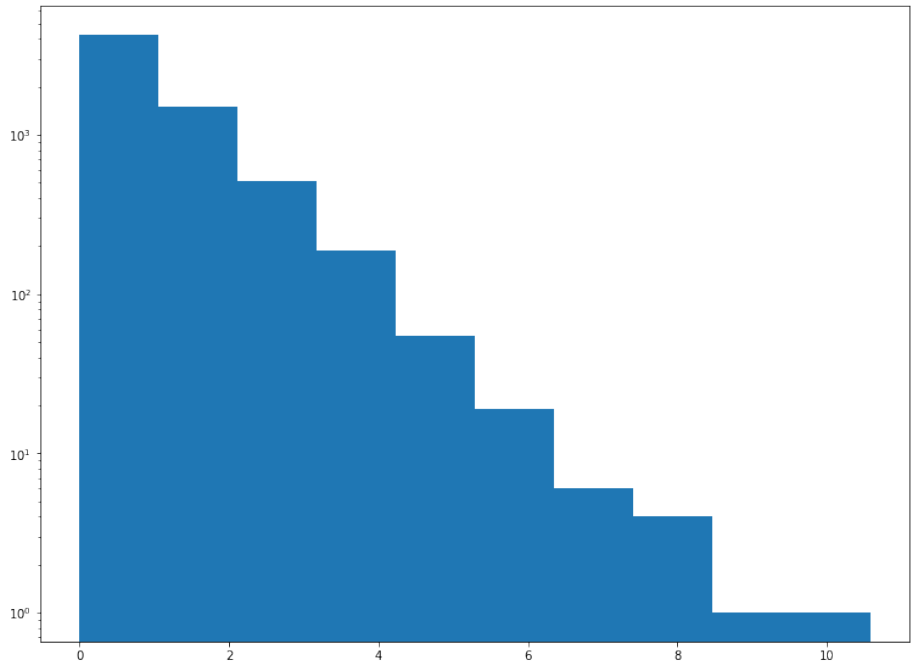
```
plt.imshow(control_mat);
```




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

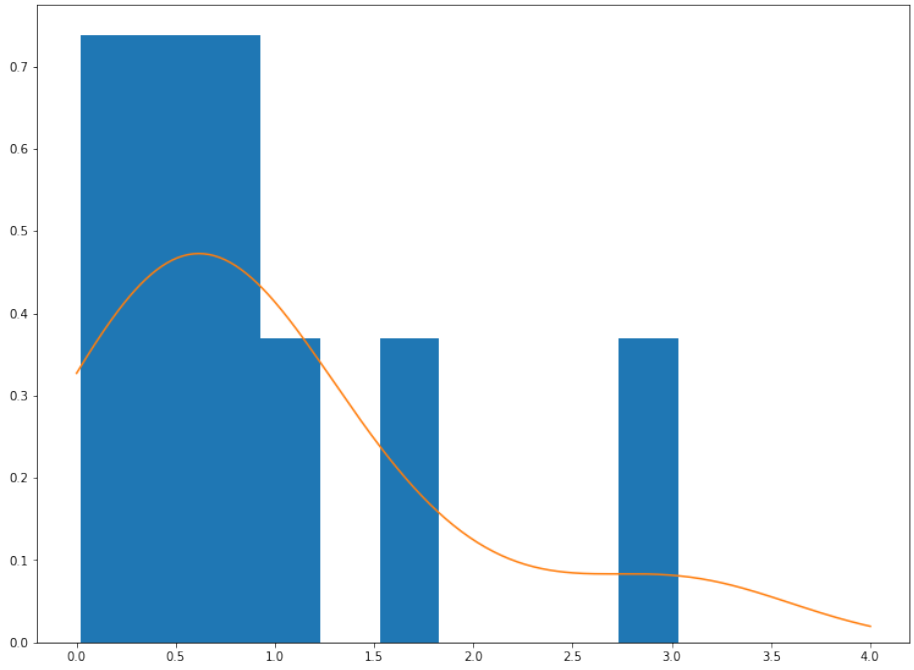



```
plt.hist(control_mat.flat)
plt.yscale('log');
```

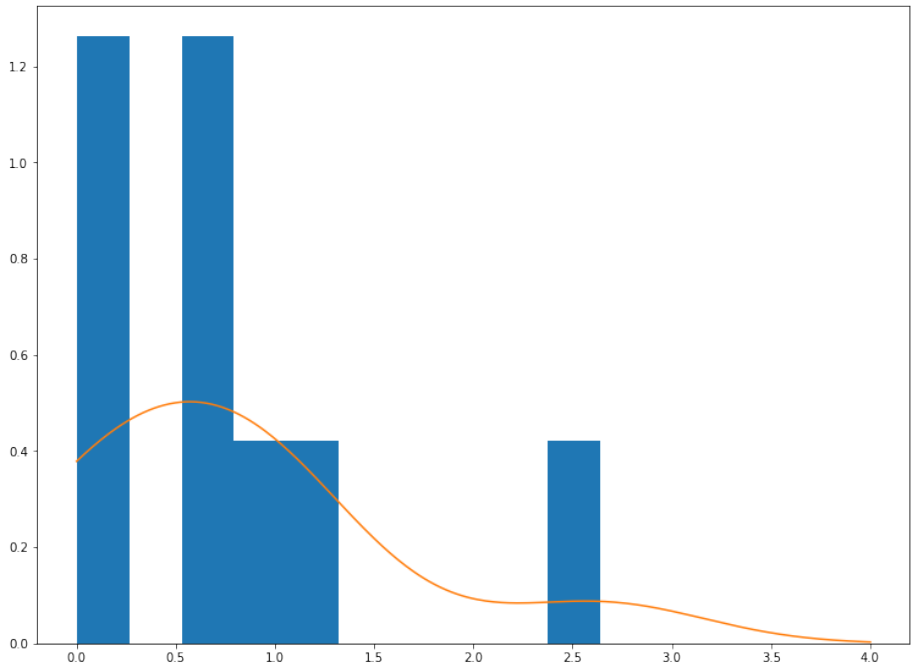


Histograms

```
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()
```

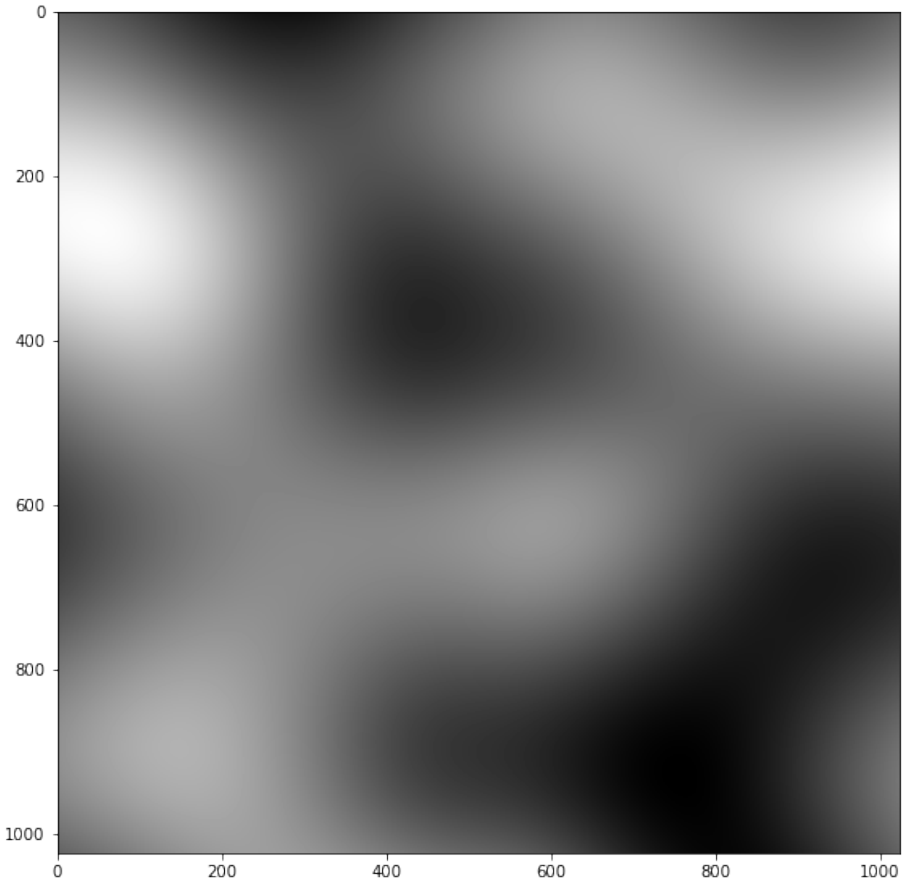


30.4 Perlin noise

```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
```

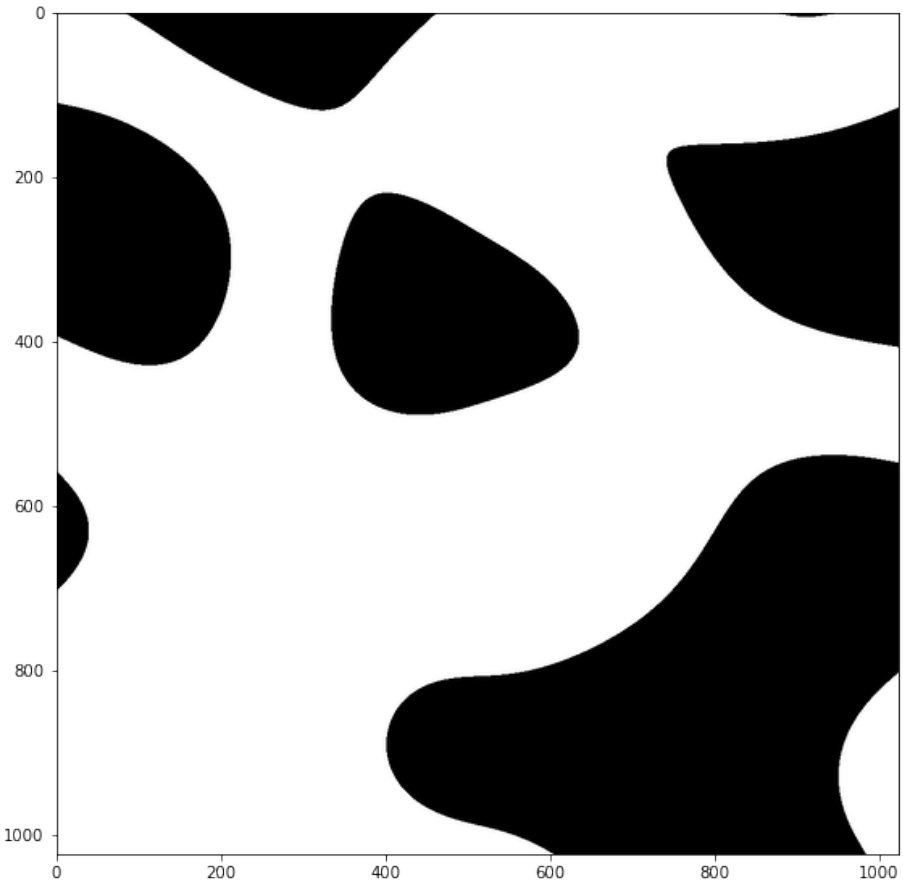
```
    t = f(grid)
    n0 = n00*(1-t[:,:,0]) + t[:,:,0]*n10
    n1 = n01*(1-t[:,:,0]) + t[:,:,0]*n11
    return np.sqrt(2)*((1-t[:,:,1])*n0 + t[:,:,1]*n1)

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');
```

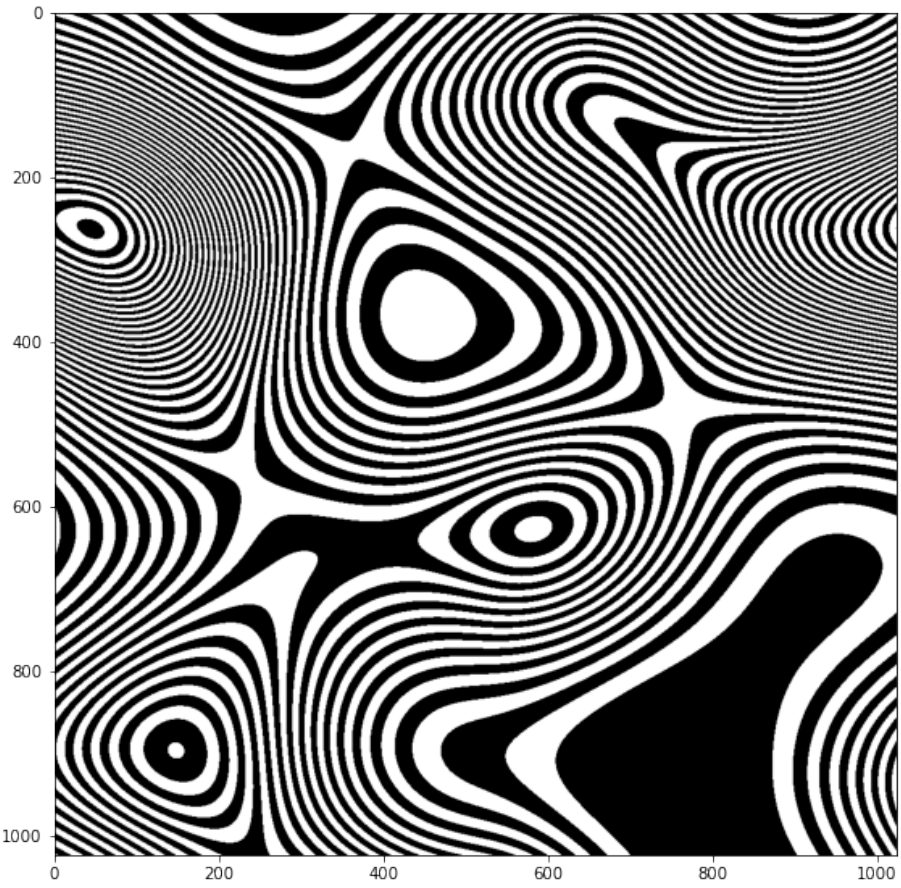



```
pmin, pmax = np.min(pn), np.max(pn)
```

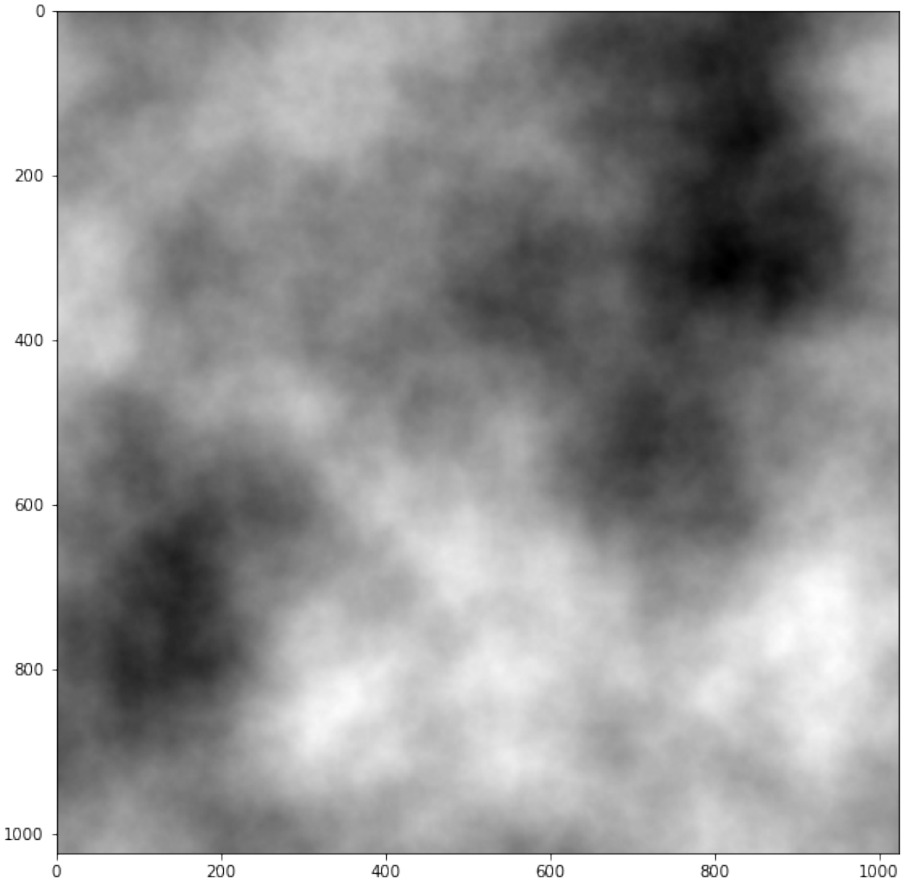
```
pn = (pn - pmin) / (pmax - pmin) # Rescale to exactly [0, 1] for thresholding
plt.imshow((0.3 < pn) & (pn < 0.7), interpolation='lanczos');
```



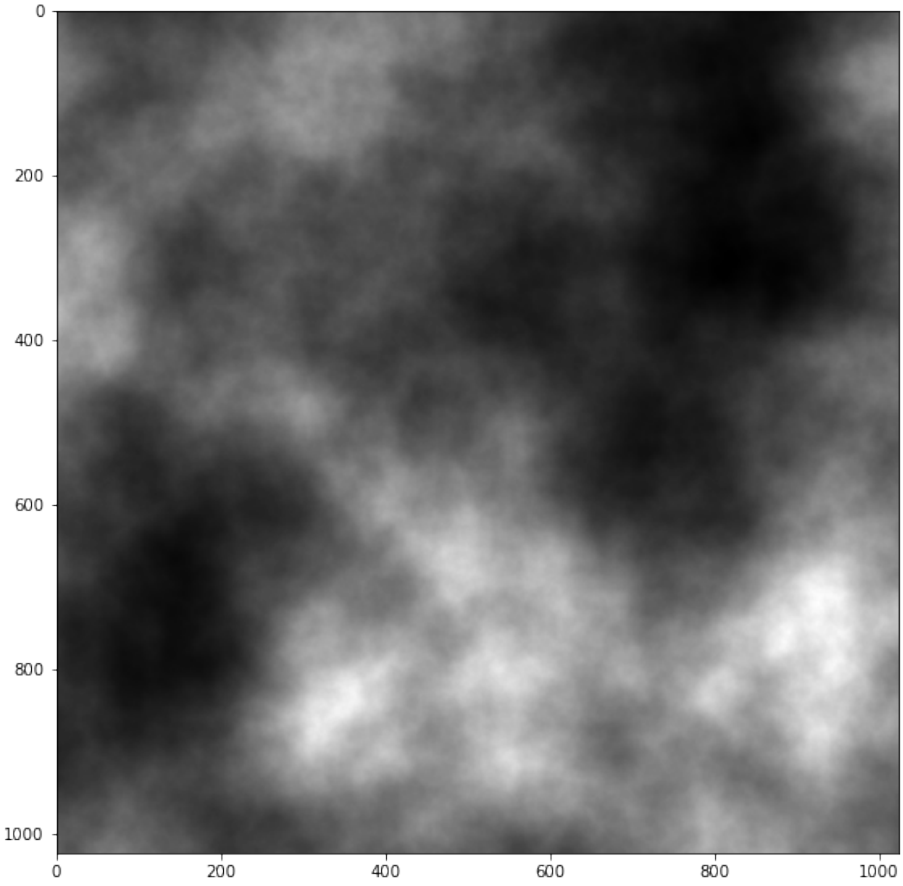

```
plt.imshow(np.floor((pn**2)*64) % 2);
```



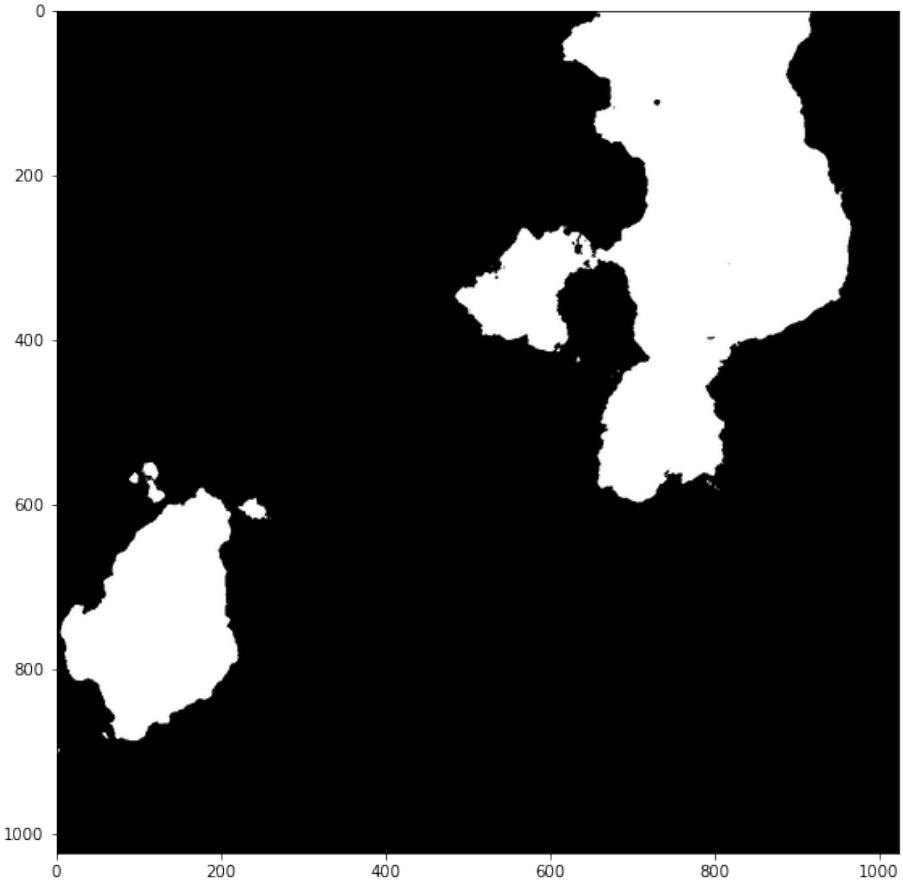

```
pfn = generate_fractal_noise_2d((1024, 1024), (2, 2), octaves=10, persistence=0.5)
pfn = 0.5 + (pfn / 4) # Rescale to within [0, 1]
plt.imshow(pfn, interpolation='lanczos');
```



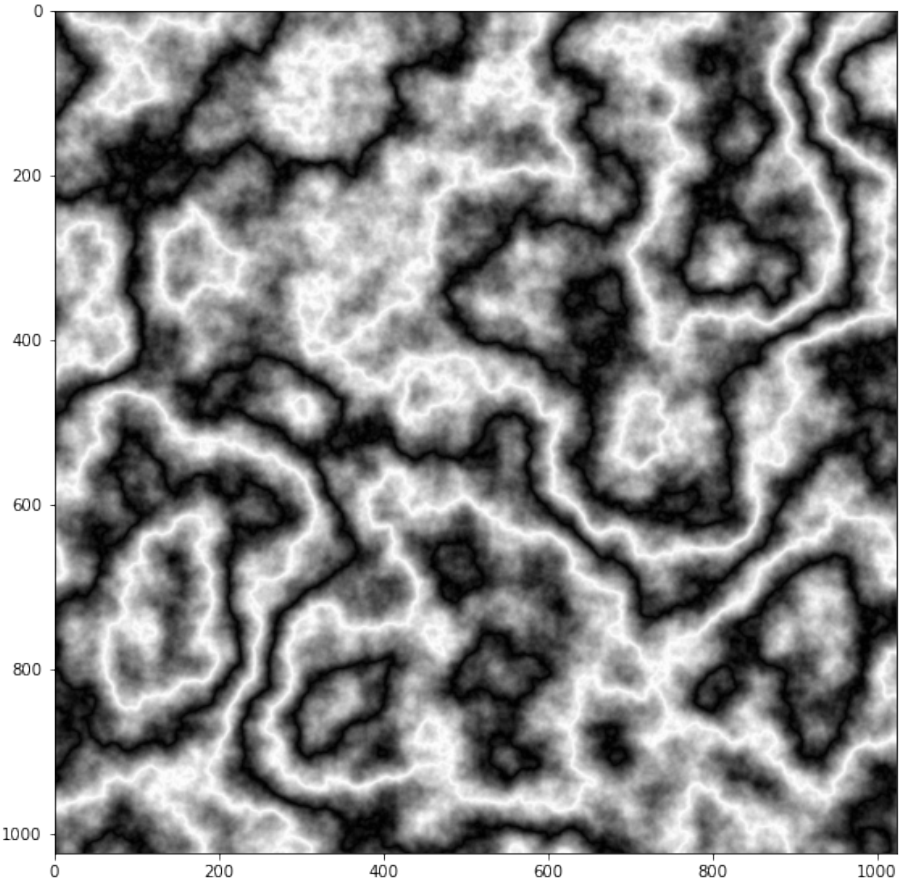

```
plt.imshow(pfn**3, interpolation='lanczos');
```



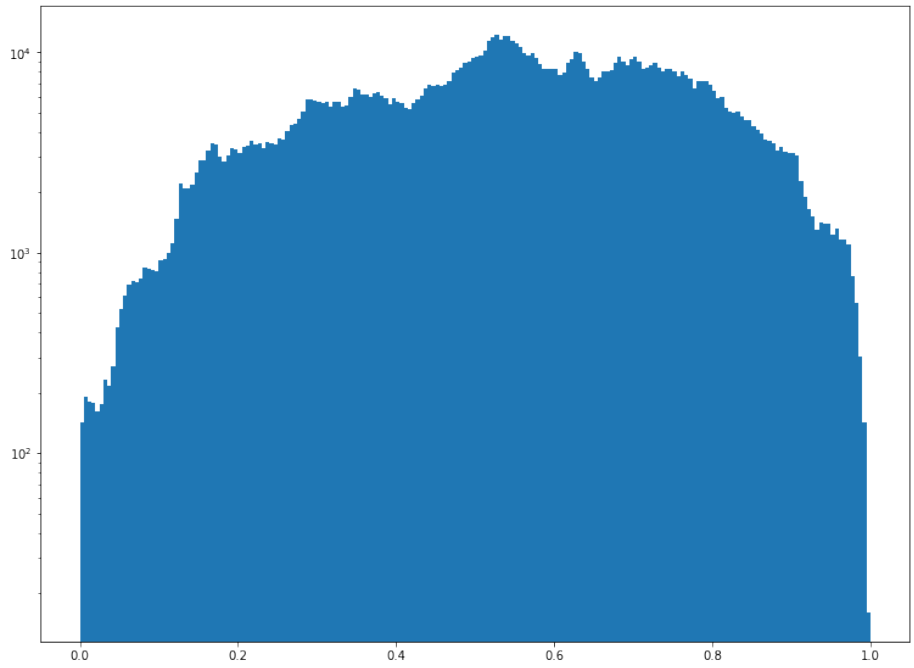

```
pmin, pmax = np.min(pfn), np.max(pfn)
pfn = (pfn - pmin) / (pmax - pmin) # Rescale to exactly [0, 1] for thresholding
plt.imshow((0.0 < pfn) & (pfn < 1/3), interpolation='lanczos');
```




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




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



31 Progress summary

July 17, 2020 This week, I continued the literature search and have started to implement an algorithm for constructing images with the variance statistics of natural images by inverting a neighborhood operation. Since the computational complexity of one step is $O(mn^3)$, where m is the number of pixels in a block and n is the number of pixels in the starting reduced image, we are limited to images with only hundreds of pixels per side. I have also considered some Perlin noise based techniques for generating fractal noise and then natural images.