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 $\mathcal{X}$, the *Shannon entropy* is

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

The *intensity-level entropy* is the Shannon entropy of the empirical distribution of intensity values.

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

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

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)
```

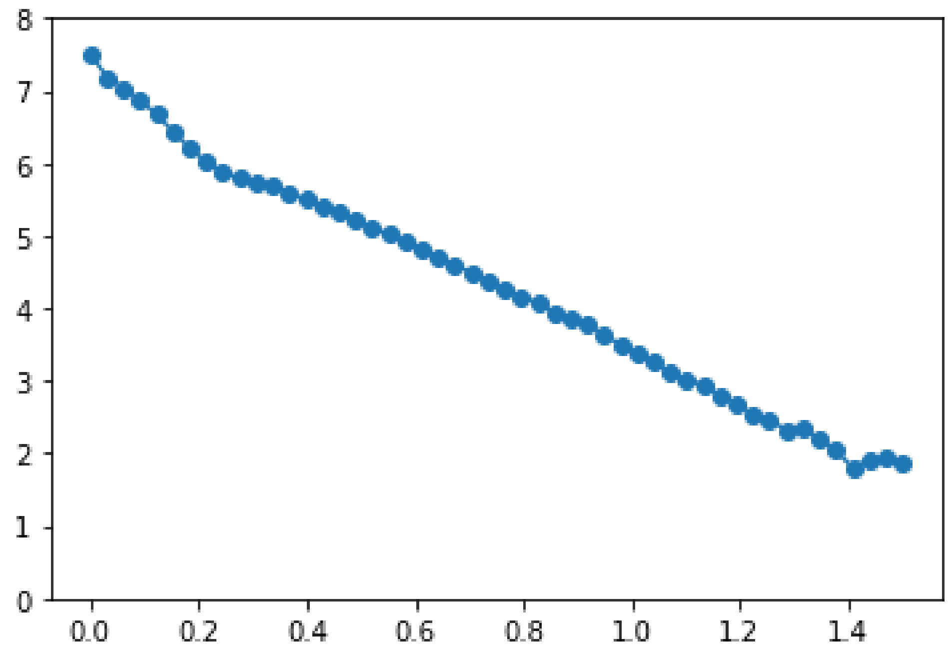
7.51132356216608

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

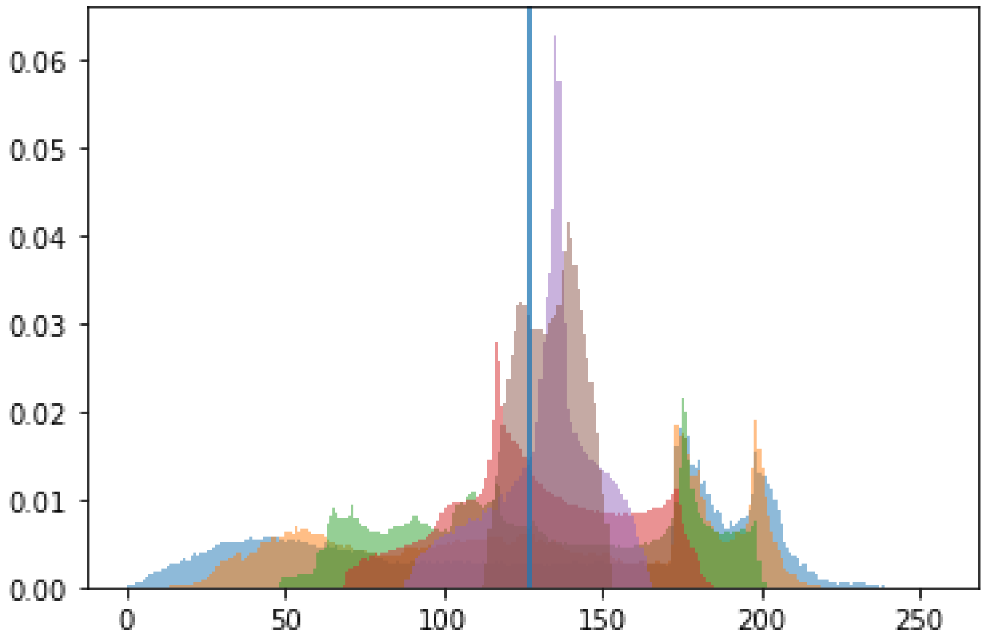
```
def intensity_blur(img, scales, display=True):
    scale = max(np.shape(img))

    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
```

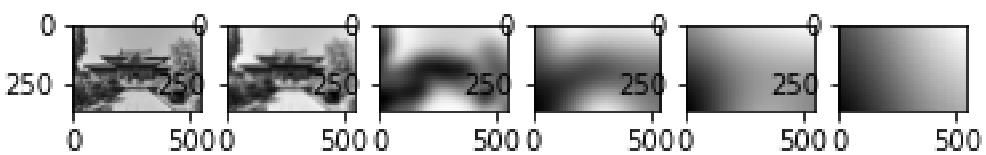
```
results = intensity_blur(img, np.linspace(0, 1.5, num=50), False)

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()
```

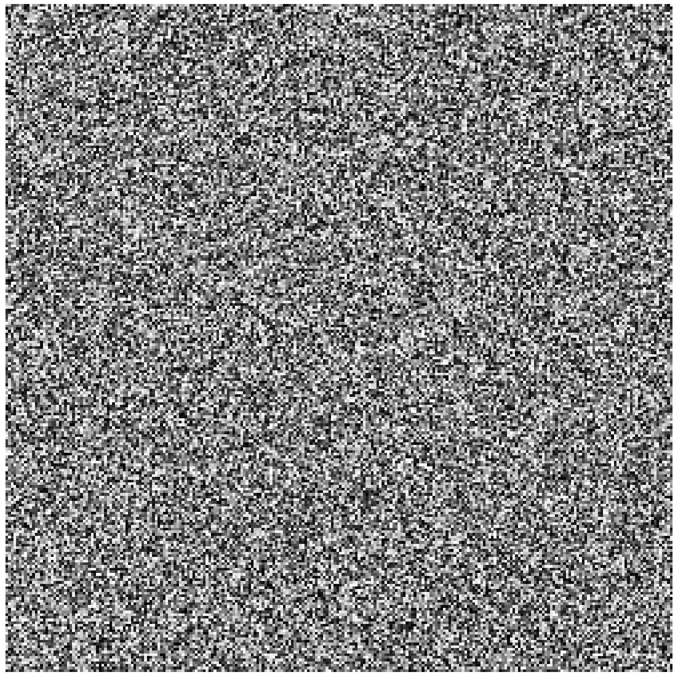



```python
_, axarr = plt.subplots(1, len(rimgs))
for i, subimg in enumerate(rimgs):
    axarr[i].imshow(subimg, cmap='gray')
plt.show()
```



4.2 Random pixel values

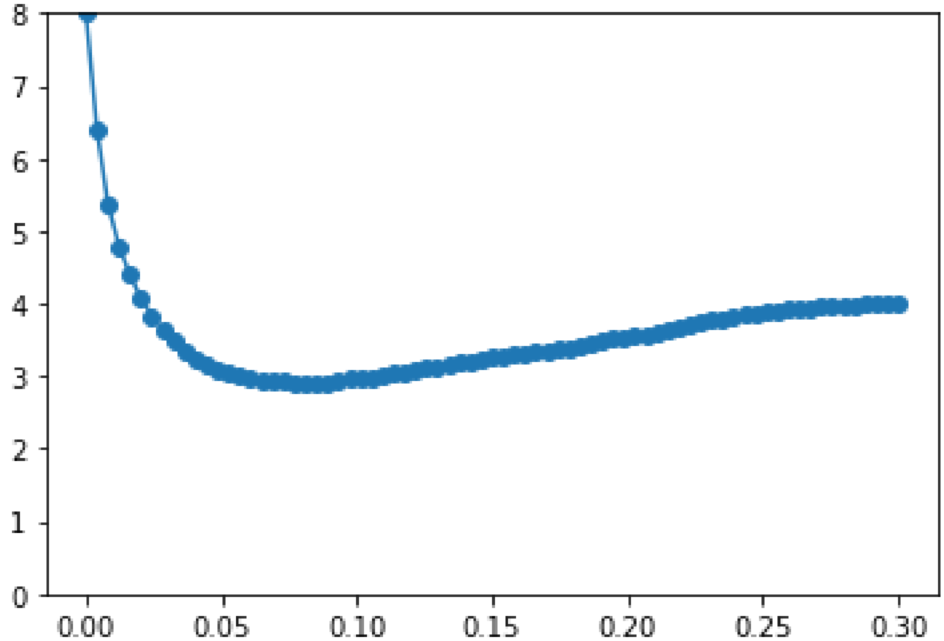
```python
rsize = 250
randimg = Image.fromarray((256*np.random.rand(*2*[rsize])).astype('uint8'))
randimg
```



4.2.1 Beware: GIGO

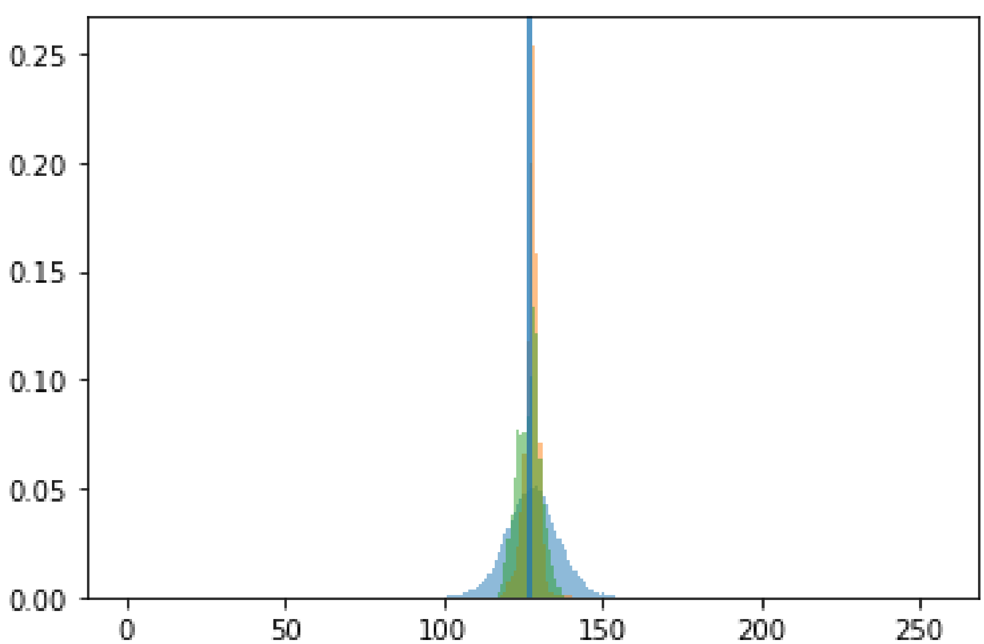
The boundary effects and discrete kernel of `ImageFilter.GaussianBlur` renders the data unreliable after the "minimum" of the intensity entropy with smoothing. This is immediately clear after even small smoothing for random pixel values, since there are no spatial correlations.

```
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)"
```

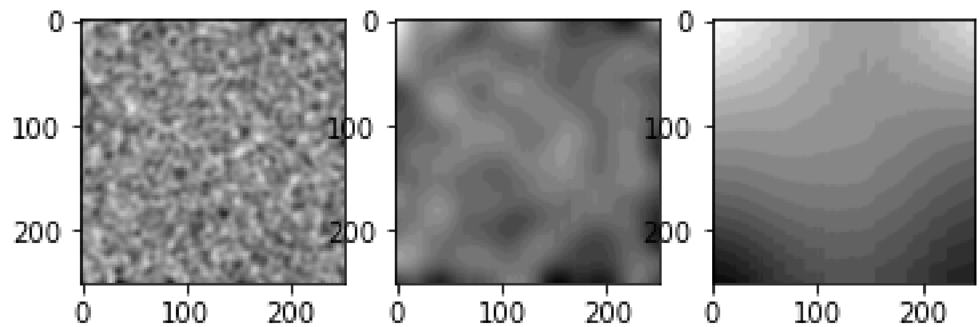



```python
rimgs = [img for _, img, _, _ in intensity_blur(randimg, [0.01, 0.05, 0.25])]
```

```python
plt.show()
```



```python
_, 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)?

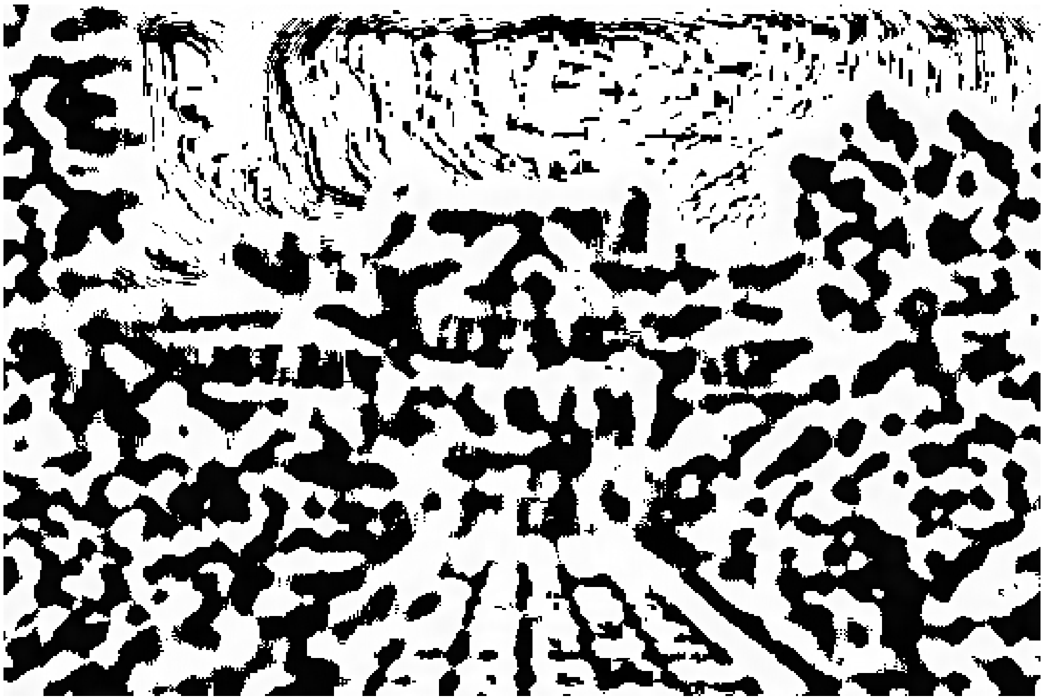
```python
nsmooths = 10
cimg = img
oneimg = cimg.filter(ImageFilter.GaussianBlur(np.sqrt(nsmooths)*2))
oneimg
```

```python
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 $I : X \times Y \longrightarrow \mathbb{Z}_n$, 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 $s : \mathbb{Z}_n \longrightarrow S$ 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.

```
import numpy as np
```

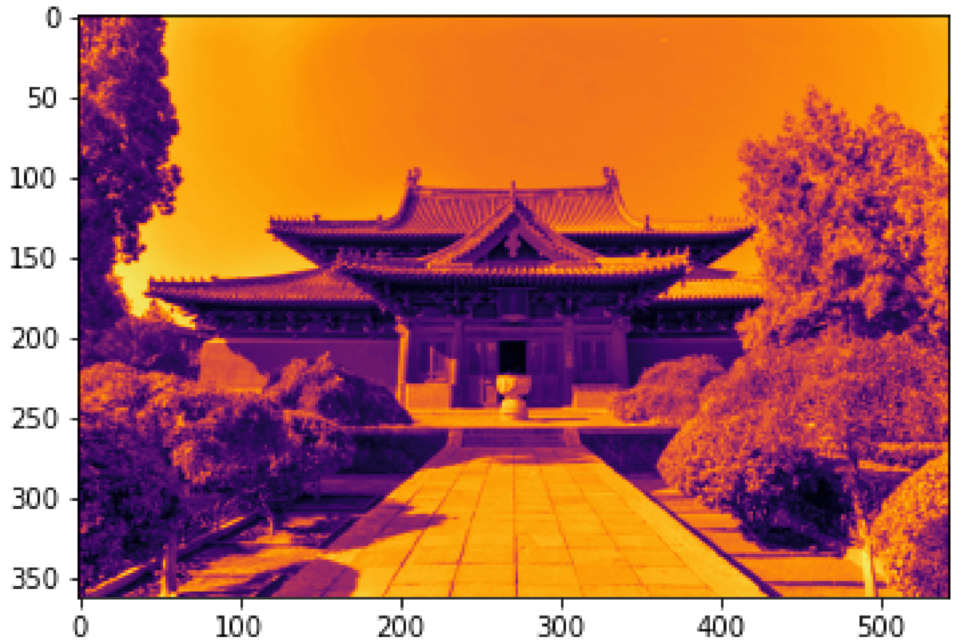
```python
def box(x, y, r):
    return np.s_[max(0, x-r) : x+r+1, max(0, y-r) : y+r+1]
def mapbox(r, f, a):
    return np.reshape([f(a[box(*i, r)]) for i in np.ndindex(np.shape(a))], np.shape(a))
def mapboxes(rs, f, a):
    return (mapbox(r, f, a) for r in rs)
def mapallboxes(f, a):
    return mapboxes(range(max(np.shape(a))),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

```python
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'] = 'inferno'
```

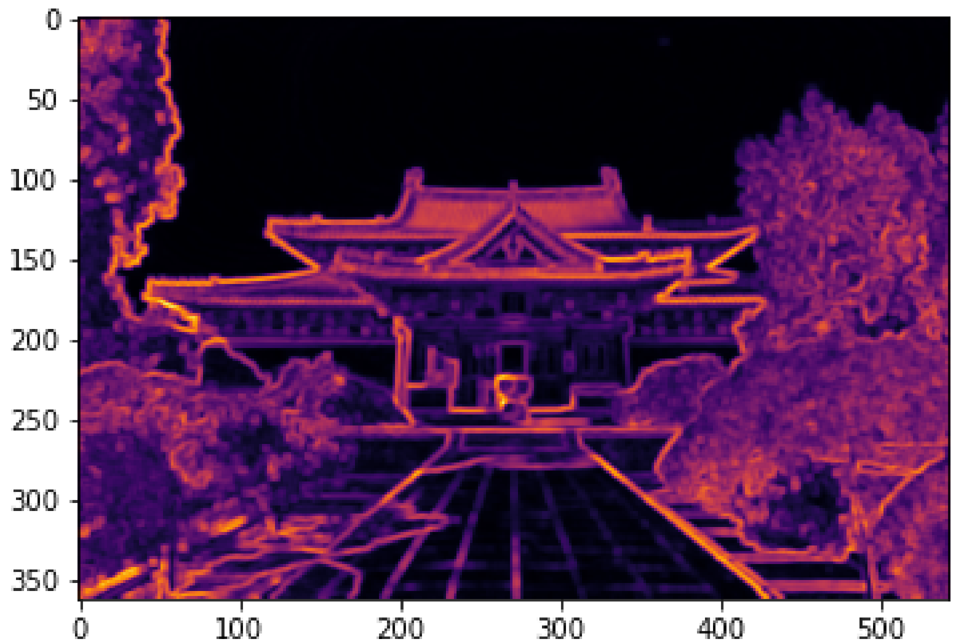
Let's compare the boxcar images for intensity entropy to those for a positive function on an image (the standard deviation) and for different functions of the induced intensity distribution.

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



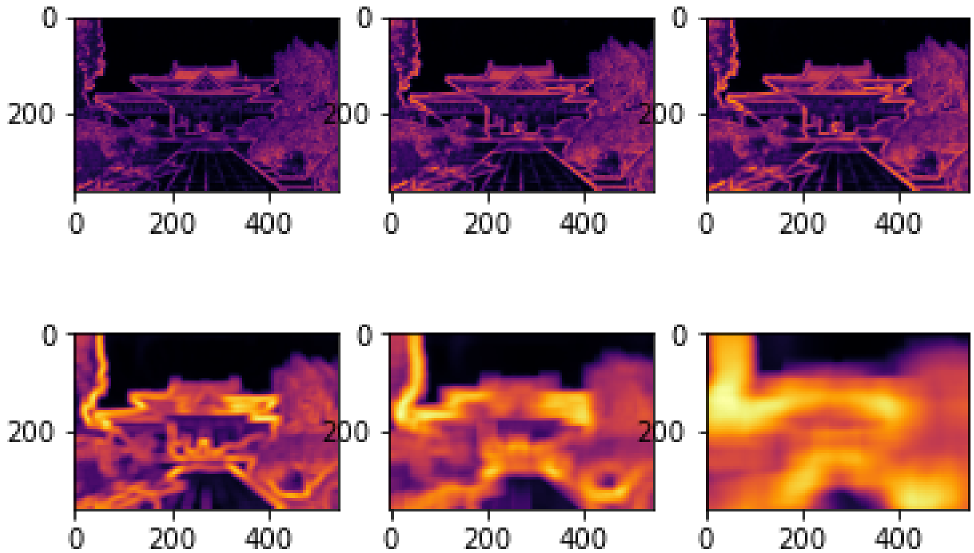
7.1 Standard deviation

```
plt.imshow(mapbox(2, np.std, np.array(img)));
```



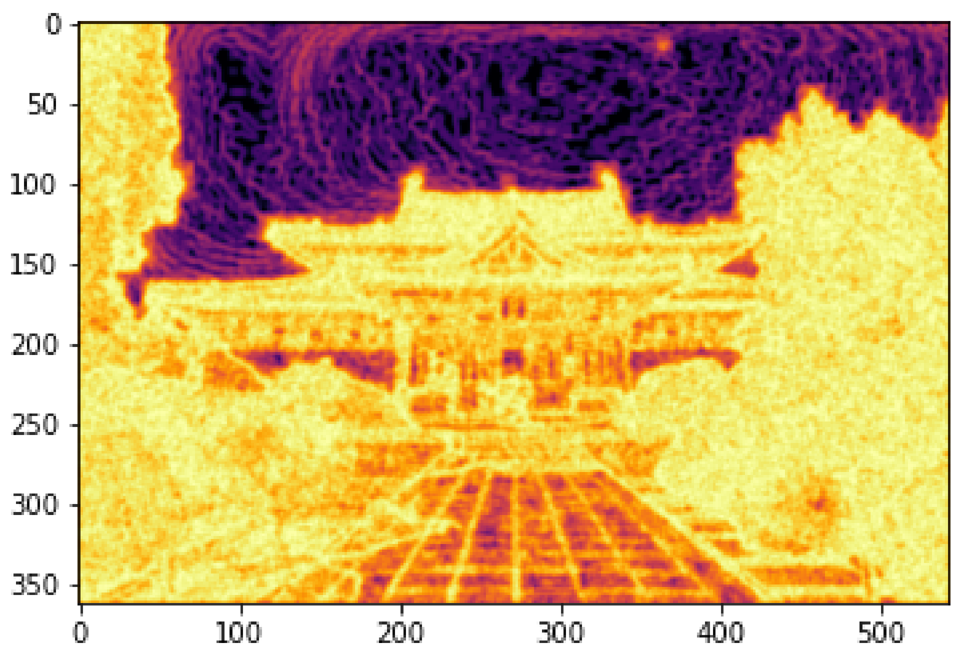

```
boxσs = list(mapboxes([1,2,3,10,20,50], np.std, np.array(img)))
```

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



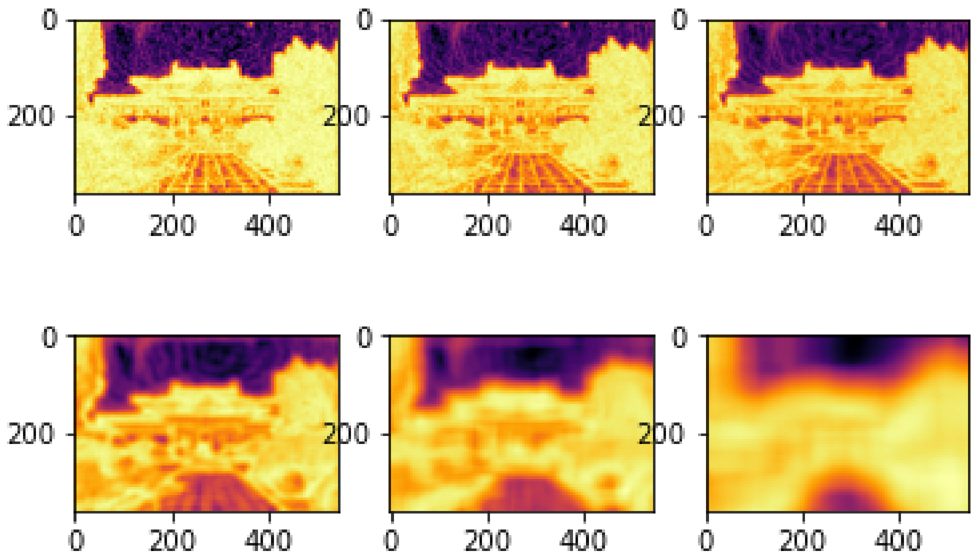
7.2 Intensity entropy

```
plt.imshow(mapbox(2, intensity_entropy, np.array(img)));
```



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

```python
_, 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()
```

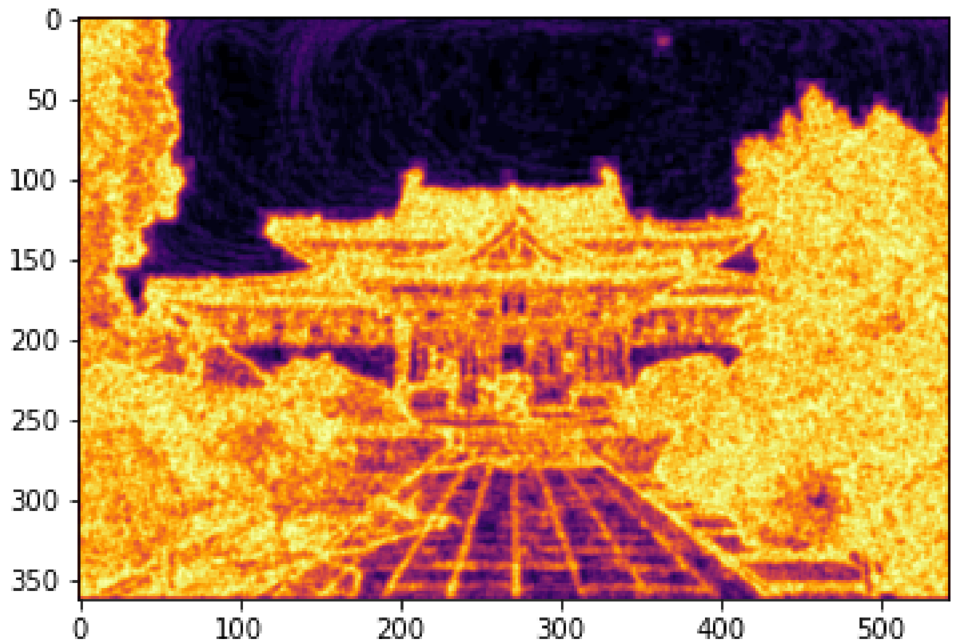


7.3 Replace surprisal with other functions

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.

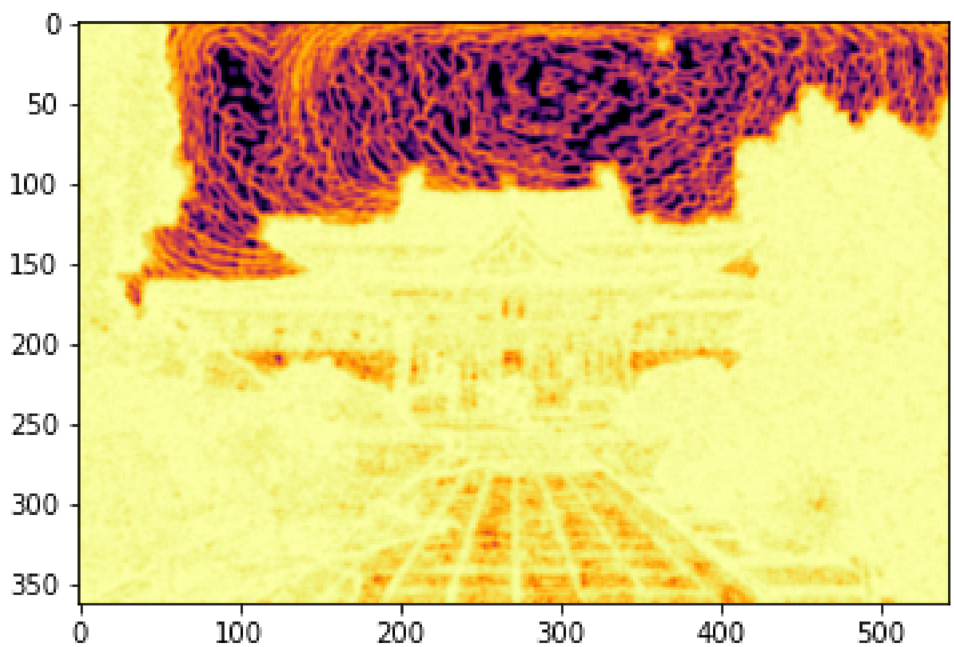
Laurent: $p \mapsto -1 + 1/p$.

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



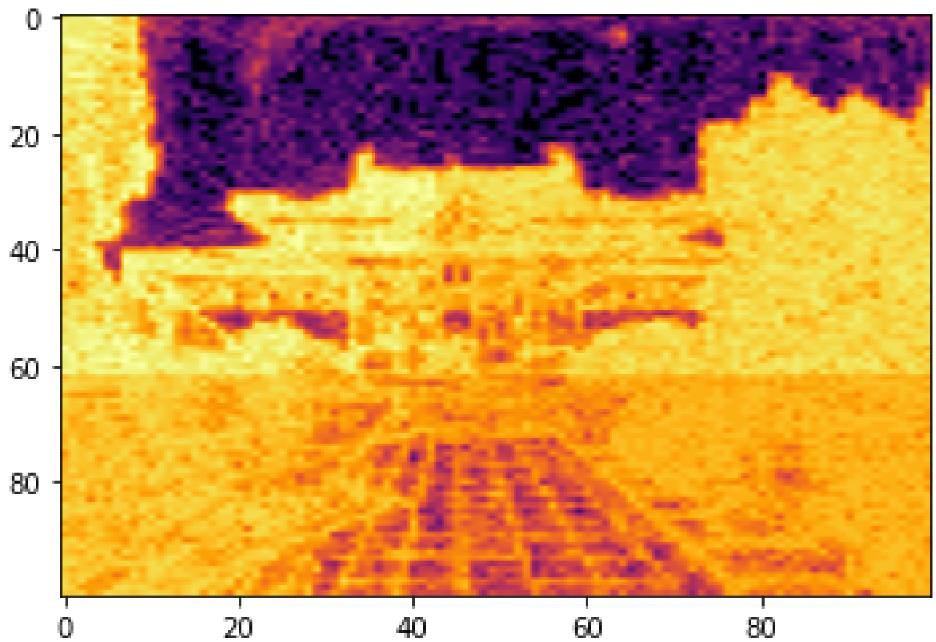
Taylor: $p \mapsto -(1+p)$.

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

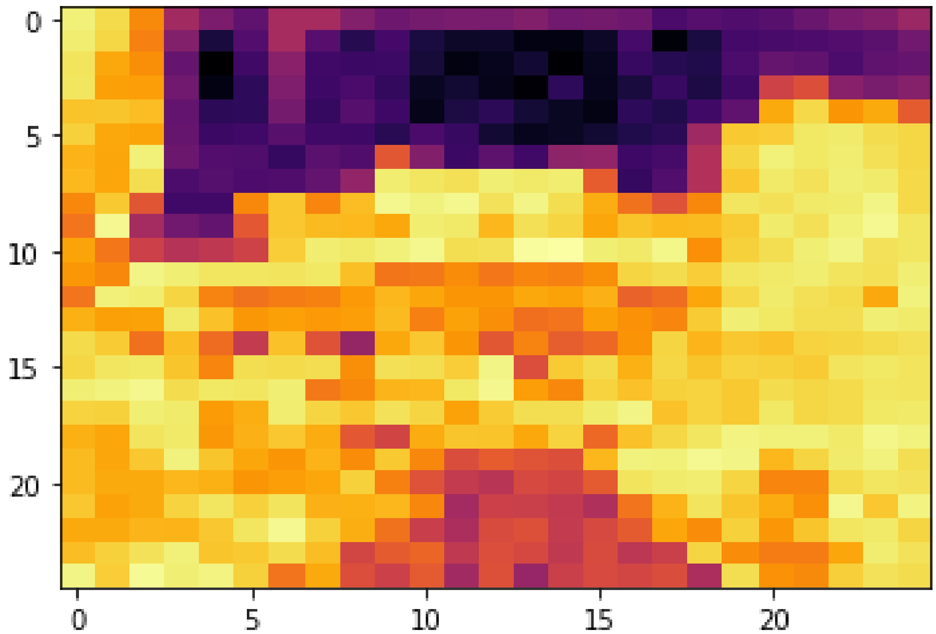


7.4 Intensity entropy on disjoint blocks

```
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 $\mathcal{X}$ is

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

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'
```

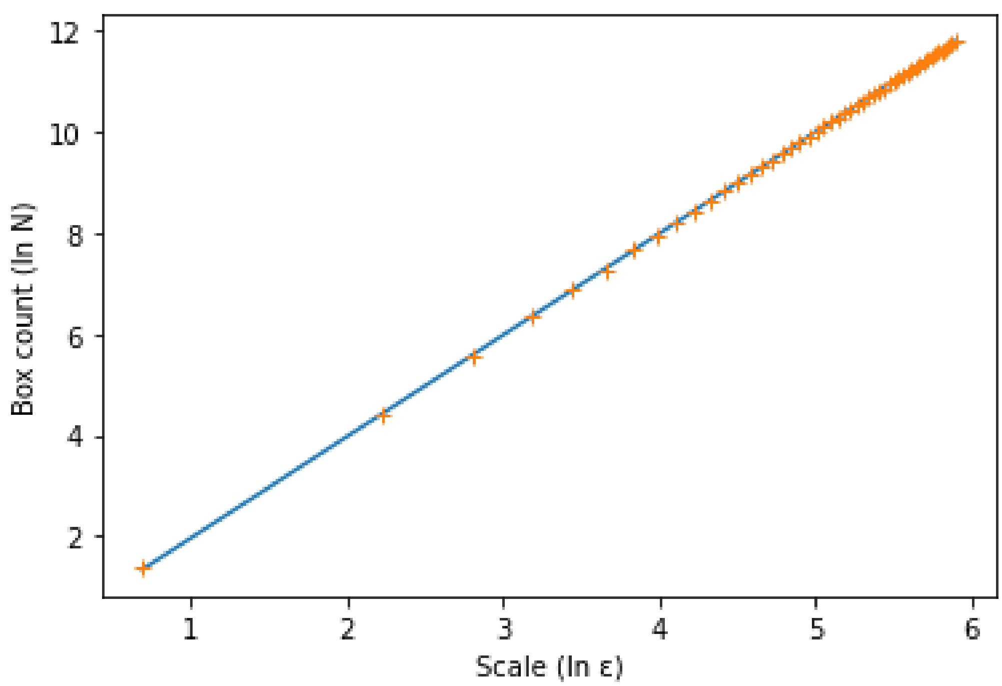
```
img = ImageOps.grayscale(Image.open('test.jpg'))
scale = max(np.shape(img))
data = np.array(img)
img
```

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)
```

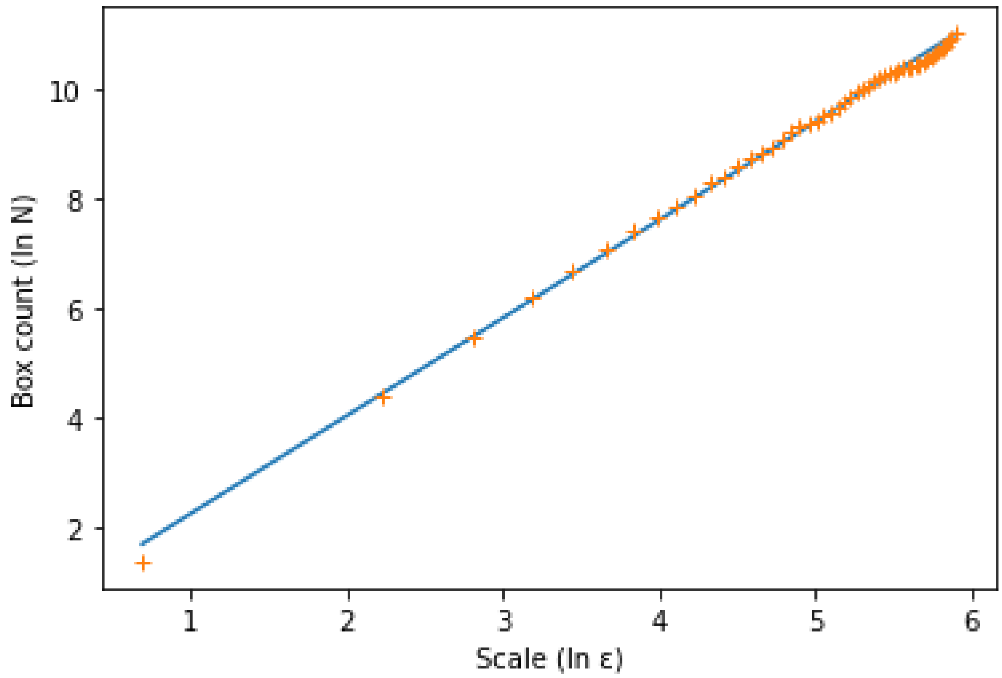
```
2.0087040269581435
```



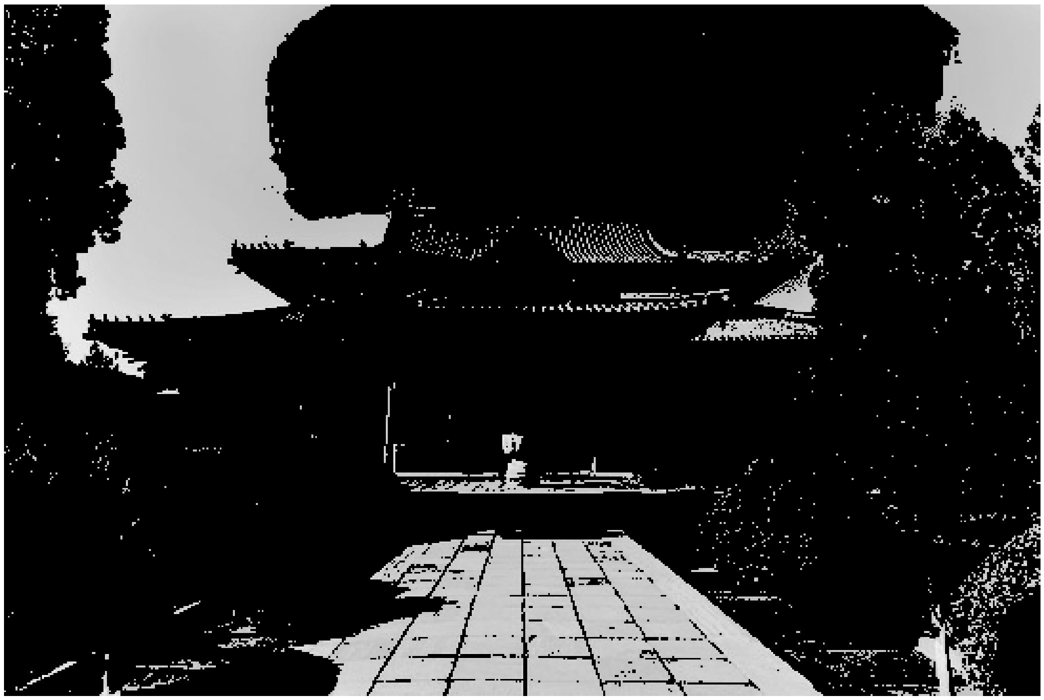

```
sky = data.copy()
sky[sky < 128+32] = 0
Image.fromarray(sky)
```

```
boxdim(sky)
```

```
1.7877778191348215
```

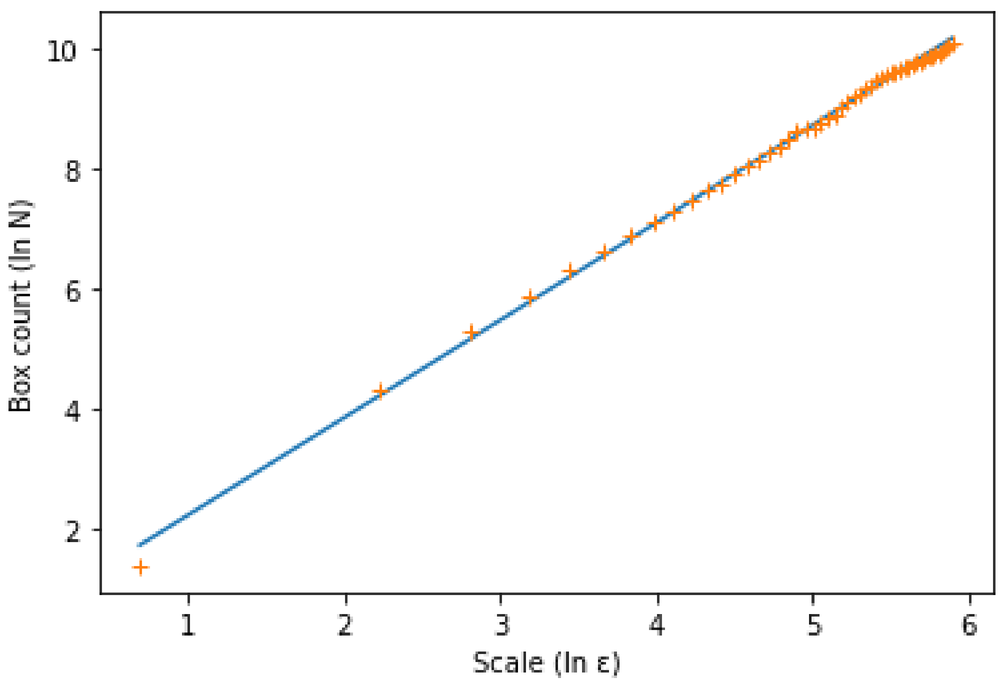



```
nosky = data.copy()
nosky[nosky < 128+64] = 0
Image.fromarray(nosky)
```



```
boxdim(nosky)
```

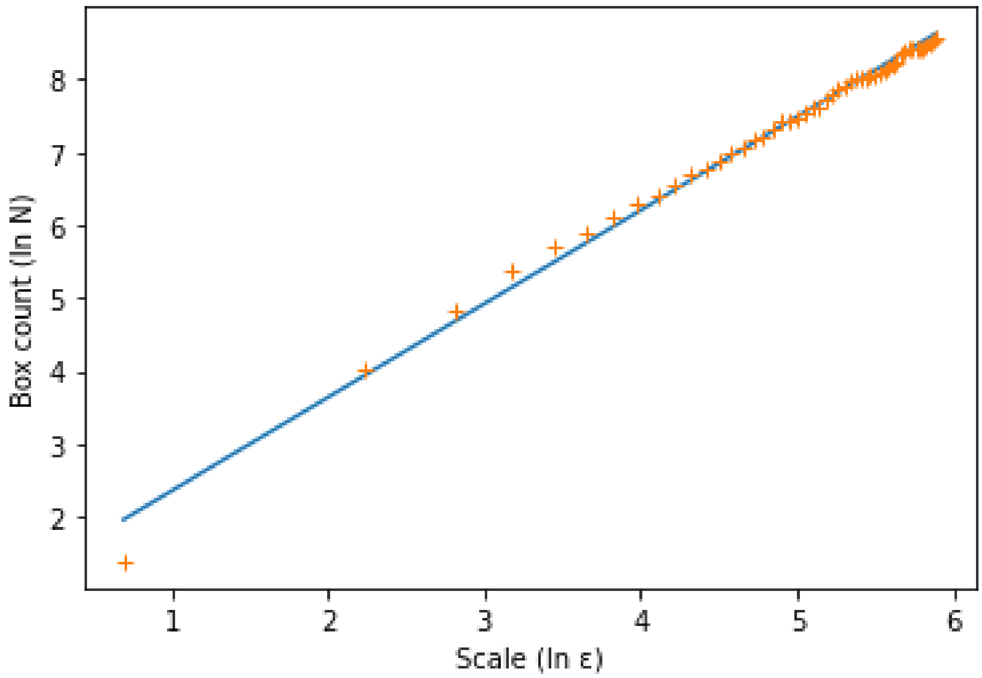
1.6214794967487127




```
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

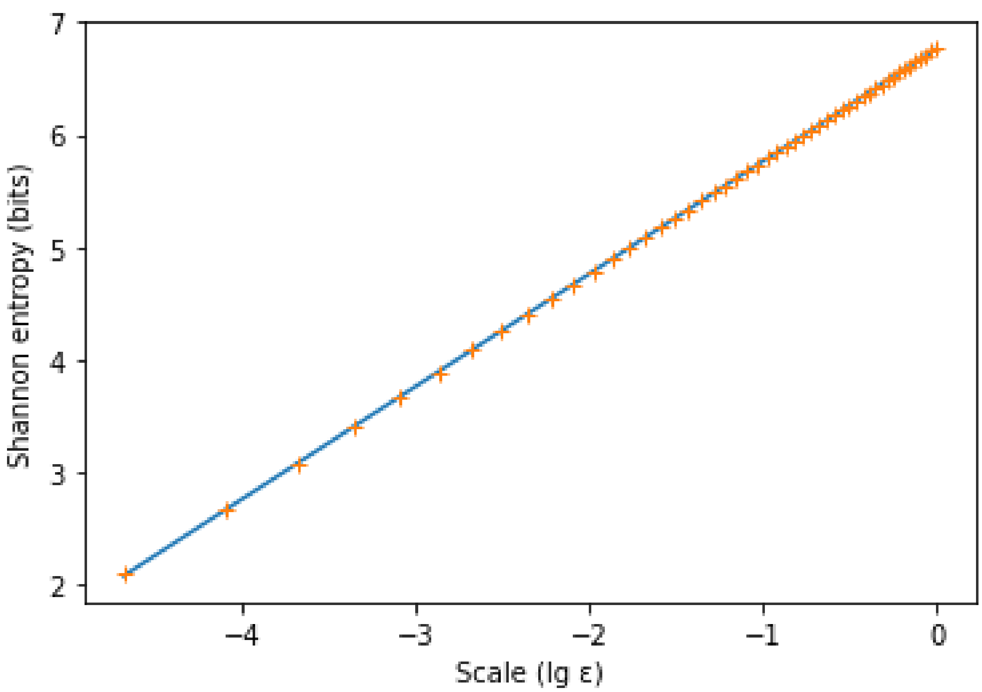
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

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))
```

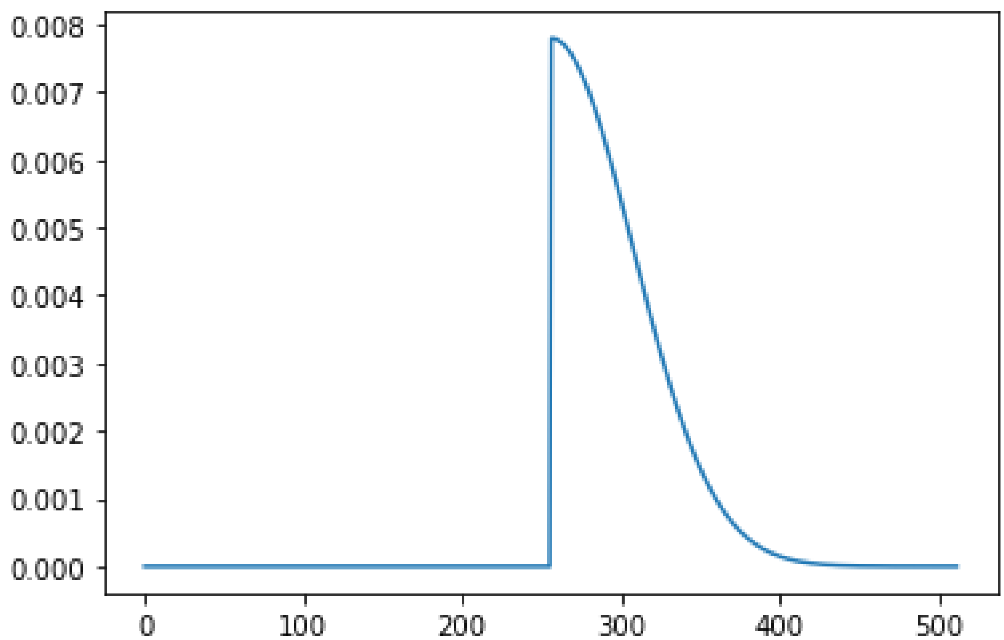
```
infodim((10/256) * gaussian(0, 1, np.linspace(-5, 5, 256)))
```

```
(1.0014184290221988, 0.015707497682893923)
```



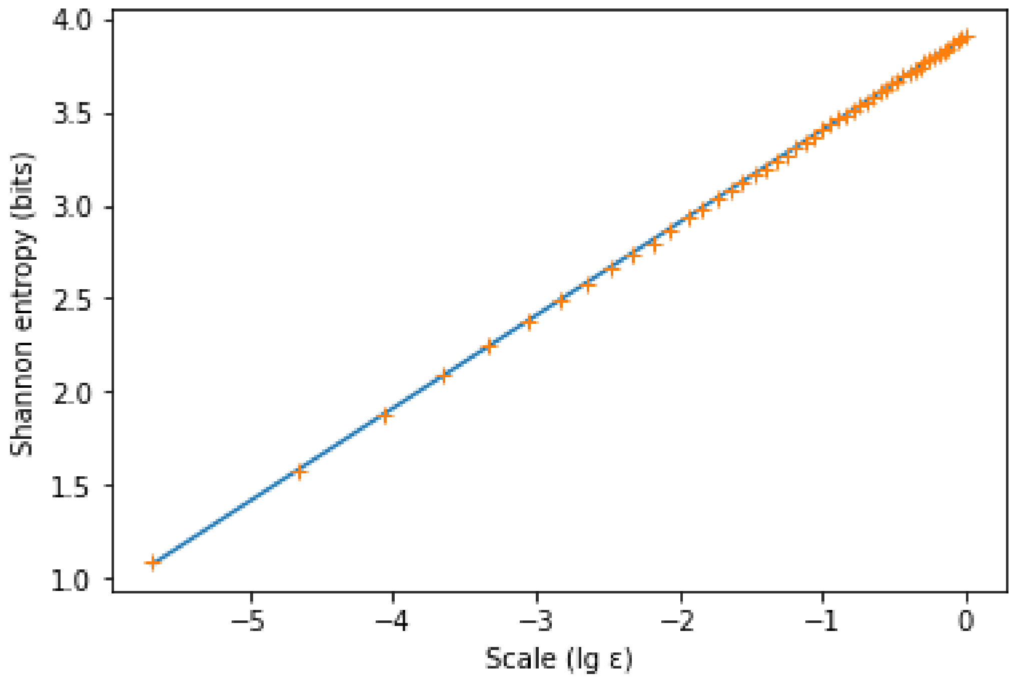
The rectified Gaussian distribution $g(x) = \Theta(x) f(x) + \delta(x)/2$ is half-continuous, so its information dimension is 1/2.

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



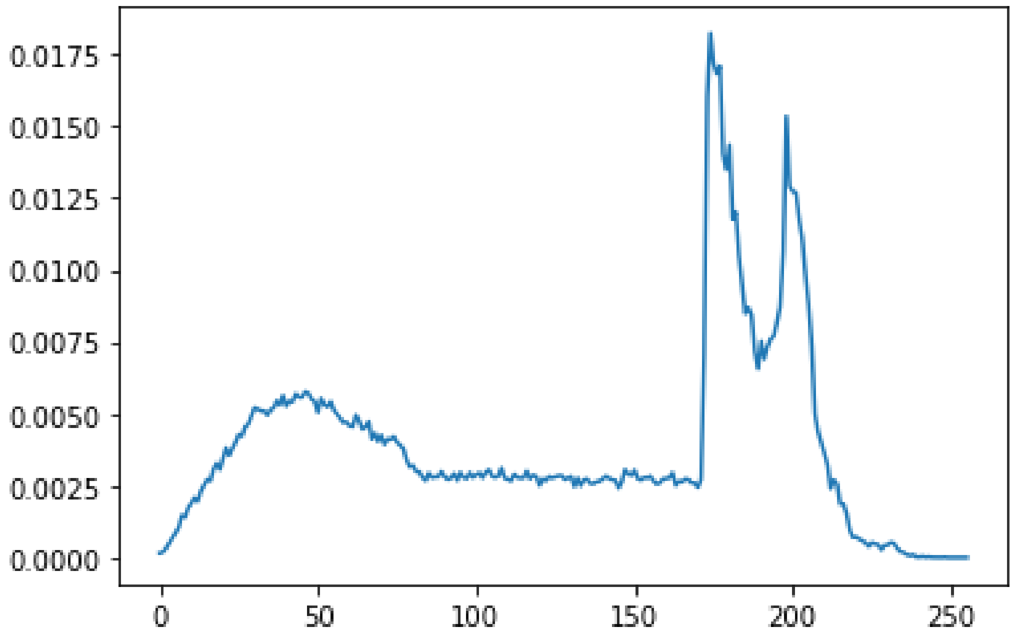

```
infodim(dist)
```

```
(0.4979088715795226, 0.012313889825394398)
```



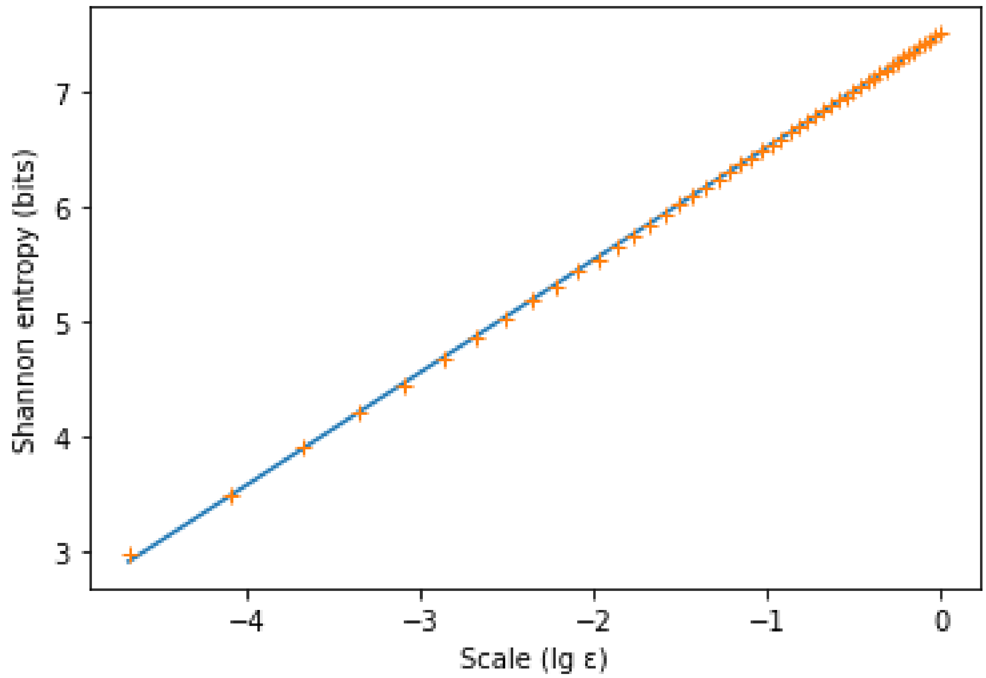
Now that we've validated `infodim`, what does it say about the intensity distribution of an image?

```
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
```

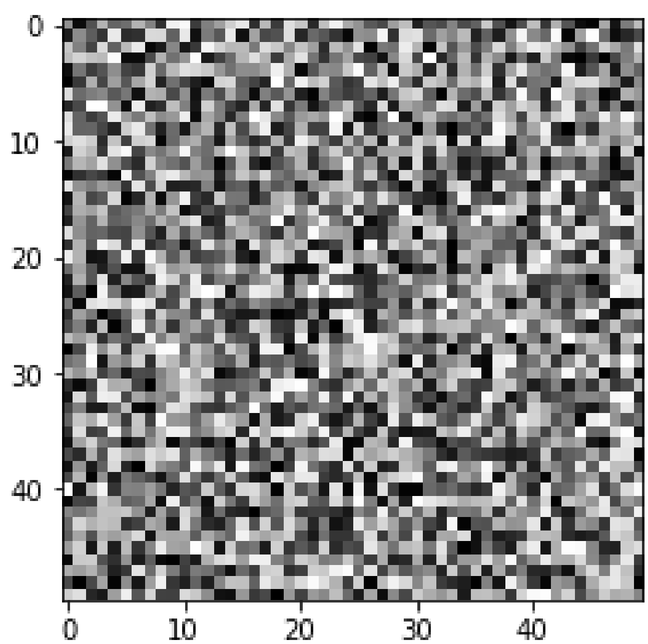
12.1 Standard Ising (on a torus)

In grayscale for fun.

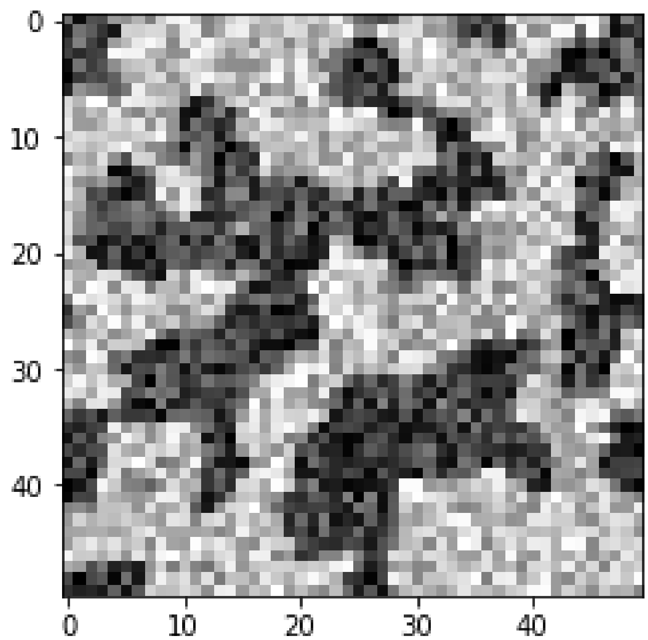
```
def neighbors(a, i, j):
    return np.hstack([a[:,j].take([i-1,i+1], mode='wrap'),
                      a[i,:].take([j-1,j+1], mode='wrap')])
```

```
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);
```



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



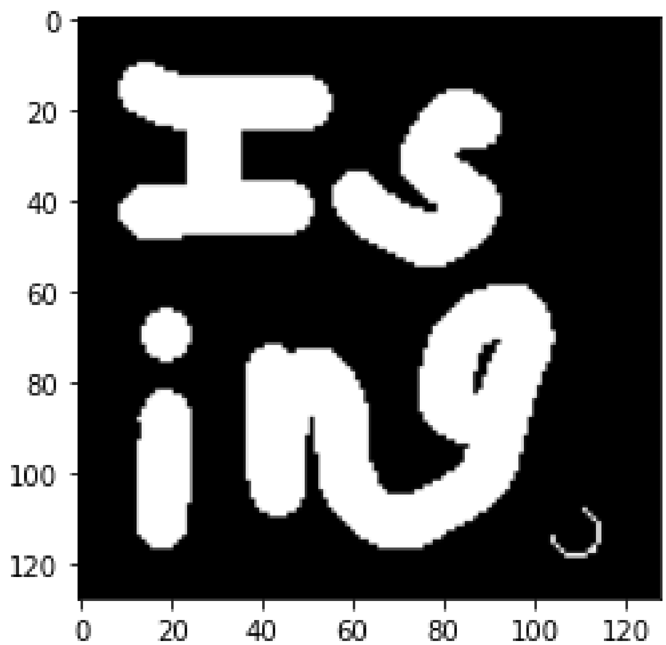
12.2 Image-edge Ising

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

```python
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);
```

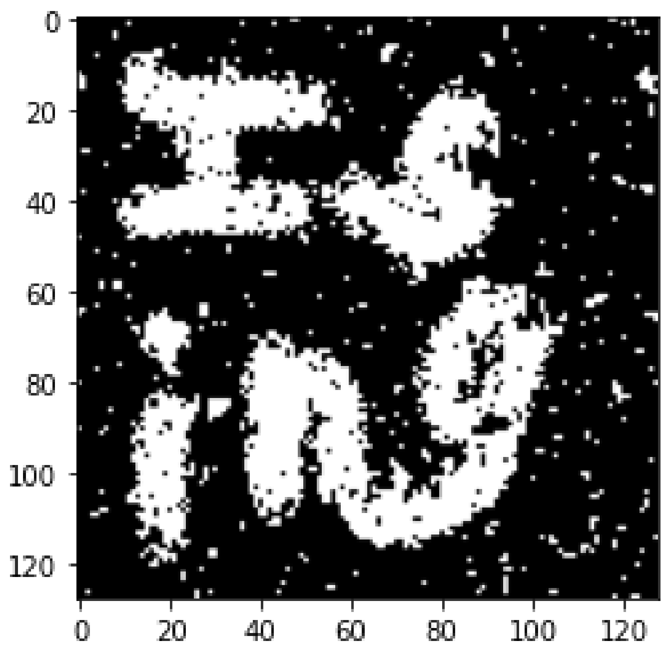


`movie.gif`: Full neighbor Ising.

```
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):
```

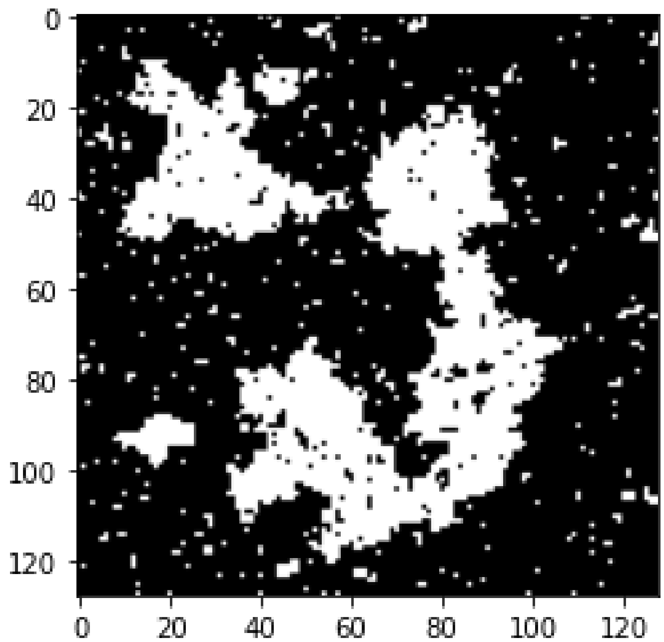
```
        eisingstep(0.5 * (np.pi / 2) / np.arctan(n - i), eimg, edata)
        if i % 1000 == 0:
            f.value += 1
            frame(writer, eimg)
plt.imshow(eimg);
```

```
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')])
```

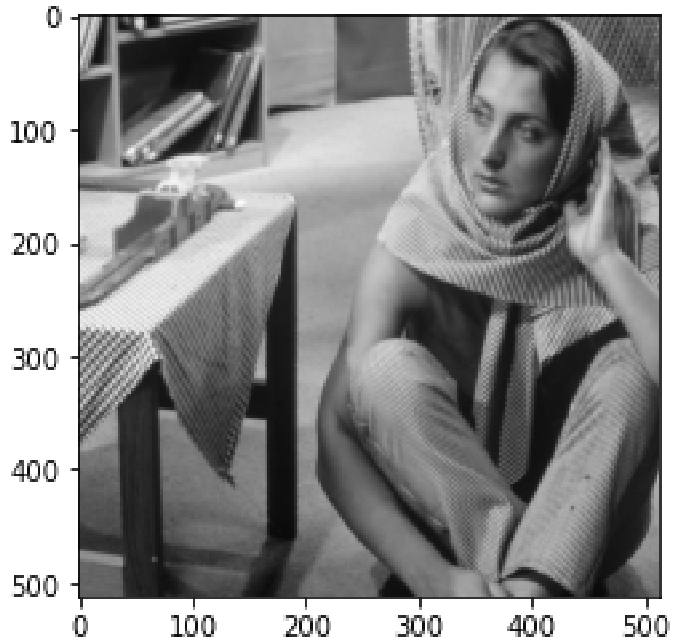
12.3.1 Unrestricted swapping motion

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);
```

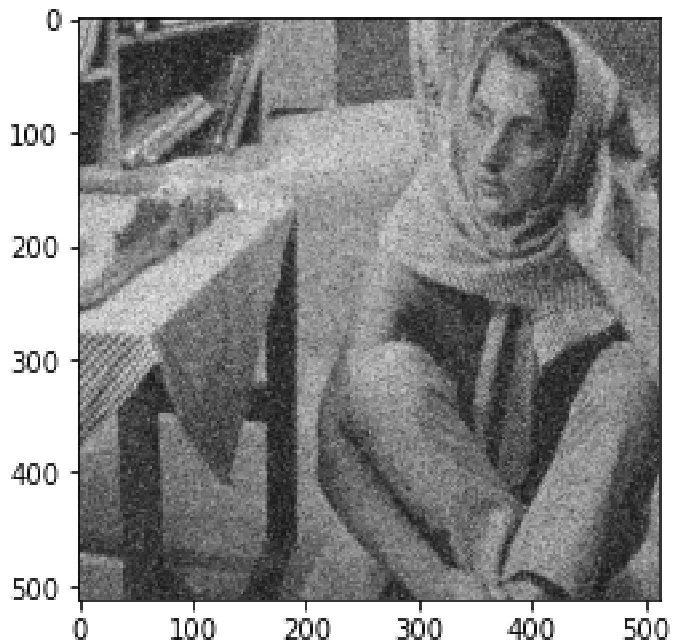


swmovie.gif: Image metric Ising (arbitrary swaps with ienergy).

```
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
```

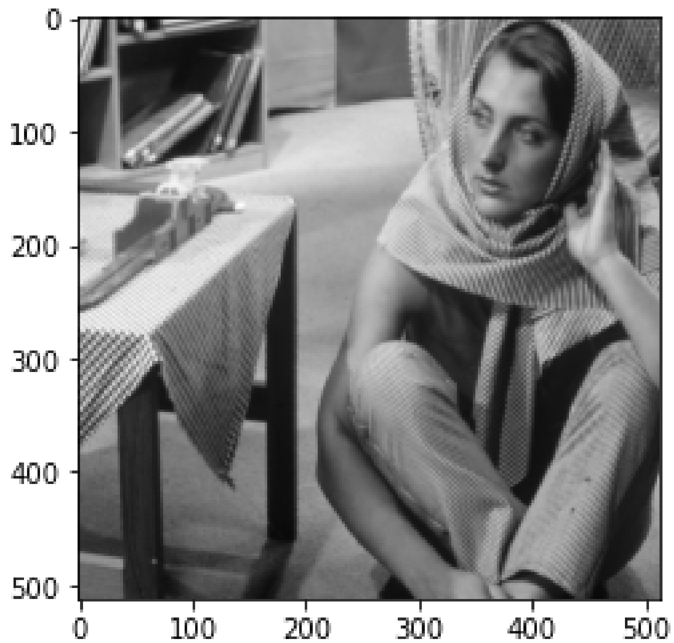
```python
#             frame(writer, eimg)

plt.imshow(eimg);
```

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



12.3.2 Nearest-neighbor swapping motion

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

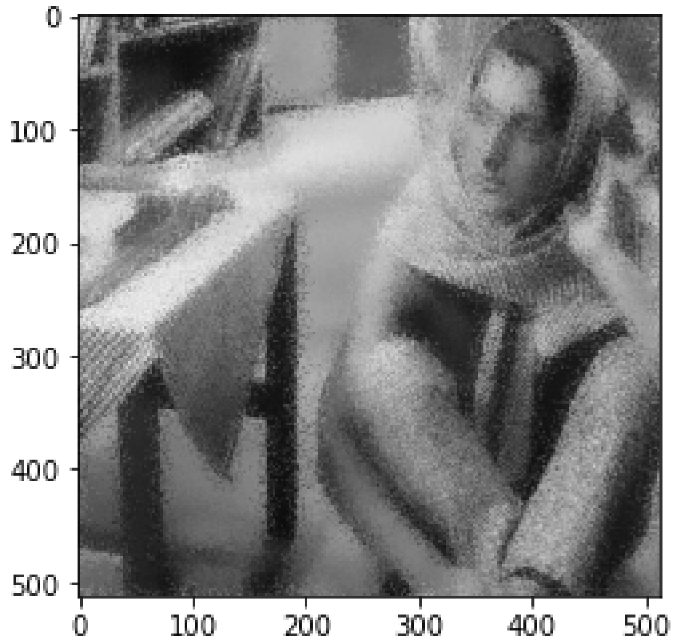


`nnmovie.gif`: Image metric Ising (neighborly swaps with `ienergy`).

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

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

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

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

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 Simulations for canonical ensemble averages

June 10, 2020

17 The Wang-Landau algorithm (density of states)

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

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate
```

The test system is the 2d Ising model.

```python
class Ising:
    def __init__(self, n):
        self.n = n
        self.spins = np.sign(np.random.rand(n, n) - 0.5)
        self.E = self.energy()
        self.Ev = self.E
    def neighbors(self, i, j):
        return np.hstack([self.spins[:,j].take([i-1,i+1], mode='wrap'),
                          self.spins[i,:].take([j-1,j+1], mode='wrap')])
    def energy(self):
        return -0.5 * sum(np.sum(s * self.neighbors(i, j))
                         for (i, j), s in np.ndenumerate(self.spins))
    def propose(self):
        i, j = np.random.randint(self.n), np.random.randint(self.n)
        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
```

Note that this class-based approach adds some overhead. For speed, instances of `Ising` should be inlined into the simulation method.

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

```
def flat(H, tol = 0.1):
    """Determines if an evenly-spaced histogram is approximately flat."""
    Hμ = np.mean(H)
    Hf = np.max(H)
    H0 = np.min(H)
    return Hf / (1 + tol) < Hμ < H0 / (1 - tol)
```

```
# Note: some parameters are hardcoded for testing
def density_sim(system):
    randint = np.random.randint
    rand = np.random.rand
    exp = np.exp

    # Parameters
    M = 1_000_000 # Monte carlo step scale
    ε = 1e-6
    logftol = np.log(1 + ε)
    logf0 = 1
    N = int(32**2 / 20) # Energy bins
    E0 = -32**2 / 4
    Ef = 32**2 / 4

    ΔE = (Ef - E0) / (N - 1)
    fiters = int(np.ceil(np.log2(logf0) - np.log2(logftol)))
    fiter = 0
    mciters = 0
    Es = np.linspace(E0, Ef, N)
    S = np.zeros(N) # Set all initial g's to 1
    H = np.zeros(N, dtype=int)
    logf = logf0
    # Linearly bin the energy
    i = max(0, min(N - 1, int(round((N - 1) * (system.E - E0) / (Ef - E0)))))
    print("ΔE = {}".format(ΔE))
    while logftol < logf:
        H[:] = 0
        logf /= 2
        iters = 0
        niters = int((M + 1) * exp(-logf / 2))
        fiter += 1
        while not flat(H[:-1]) and iters < niters:
```

```
            system.propose()
            Ev = system.Ev
            j = max(0, min(N - 1, int(round((N - 1) * (Ev - E0) / (Ef - E0)))))
            if E0 - ΔE/2 ≤ Ev ≤ Ef + ΔE/2 and (S[j] < S[i] or rand() < exp(S[i] - S[j])):
                system.accept()
                i = j
            H[i] += 1
            S[i] += logf
            iters += 1
        mciters += iters
        print("f: {} / {}\t({} / {})".format(fiter, fiters, iters, niters))

    print("Done: {} total MC iterations.".format(mciters))
    return Es, S, H
```

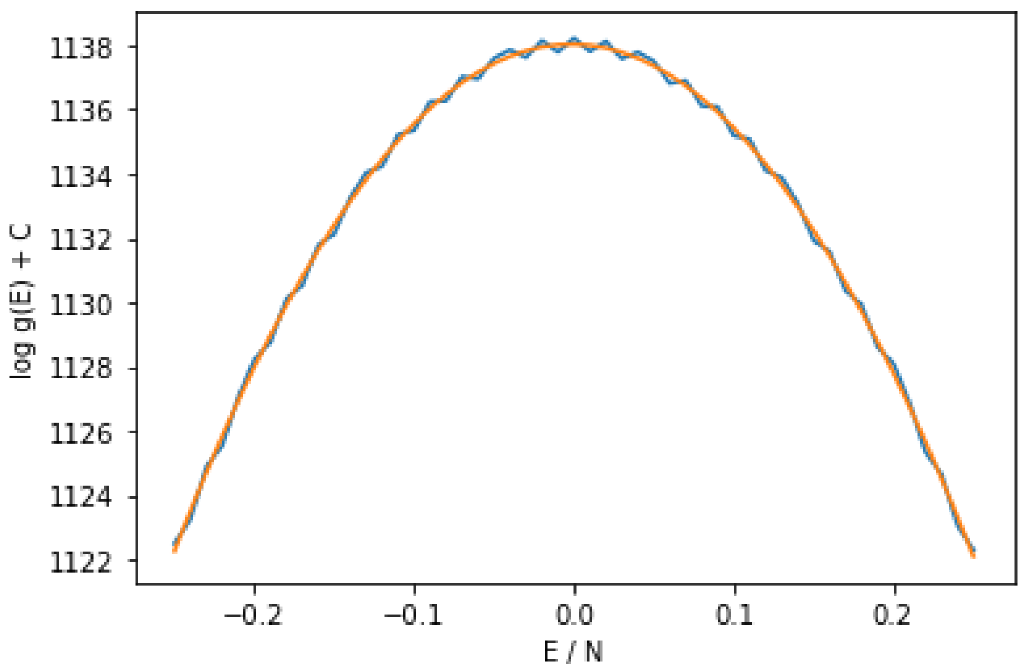
```
isingn = 32
sys = Ising(isingn)
Es, S, H = density_sim(sys);
```

```
ΔE = 10.24
f: 1 / 20   (46498 / 778801)
f: 2 / 20   (51746 / 882497)
f: 3 / 20   (78519 / 939414)
f: 4 / 20   (51944 / 969234)
f: 5 / 20   (62813 / 984497)
f: 6 / 20   (171583 / 992218)
f: 7 / 20   (168143 / 996102)
f: 8 / 20   (237575 / 998049)
f: 9 / 20   (303706 / 999024)
f: 10 / 20  (280809 / 999512)
f: 11 / 20  (577765 / 999756)
f: 12 / 20  (999878 / 999878)
f: 13 / 20  (927226 / 999939)
f: 14 / 20  (999970 / 999970)
f: 15 / 20  (999985 / 999985)
f: 16 / 20  (999993 / 999993)
f: 17 / 20  (867712 / 999997)
f: 18 / 20  (999999 / 999999)
f: 19 / 20  (1000000 / 1000000)
f: 20 / 20  (1000000 / 1000000)
Done: 10825864 total MC iterations.
```

17.1 Calculating canonical ensemble averages

```
gspl = interpolate.splrep(Es, S, s=2*np.sqrt(2))
gs = np.exp(interpolate.splev(Es, gspl) - min(S))
```

```
plt.plot(Es / isingn**2, S)
plt.plot(Es / isingn**2, interpolate.splev(Es, gspl))
plt.xlabel("E / N")
plt.ylabel("log g(E) + C");
```



Translate energies to have minimum zero so that Z is representable.

```
nEs = Es - min(Es)
```

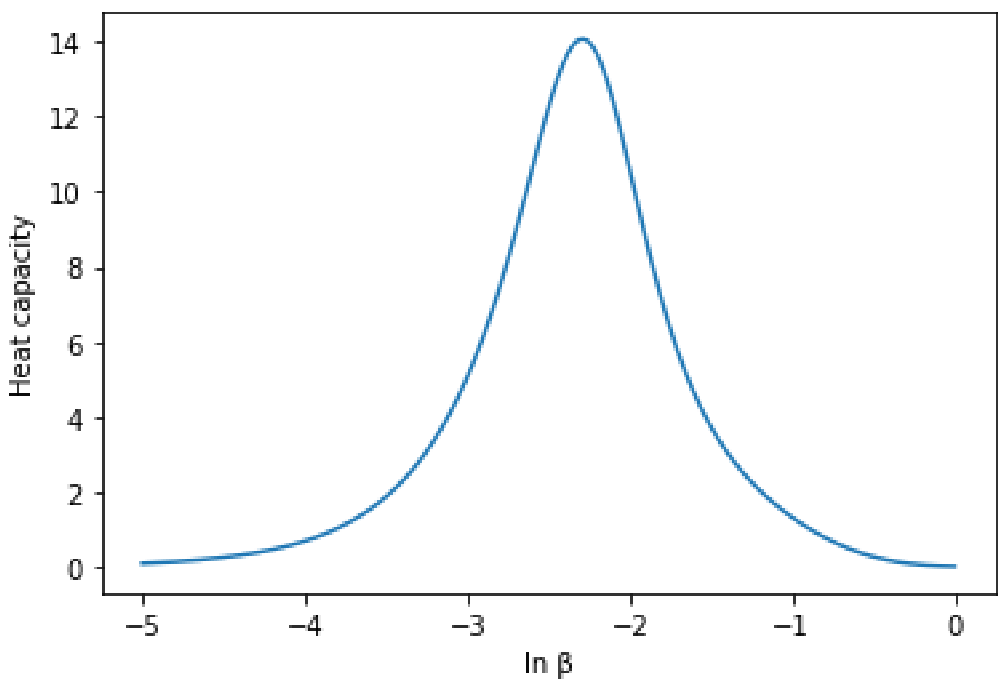
```
Z = lambda β: np.sum(gs * np.exp(-β * nEs))
```

Ensemble averages

```
βs = [np.exp(k) for k in np.linspace(-5, 0, 200)]
Eμ = lambda β: np.sum(nEs * gs * np.exp(-β * nEs)) / Z(β)
E2 = lambda β: np.sum(nEs**2 * gs * np.exp(-β * nEs)) / Z(β)
CV = lambda β: (E2(β) - Eμ(β)**2) * β**2
F  = lambda β: -np.log(Z(β)) / β
Sc = lambda β: β*Eμ(β) + np.log(Z(β))
```

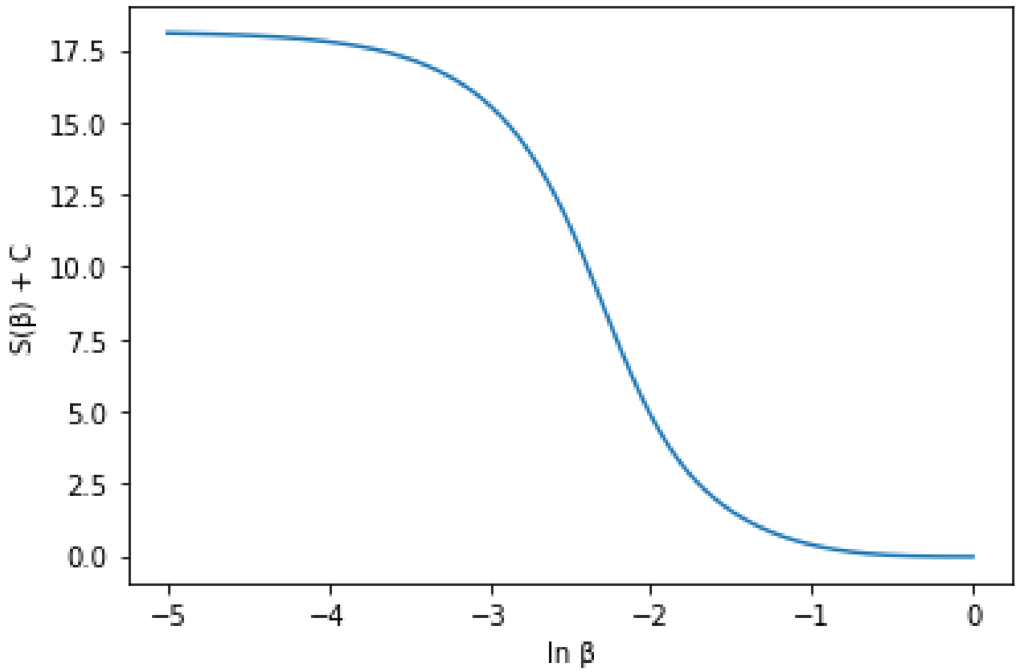
Heat capacity

```
plt.plot(np.log(βs), [CV(β) for β in βs])
plt.xlabel("ln β")
plt.ylabel("Heat capacity")
plt.show()
```



Entropy

```
plt.plot(np.log(βs), [Sc(β) for β in βs])
plt.xlabel("ln β")
plt.ylabel("S(β) + C")
plt.show()
```


S(β) + C
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−5
−4
−3
−2
−1
0
ln β