

1 The Wang-Landau algorithm (density of states)

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

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy import interpolate, special
```

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

```
1 def flat(H, tol = 0.2):
2     """Determines if an evenly-spaced histogram is approximately flat."""
3     Hμ = np.mean(H)
4     Hf = np.max(H)
5     H0 = np.min(H)
6     return Hf / (1 + tol) < Hμ < H0 / (1 - tol)
7 # def flat(H, tol = 0.2):
8 #     """Determines if an evenly-spaced histogram is approximately flat."""
9 #     Hμ = np.mean(H)
10 #     return not np.any(H < (1 - tol) * Hμ) and np.all(H ≠ 0)

1 def wanglandau(system,
2                 M = 1_00_000, # Monte carlo step scale
3                 ε = 1e-8,     # f tolerance
4                 logf0 = 1,    # Initial log f
5                 N = 8**2 + 1, # Number of energy bins
6                 E0 = -2 * 8**2, # Minimum energy
7                 Ef = 2 * 8**2, # Maximum energy
8                 ):
9     # Initial values
10    logf = logf0
11    logftol = np.log(1 + ε)
12    Es = np.linspace(E0, Ef, N)
13    S = np.zeros(N) # Set all initial g's to 1
14    H = np.zeros(N, dtype=int)
15    # Linearly bin the energy
16    i = max(0, min(N - 1, int(round((N - 1) * (system.E - E0) / (Ef - E0)))))
17
18    # Logging
19    mciters = 0
20    fiter = 0
21    ΔE = (Ef - E0) / (N - 1)
22    fiters = int(np.ceil(np.log2(logf0) - np.log2(logftol)))
```

```

23     print("ΔE = {}".format(ΔE))
24
25     while logftol < logf:
26         H[:] = 0
27         logf /= 2
28         iters = 0
29         niters = int((M + 1) * np.exp(-logf / 2))
30         fiter += 1
31         while not flat(H[2:-2]) and iters < niters: # Ising-specific histogram
32             # while not flat(H) and iters < niters:
33                 system.propose()
34                 Ev = system.Ev
35                 j = max(0, min(N - 1, int(round((N - 1) * (Ev - E0) / (Ef - E0))))))
36                 if E0 - ΔE/2 ≤ Ev ≤ Ef + ΔE/2 and (S[j] < S[i] or np.random.rand() < np.exp(S[i] -
37                     ↪ S[j])):
38                     system.accept()
39                     i = j
40                     H[i] += 1
41                     S[i] += logf
42                     iters += 1
43                 mciters += iters
44                 print("f: {} / {} \t({} / {}".format(fiter, fitters, iters, niters))
45
46     print("Done: {} total MC iterations.".format(mciters))
47     return Es, S, H

```

1.1 The 2D Ising model

```

1 class Ising:
2     def __init__(self, n):
3         self.n = n
4         self.spins = np.sign(np.random.rand(n, n) - 0.5)
5         self.E = self.energy()
6         self.Ev = self.E
7     def neighbors(self, i, j):
8         return np.hstack([self.spins[:,j].take([i-1,i+1], mode='wrap'),
9             self.spins[i,:].take([j-1,j+1], mode='wrap')])
10    def energy(self):
11        return -0.5 * sum(np.sum(s * self.neighbors(i, j))
12            for (i, j), s in np.ndenumerate(self.spins))
13    def propose(self):
14        i, j = np.random.randint(self.n), np.random.randint(self.n)
15        self.i, self.j = i, j
16        dE = 2 * np.sum(self.spins[i, j] * self.neighbors(i, j))
17        self.dE = dE

```

```

18         self.Ev = self.E + dE
19     def accept(self):
20         self.spins[self.i, self.j] *= -1
21         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.

```

1  isingn = 8
2  sys = Ising(isingn)
3  Es, S, H = wanglandau(sys);

```

```

ΔE = 4.0
f: 1 / 27 (55874 / 77880)
f: 2 / 27 (32200 / 88250)
f: 3 / 27 (72964 / 93942)
f: 4 / 27 (70127 / 96924)
f: 5 / 27 (78111 / 98450)
f: 6 / 27 (43058 / 99222)
f: 7 / 27 (99611 / 99611)
f: 8 / 27 (99805 / 99805)
f: 9 / 27 (99903 / 99903)
f: 10 / 27 (99952 / 99952)
f: 11 / 27 (99976 / 99976)
f: 12 / 27 (99988 / 99988)
f: 13 / 27 (99994 / 99994)
f: 14 / 27 (99997 / 99997)
f: 15 / 27 (99999 / 99999)
f: 16 / 27 (100000 / 100000)
f: 17 / 27 (100000 / 100000)
f: 18 / 27 (100000 / 100000)
f: 19 / 27 (100000 / 100000)
f: 20 / 27 (100000 / 100000)
f: 21 / 27 (100000 / 100000)
f: 22 / 27 (100000 / 100000)
f: 23 / 27 (100000 / 100000)
f: 24 / 27 (100000 / 100000)
f: 25 / 27 (100000 / 100000)
f: 26 / 27 (100000 / 100000)

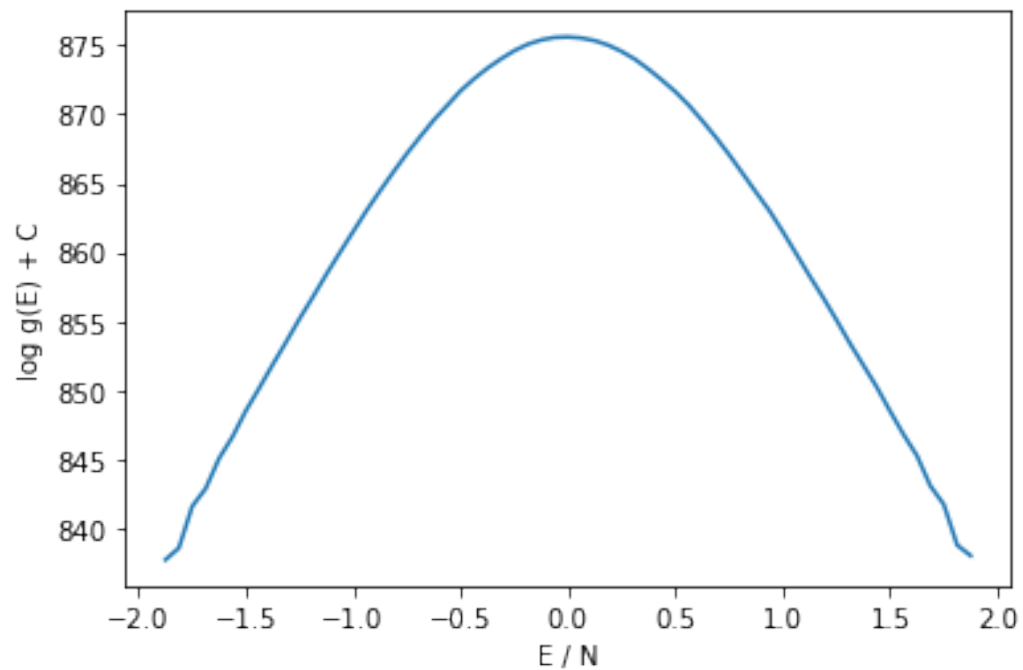
```

f: 27 / 27 (100000 / 100000)
Done: 2451559 total MC iterations.

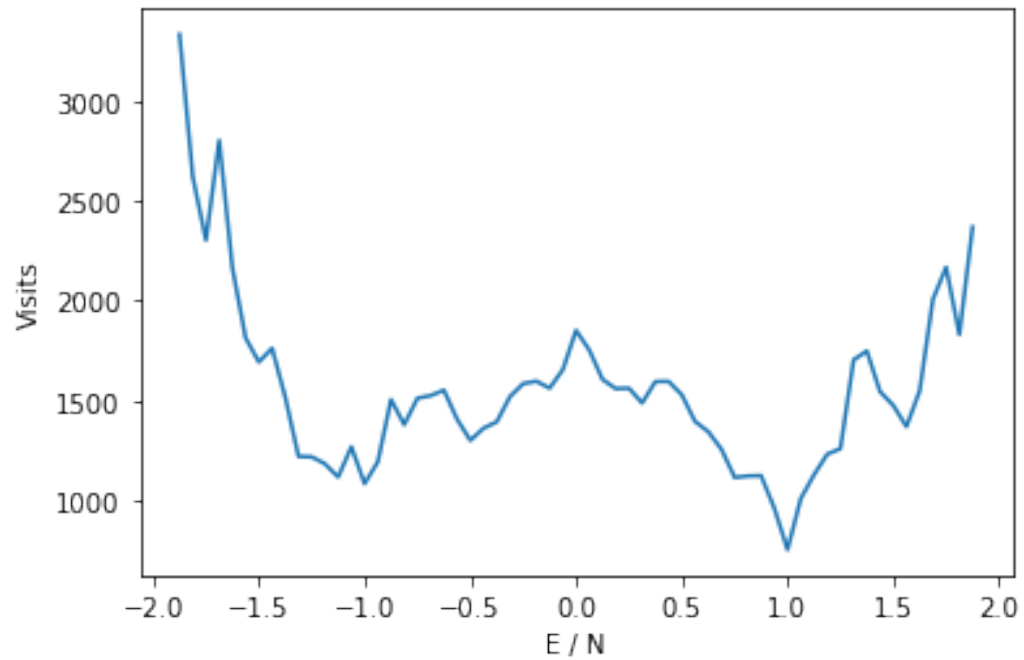
The energies at indices 1 and -1 are not occupied in the Ising model.

```
1 Es, S, H = Es[2:-2], S[2:-2], H[2:-2]
```

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



```
1 plt.plot(Es / isingn**2, H)  
2 plt.xlabel("E / N")  
3 plt.ylabel("Visits");
```



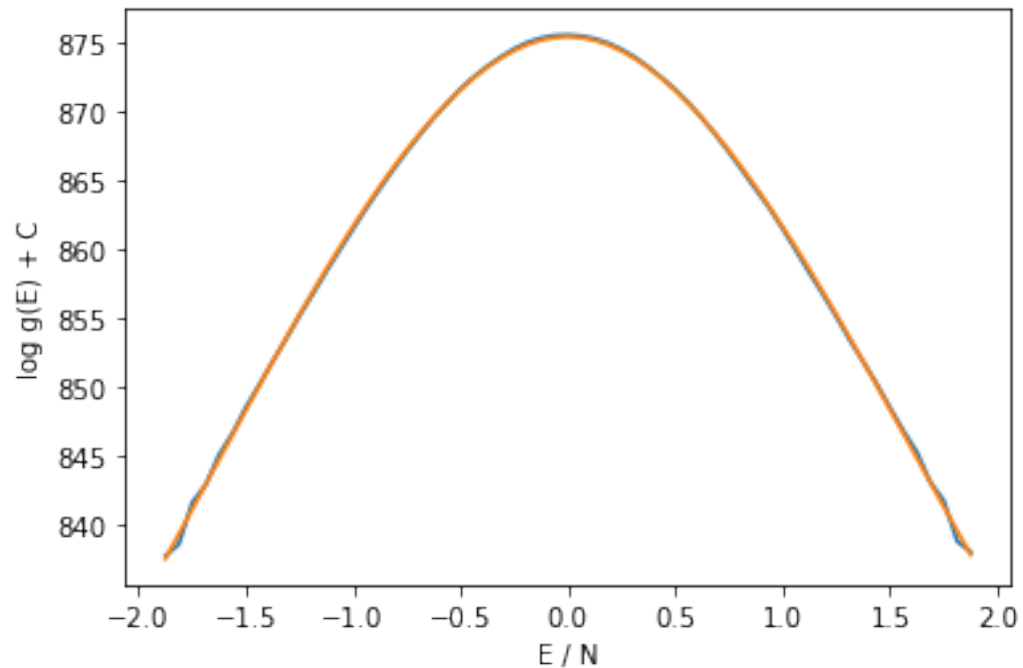
1.1.1 Calculating canonical ensemble averages

```

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

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

```



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

```

1 nEs = Es - min(Es)

1 Z = lambda beta: np.sum(gEs * np.exp(-beta * nEs))

```

Ensemble averages

```

1 beta_s = [np.exp(k) for k in np.linspace(-3, 1, 200)]
2 E_mu = lambda beta: np.sum(nEs * gEs * np.exp(-beta * nEs)) / Z(beta)
3 E2 = lambda beta: np.sum(nEs**2 * gEs * np.exp(-beta * nEs)) / Z(beta)
4 CV = lambda beta: (E2(beta) - E_mu(beta)**2) * beta**2
5 F = lambda beta: -np.log(Z(beta)) / beta
6 Sc = lambda beta: beta * E_mu(beta) + np.log(Z(beta))

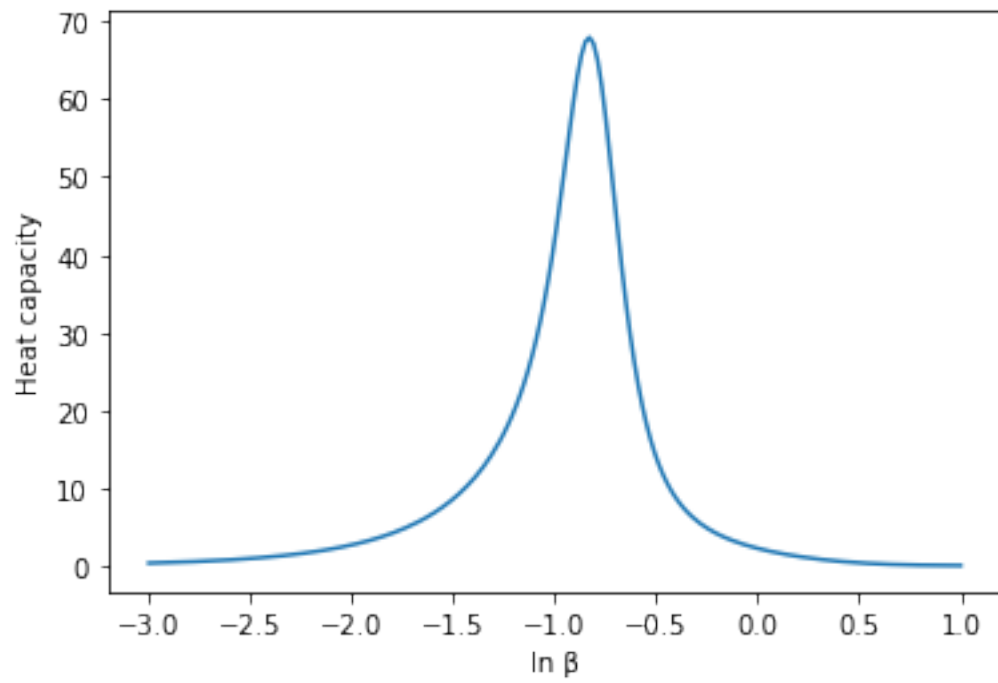
```

Heat capacity

```

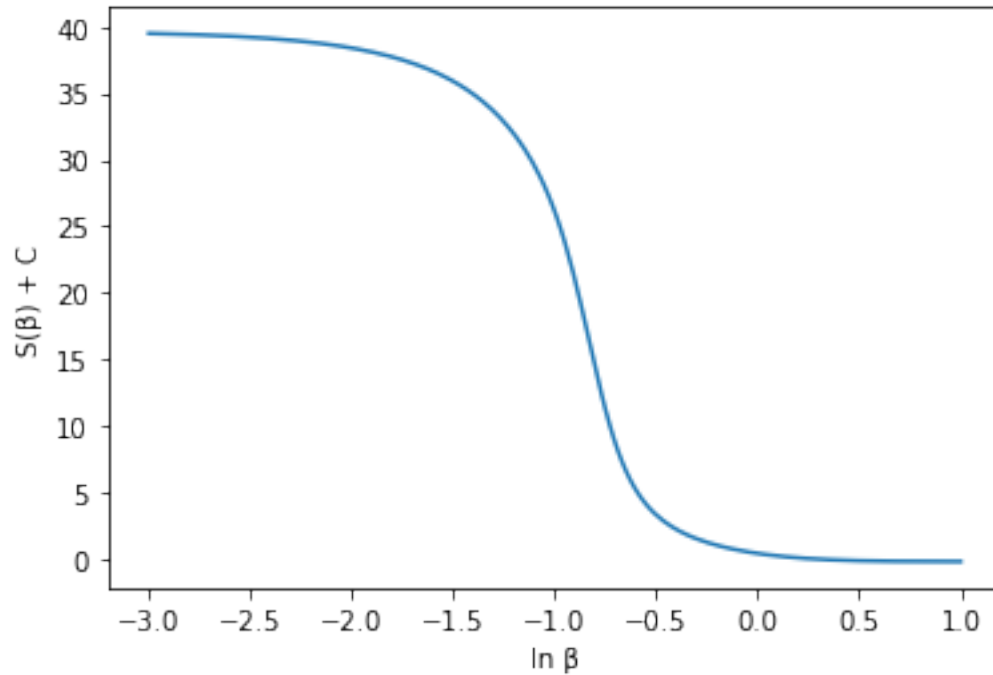
1 plt.plot(np.log(beta_s), [CV(beta) for beta in beta_s])
2 plt.xlabel("ln beta")
3 plt.ylabel("Heat capacity")
4 plt.show()

```



Entropy

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



1.2 Thermal calculations on images

```

1 class StatisticalImage:
2     def __init__(self, I0):
3         self.I0 = I0
4         self.I = I0.copy()
5         self.w, self.h = np.shape(I0)
6         self.E = self.energy()
7         self.Ev = self.E
8     def energy(self):
9         return sum(x0 - x if x < x0 else x - x0
10                  for x, x0 in zip(self.I.flat, self.I0.flat))
11    def propose(self):
12        i, j = np.random.randint(self.w), np.random.randint(self.h)
13        self.i, self.j = i, j
14        x0 = self.I0[i, j]
15        x = self.I[i, j]
16        r = 16
17        dx = np.random.randint(-min(r, x), min(r, 255 - x) + 1)
18        x1 = x + dx
19        dE = (x0 - x1 if x1 < x0 else x1 - x0) - (x0 - x if x < x0 else x - x0)
20        self.dx = dx

```



```

21         self.dE = dE
22         self.Ev = self.E + dE
23     def accept(self):
24         self.I[self.i, self.j] += self.dx
25         self.E = self.Ev

1  Ls = range(1, 11, 2)
2  wlresults = [wanglandau(StatisticalImage(128 * np.ones((L, L), dtype=int)),
3                                     M=1_000_000, N=127*L*L + 1, E0=0, Ef=127*L*L)
4               for L in Ls]

```

$\Delta E = 1.0$

```

f: 1 / 27  (3903 / 778801)
f: 2 / 27  (9528 / 882497)
f: 3 / 27  (9083 / 939414)
f: 4 / 27  (10139 / 969234)
f: 5 / 27  (26254 / 984497)
f: 6 / 27  (27029 / 992218)
f: 7 / 27  (34230 / 996102)
f: 8 / 27  (32353 / 998049)
f: 9 / 27  (38764 / 999024)
f: 10 / 27 (16429 / 999512)
f: 11 / 27 (38150 / 999756)
f: 12 / 27 (64131 / 999878)
f: 13 / 27 (129645 / 999939)
f: 14 / 27 (25586 / 999970)
f: 15 / 27 (53326 / 999985)
f: 16 / 27 (31720 / 999993)
f: 17 / 27 (33121 / 999997)
f: 18 / 27 (24031 / 999999)
f: 19 / 27 (49997 / 1000000)
f: 20 / 27 (49427 / 1000000)
f: 21 / 27 (42771 / 1000000)
f: 22 / 27 (34318 / 1000000)
f: 23 / 27 (39775 / 1000000)
f: 24 / 27 (26611 / 1000000)
f: 25 / 27 (52471 / 1000000)
f: 26 / 27 (26318 / 1000000)
f: 27 / 27 (21238 / 1000000)

```

Done: 950348 total MC iterations.

$\Delta E = 1.0$
f: 1 / 27 (778801 / 778801)
f: 2 / 27 (882497 / 882497)
f: 3 / 27 (693591 / 939414)
f: 4 / 27 (969234 / 969234)
f: 5 / 27 (984497 / 984497)
f: 6 / 27 (992218 / 992218)
f: 7 / 27 (885823 / 996102)
f: 8 / 27 (998049 / 998049)
f: 9 / 27 (999024 / 999024)
f: 10 / 27 (999512 / 999512)
f: 11 / 27 (999756 / 999756)
f: 12 / 27 (999878 / 999878)
f: 13 / 27 (999939 / 999939)
f: 14 / 27 (999970 / 999970)
f: 15 / 27 (999985 / 999985)
f: 16 / 27 (999993 / 999993)
f: 17 / 27 (999997 / 999997)
f: 18 / 27 (999999 / 999999)
f: 19 / 27 (1000000 / 1000000)
f: 20 / 27 (1000000 / 1000000)
f: 21 / 27 (1000000 / 1000000)
f: 22 / 27 (1000000 / 1000000)
f: 23 / 27 (1000000 / 1000000)
f: 24 / 27 (1000000 / 1000000)
f: 25 / 27 (1000000 / 1000000)
f: 26 / 27 (1000000 / 1000000)
f: 27 / 27 (1000000 / 1000000)
Done: 26182763 total MC iterations.
 $\Delta E = 1.0$
f: 1 / 27 (778801 / 778801)
f: 2 / 27 (882497 / 882497)
f: 3 / 27 (939414 / 939414)
f: 4 / 27 (969234 / 969234)
f: 5 / 27 (984497 / 984497)
f: 6 / 27 (992218 / 992218)
f: 7 / 27 (996102 / 996102)
f: 8 / 27 (998049 / 998049)

```

f: 9 / 27 (999024 / 999024)
f: 10 / 27 (999512 / 999512)
f: 11 / 27 (999756 / 999756)
f: 12 / 27 (999878 / 999878)
f: 13 / 27 (999939 / 999939)
f: 14 / 27 (999970 / 999970)
f: 15 / 27 (999985 / 999985)
f: 16 / 27 (999993 / 999993)
f: 17 / 27 (999997 / 999997)
f: 18 / 27 (999999 / 999999)
f: 19 / 27 (1000000 / 1000000)
f: 20 / 27 (1000000 / 1000000)
f: 21 / 27 (1000000 / 1000000)
f: 22 / 27 (1000000 / 1000000)
f: 23 / 27 (1000000 / 1000000)
f: 24 / 27 (1000000 / 1000000)
f: 25 / 27 (1000000 / 1000000)
f: 26 / 27 (1000000 / 1000000)
f: 27 / 27 (1000000 / 1000000)
Done: 26538865 total MC iterations.
ΔE = 1.0
f: 1 / 27 (778801 / 778801)
f: 2 / 27 (882497 / 882497)
f: 3 / 27 (939414 / 939414)
f: 4 / 27 (969234 / 969234)
f: 5 / 27 (984497 / 984497)
f: 6 / 27 (992218 / 992218)
f: 7 / 27 (996102 / 996102)
f: 8 / 27 (998049 / 998049)
f: 9 / 27 (999024 / 999024)
f: 10 / 27 (999512 / 999512)
f: 11 / 27 (999756 / 999756)
f: 12 / 27 (999878 / 999878)
f: 13 / 27 (999939 / 999939)
f: 14 / 27 (999970 / 999970)
f: 15 / 27 (999985 / 999985)
f: 16 / 27 (999993 / 999993)
f: 17 / 27 (999997 / 999997)

```

```

f: 18 / 27 (999999 / 999999)
f: 19 / 27 (1000000 / 1000000)
f: 20 / 27 (1000000 / 1000000)
f: 21 / 27 (1000000 / 1000000)
f: 22 / 27 (1000000 / 1000000)
f: 23 / 27 (1000000 / 1000000)
f: 24 / 27 (1000000 / 1000000)
f: 25 / 27 (1000000 / 1000000)
f: 26 / 27 (1000000 / 1000000)
f: 27 / 27 (1000000 / 1000000)
Done: 26538865 total MC iterations.
ΔE = 1.0
f: 1 / 27 (778801 / 778801)
f: 2 / 27 (882497 / 882497)
f: 3 / 27 (939414 / 939414)
f: 4 / 27 (969234 / 969234)
f: 5 / 27 (984497 / 984497)
f: 6 / 27 (992218 / 992218)
f: 7 / 27 (996102 / 996102)
f: 8 / 27 (998049 / 998049)
f: 9 / 27 (999024 / 999024)
f: 10 / 27 (999512 / 999512)
f: 11 / 27 (999756 / 999756)
f: 12 / 27 (999878 / 999878)
f: 13 / 27 (999939 / 999939)
f: 14 / 27 (999970 / 999970)
f: 15 / 27 (999985 / 999985)
f: 16 / 27 (999993 / 999993)
f: 17 / 27 (999997 / 999997)
f: 18 / 27 (999999 / 999999)
f: 19 / 27 (1000000 / 1000000)
f: 20 / 27 (1000000 / 1000000)
f: 21 / 27 (1000000 / 1000000)
f: 22 / 27 (1000000 / 1000000)
f: 23 / 27 (1000000 / 1000000)
f: 24 / 27 (1000000 / 1000000)
f: 25 / 27 (1000000 / 1000000)
f: 26 / 27 (1000000 / 1000000)

```

f: 27 / 27 (1000000 / 1000000)
 Done: 26538865 total MC iterations.

```

1 import pickle
2 with open('wlresults.pickle', 'wb') as f:
3     pickle.dump(list(Ls), f)
4     pickle.dump(wlresults, f)

1 L = Ls[3]
2 wlEs, S, H = wlresults[3]
3 L

7

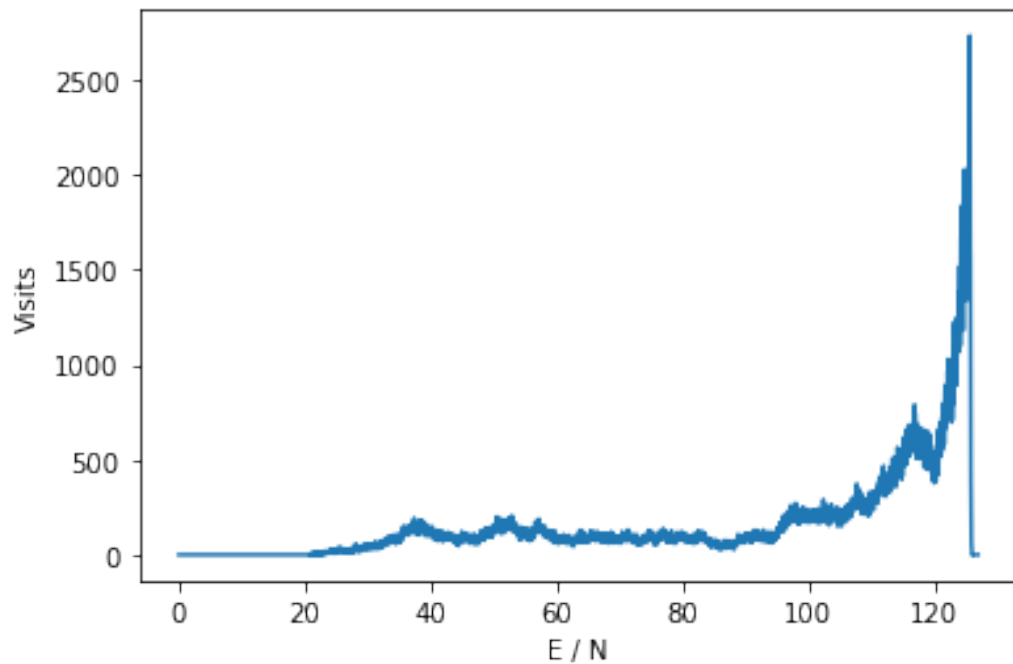
```

Look at the histogram to see how the last WL iteration went.

```

1 plt.plot(wlEs / L**2, H)
2 plt.xlabel("E / N")
3 plt.ylabel("Visits");

```



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

```

1 gsp1 = interpolate.splrep(wlEs, S, s=0*np.sqrt(2))
2 wlgsC = np.exp(interpolate.splev(wlEs, gsp1) - min(S))

```

1.2.1 Exact solution

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.

```
1 def bw_g(E, N, M, exact=True):
2     return sum((-1)**k * special.comb(N, k, exact=exact) * special.comb(E + N - 1 - k*(M + 1), E
3         ↪ - k*(M + 1), exact=exact)
4         for k in range(int(E / M) + 1))
5 def gray_g(E, N, M, exact=True):
6     return 2 * bw_g(E, N, M, exact=exact)
```

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

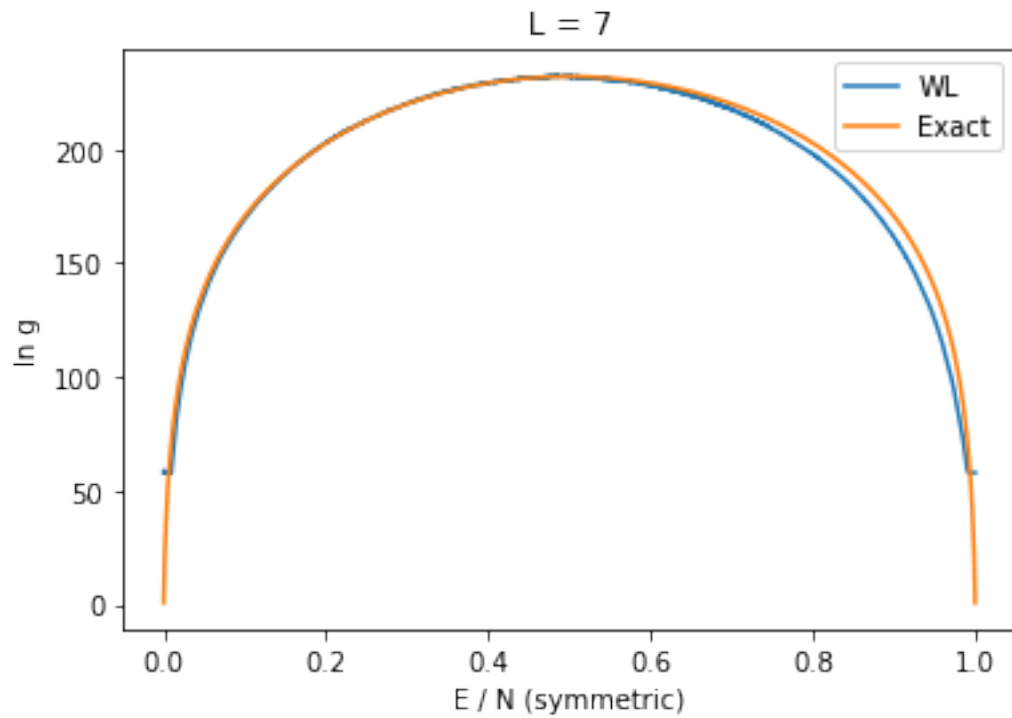
```
1 def reflect(a):
2     return np.hstack([a[:-2], a[-1], a[-2::-1]])
3 def gray_gs(N, M):
4     Es = np.arange(N*M + 1)
5     gs = np.vectorize(gray_g)(np.arange(1 + N*M / 2), N, M, exact=False)
6     return Es, reflect(gs)
1 Es, gs = gray_gs(N=L**2, M=2**7 - 1)
```

Renormalize the WL result

```
1 wlgs = wlgsC * (gs[len(gs) // 2] / wlgsC[len(wlgsC) // 2])
```

Compare the exact result to the WL result.

```
1 plt.plot(wlEs / len(wlEs), np.log(wlgs), label='WL')
2 plt.plot(Es / len(Es), np.log(gs), label='Exact')
3 plt.xlabel('E / N (symmetric)')
4 plt.ylabel('ln g')
5 plt.title('L = {}'.format(L))
6 plt.legend();
```



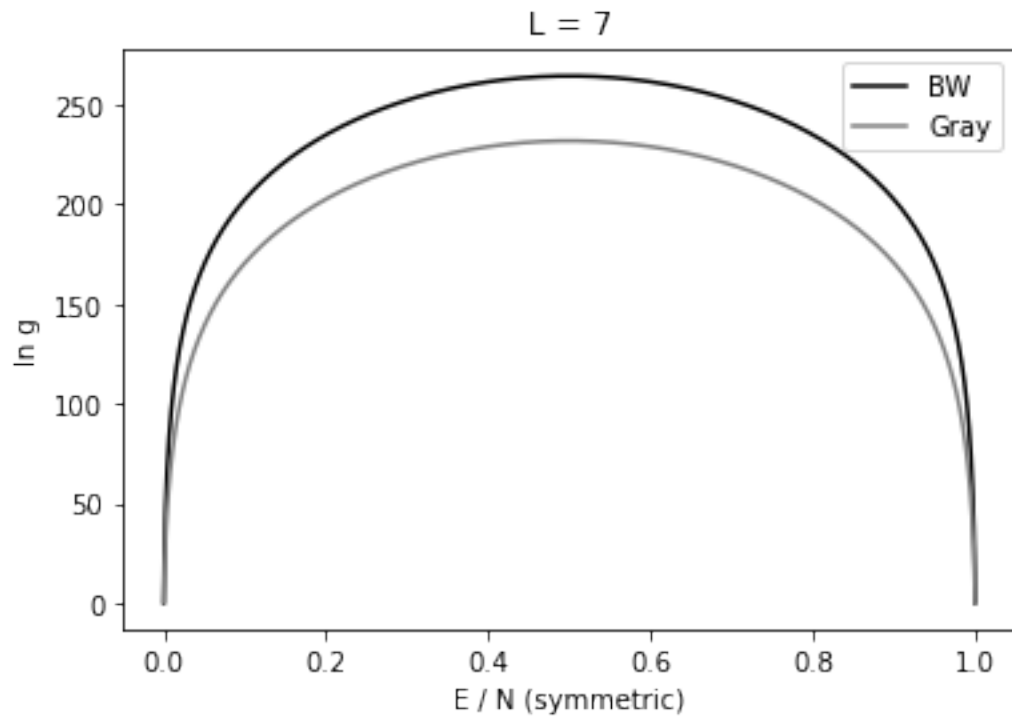
Presumably all of the densities of states for different images fall in the region between the all-gray and all-black/white curves.

```

1 bwEs, bwgs = gray_gs(N=L**2, M=2**8 - 1)
2 bwgs /= 2 # Undo gray_gs degeneracy

1 plt.plot(bwEs / len(bwEs), np.log(bwgs), 'black', label='BW')
2 plt.plot(Es / len(Es), np.log(gs), 'gray', label='Gray')
3 plt.xlabel('E / N (symmetric)')
4 plt.ylabel('ln g')
5 plt.title('L = {}'.format(L))
6 plt.legend();

```



1.2.2 Calculating canonical ensemble averages

```

1 class CanonicalEnsemble:
2     def __init__(self, Es, gs, name):
3         self.Es = Es
4         self.gs = gs
5         self.name = name
6     def Z(self,  $\beta$ ):
7         return np.sum(self.gs * np.exp(- $\beta$  * self.Es))
8     def average(self, f,  $\beta$ ):
9         return np.sum(f(self) * self.gs * np.exp(- $\beta$  * self.Es)) / self.Z( $\beta$ )
10    def energy(self,  $\beta$ ):
11        return self.average(lambda ens: ens.Es,  $\beta$ )
12    def energy2(self,  $\beta$ ):
13        return self.average(lambda ens: ens.Es**2,  $\beta$ )
14    def heat_capacity(self,  $\beta$ ):
15        return self.energy2( $\beta$ ) - self.energy( $\beta$ )**2
16    def free_energy(self,  $\beta$ ):
17        return -np.log(self.Z( $\beta$ )) /  $\beta$ 
18    def entropy(self,  $\beta$ ):
19        return  $\beta$  * self.energy( $\beta$ ) + np.log(self.Z( $\beta$ ))

```



```

1  $\beta$ s = [np.exp(k) for k in np.linspace(-8, 2, 500)]
2 wlens = CanonicalEnsemble(wlEs, wlgs, 'WL') # Wang-Landau results
3 xens = CanonicalEnsemble(Es, gs, 'Exact') # Exact
4 ensembles = [wlens, xens]

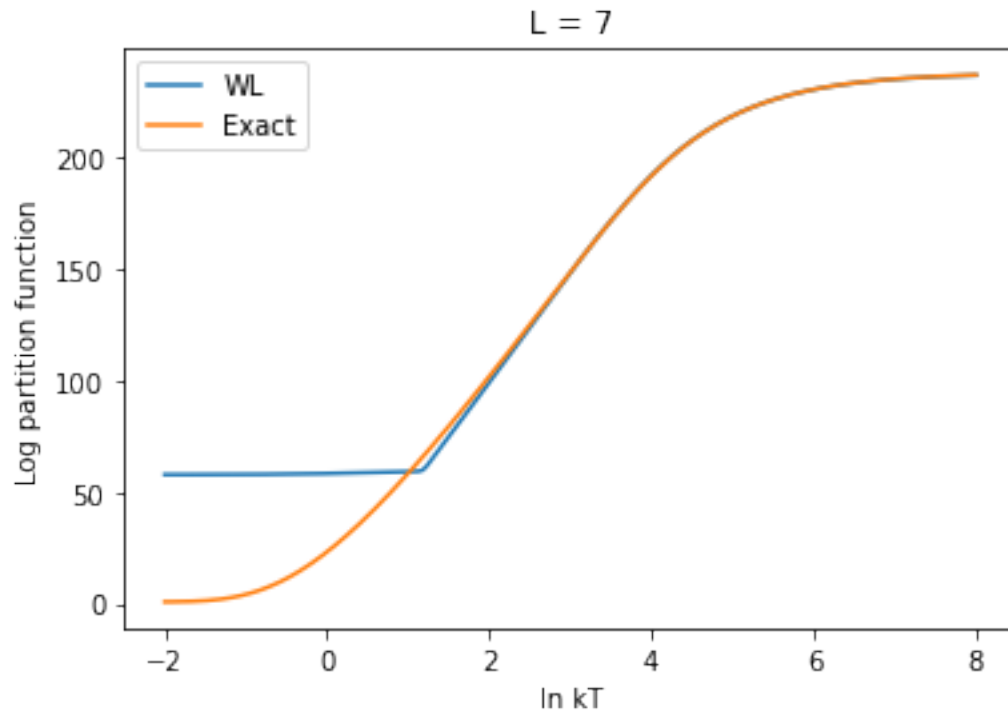
```

Partition function

```

1 for ens in ensembles:
2     plt.plot(-np.log( $\beta$ s), np.log(np.vectorize(ens.Z)( $\beta$ s)), label=ens.name)
3 plt.xlabel("ln kT")
4 plt.ylabel("Log partition function")
5 plt.title('L = {}'.format(L))
6 plt.legend();

```

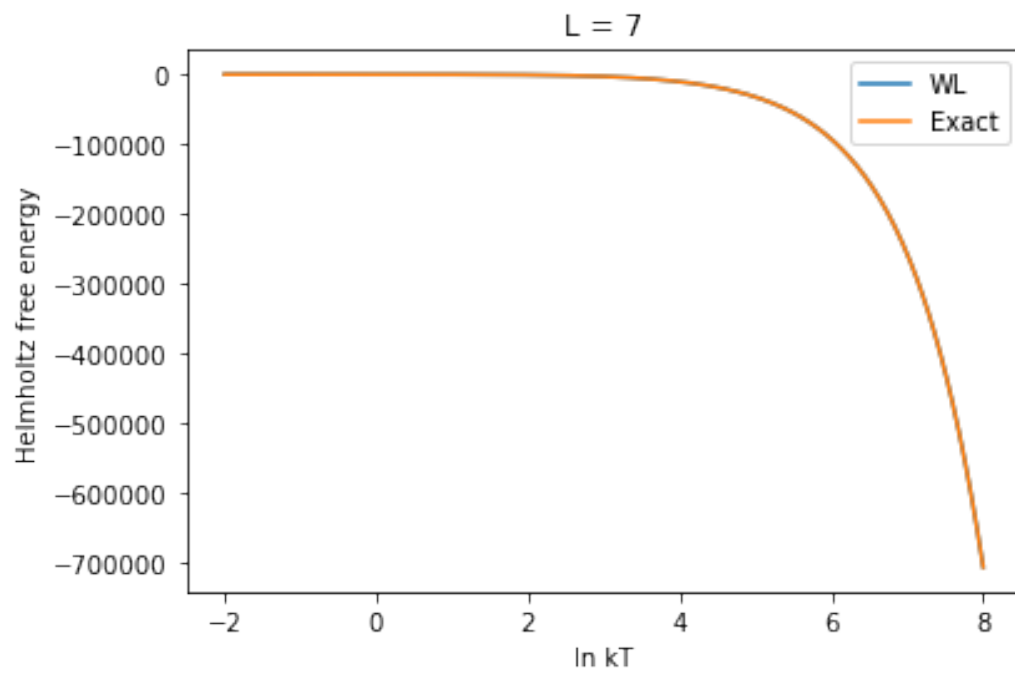


Helmholtz free energy

```

1 for ens in ensembles:
2     plt.plot(-np.log( $\beta$ s), np.vectorize(ens.free_energy)( $\beta$ s), label=ens.name)
3 plt.xlabel("ln kT")
4 plt.ylabel("Helmholtz free energy")
5 plt.title('L = {}'.format(L))
6 plt.legend();

```

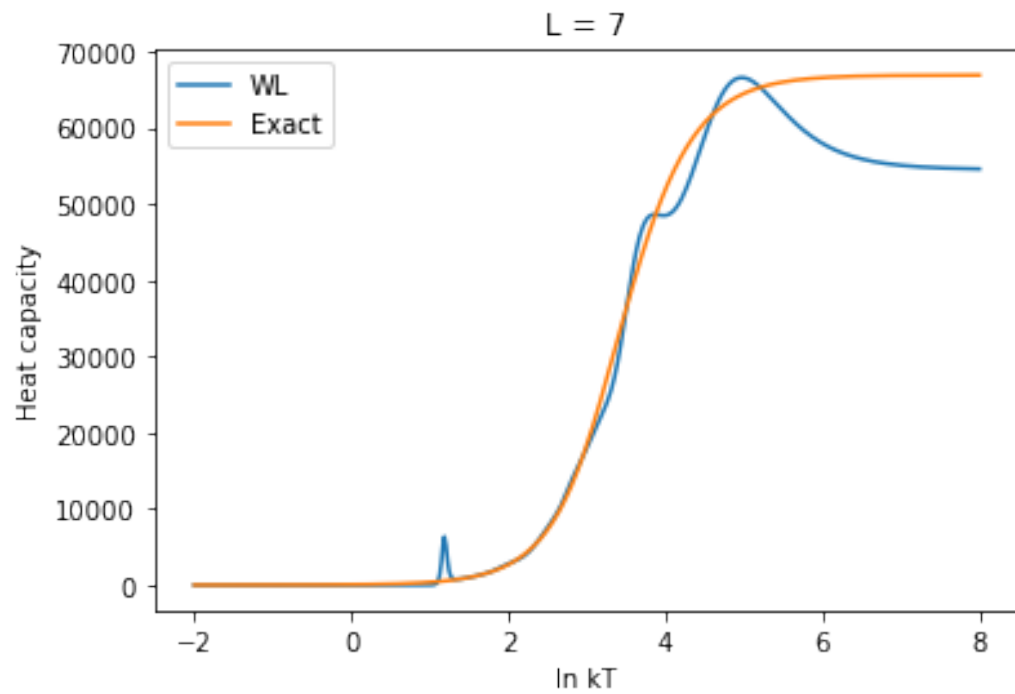


Heat capacity

```

1  for ens in ensembles:
2      plt.plot(-np.log( $\beta$ s), np.vectorize(ens.heat_capacity)( $\beta$ s), label=ens.name)
3  plt.xlabel("ln kT")
4  plt.ylabel("Heat capacity")
5  plt.title('L = {}'.format(L))
6  plt.legend();

```



Entropy

```

1  for ens in ensembles:
2      plt.plot(-np.log(βs), np.vectorize(ens.entropy)(βs), label=ens.name)
3  plt.xlabel("ln kT")
4  plt.ylabel("Canonical entropy")
5  plt.title('L = {}'.format(L))
6  plt.legend();

```

