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.

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

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 = 0.2):
    """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)
# def flat(H, tol = 0.2):
#     """Determines if an evenly-spaced histogram is approximately flat."""
#     Hμ = np.mean(H)
#     return not np.any(H < (1 - tol) * Hμ) and np.all(H ≠ 0)
```

```
def wanglandau(system,
               M = 1_00_000,  # Monte carlo step scale
               ε = 1e-8,      # f tolerance
               logf0 = 1,     # Initial log f
               N = 8**2 + 1,  # Number of energy bins
               E0 = -2 * 8**2, # Minimum energy
               Ef = 2 * 8**2   # Maximum energy
              ):
    # Initial values
    logf = logf0
    logftol = np.log(1 + ε)
    Es = np.linspace(E0, Ef, N)
    S = np.zeros(N) # Set all initial g's to 1
    H = np.zeros(N, dtype=int)
    # Linearly bin the energy
    i = max(0, min(N - 1, int(round((N - 1) * (system.E - E0) / (Ef - E0)))))

    # Logging
    mciters = 0
    fiter = 0
    ΔE = (Ef - E0) / (N - 1)
    fiters = int(np.ceil(np.log2(logf0) - np.log2(logftol)))
```

```
    print("ΔE = {}".format(ΔE))

    while logftol < logf:
        H[:] = 0
        logf /= 2
        iters = 0
        niters = int((M + 1) * np.exp(-logf / 2))
        fiter += 1
        while not flat(H[2:-2]) and iters < niters: # Ising-specific histogram
#         while not flat(H) 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 np.random.rand() < np.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
```

1.1 The 2D Ising model

```
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.

```
isingn = 8
sys = Ising(isingn)
Es, S, H = wanglandau(sys);
```

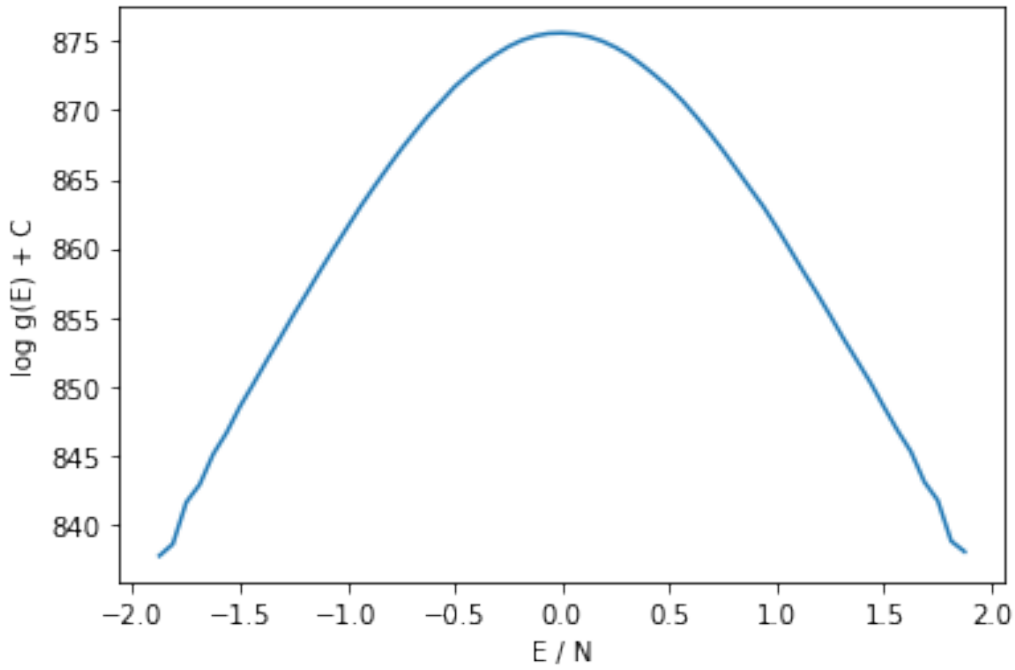
```
ΔE = 4.0
f: 1 / 27   (55874 / 77880)
f: 2 / 27   (32200 / 88250)
f: 3 / 27   (72964 / 93942)
f: 4 / 27   (70127 / 96924)
f: 5 / 27   (78111 / 98450)
f: 6 / 27   (43058 / 99222)
f: 7 / 27   (99611 / 99611)
f: 8 / 27   (99805 / 99805)
f: 9 / 27   (99903 / 99903)
f: 10 / 27  (99952 / 99952)
f: 11 / 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.
```

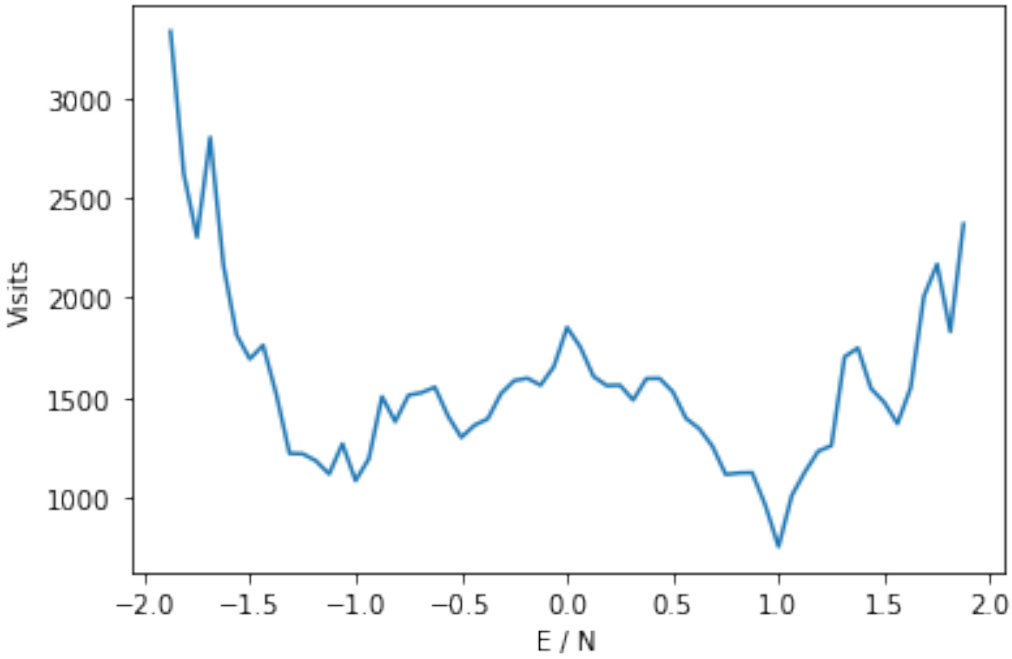
The energies at indices 1 and -1 are not occupied in the Ising model.

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

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



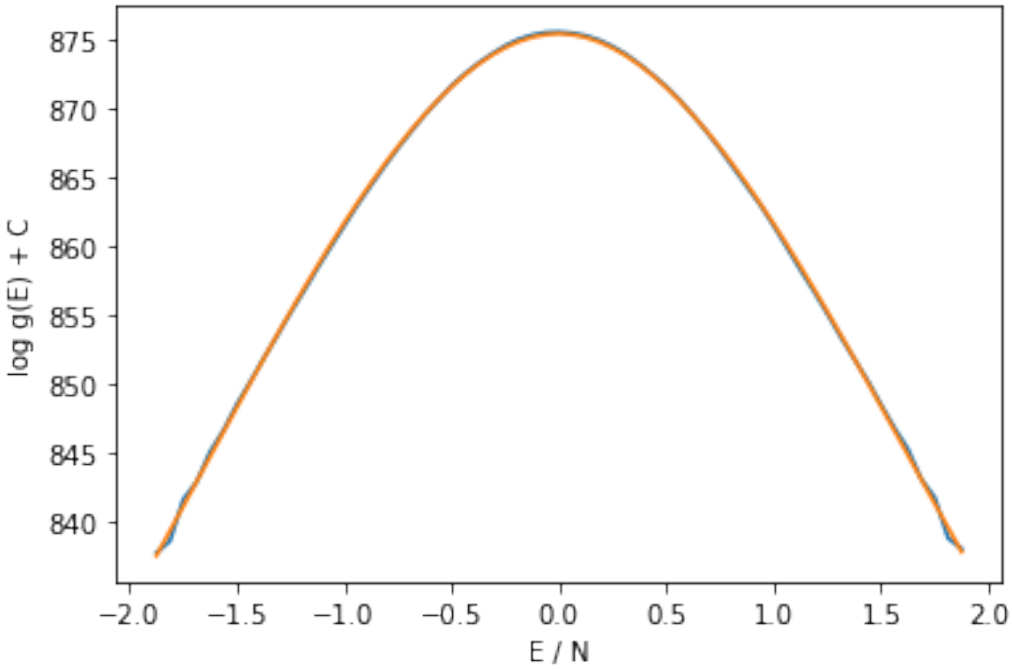

```
plt.plot(Es / isingn**2, H)
plt.xlabel("E / N")
plt.ylabel("Visits");
```



1.1.1 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)
```

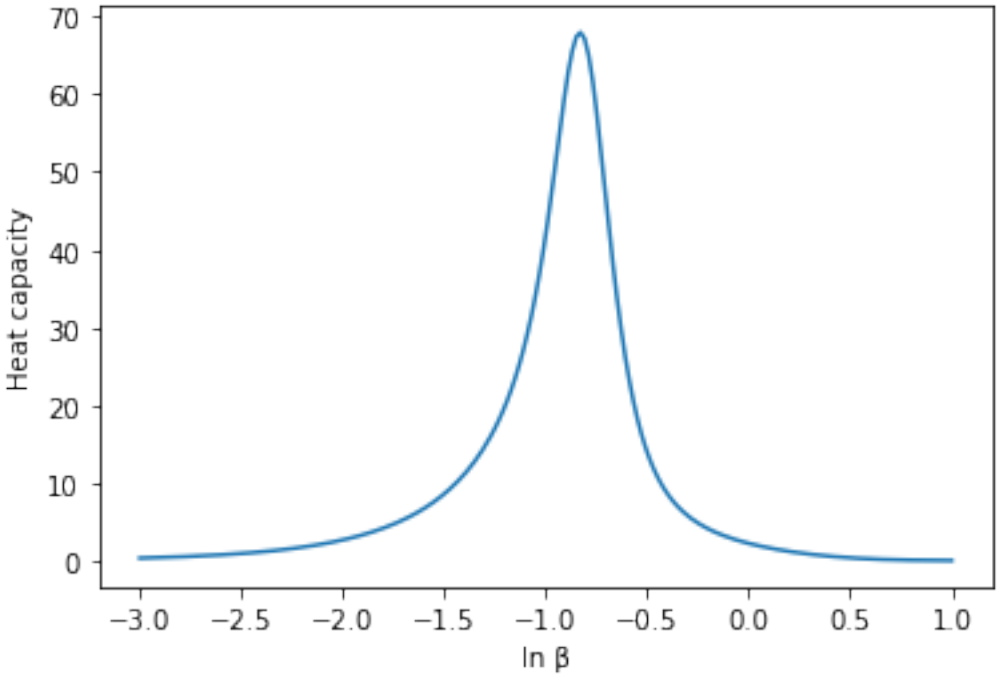
```
Z = lambda β: np.sum(gs * np.exp(-β * nEs))
```

Ensemble averages

```
βs = [np.exp(k) for k in np.linspace(-3, 1, 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))
```

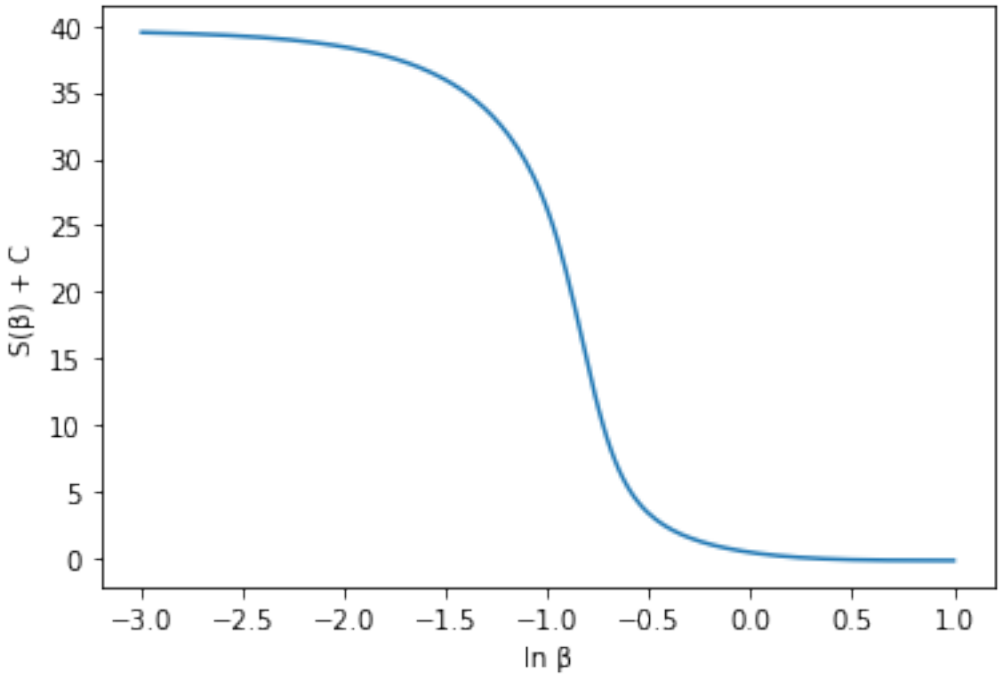
Heat capacity

```
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()
```



1.2 Thermal calculations on images

```
class StatisticalImage:
    def __init__(self, I0):
        self.I0 = I0
        self.I = I0.copy()
        self.w, self.h = np.shape(I0)
        self.E = self.energy()
        self.Ev = self.E
    def energy(self):
        return sum(x0 - x if x < x0 else x - x0
                   for x, x0 in zip(self.I.flat, self.I0.flat))
    def propose(self):
        i, j = np.random.randint(self.w), np.random.randint(self.h)
        self.i, self.j = i, j
        x0 = self.I0[i, j]
        x = self.I[i, j]
        r = 16
        dx = np.random.randint(-min(r, x), min(r, 255 - x) + 1)
        x1 = x + dx
        dE = (x0 - x1 if x1 < x0 else x1 - x0) - (x0 - x if x < x0 else x - x0)
        self.dx = dx
```

```
        self.dE = dE
        self.Ev = self.E + dE
    def accept(self):
        self.I[self.i, self.j] += self.dx
        self.E = self.Ev
```

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

```
ΔE = 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.
```

```
ΔE = 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.
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.
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)
```

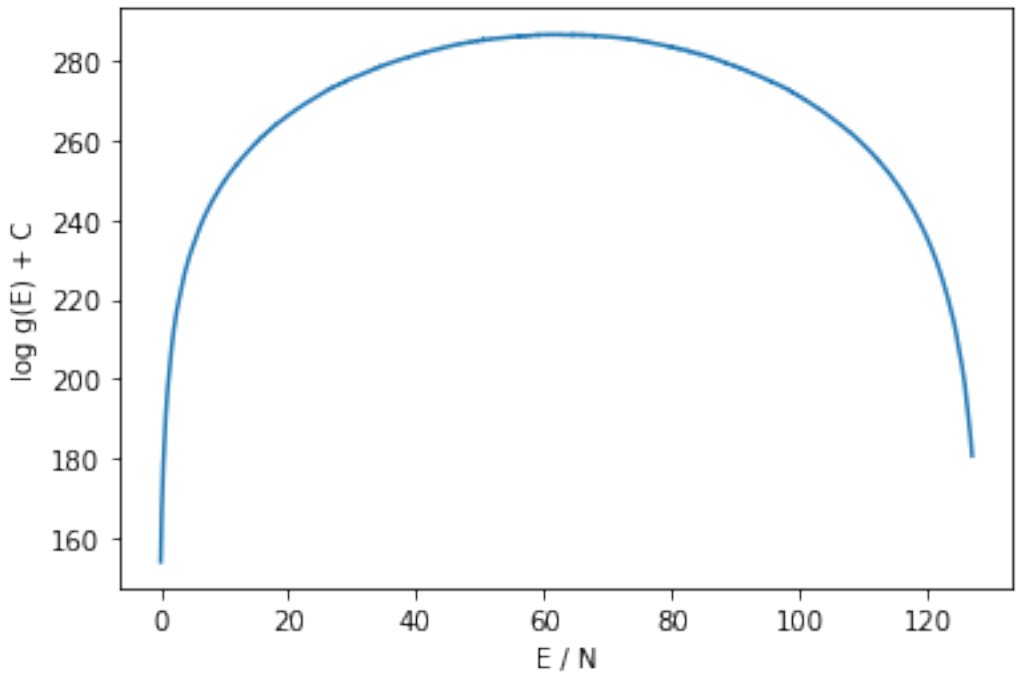
```
f: 27 / 27  (1000000 / 1000000)
Done: 26538865 total MC iterations.
```

```
import pickle
with open('wlresults.pickle', 'wb') as f:
    pickle.dump(list(Ls), f)
    pickle.dump(wlresults, f)
```

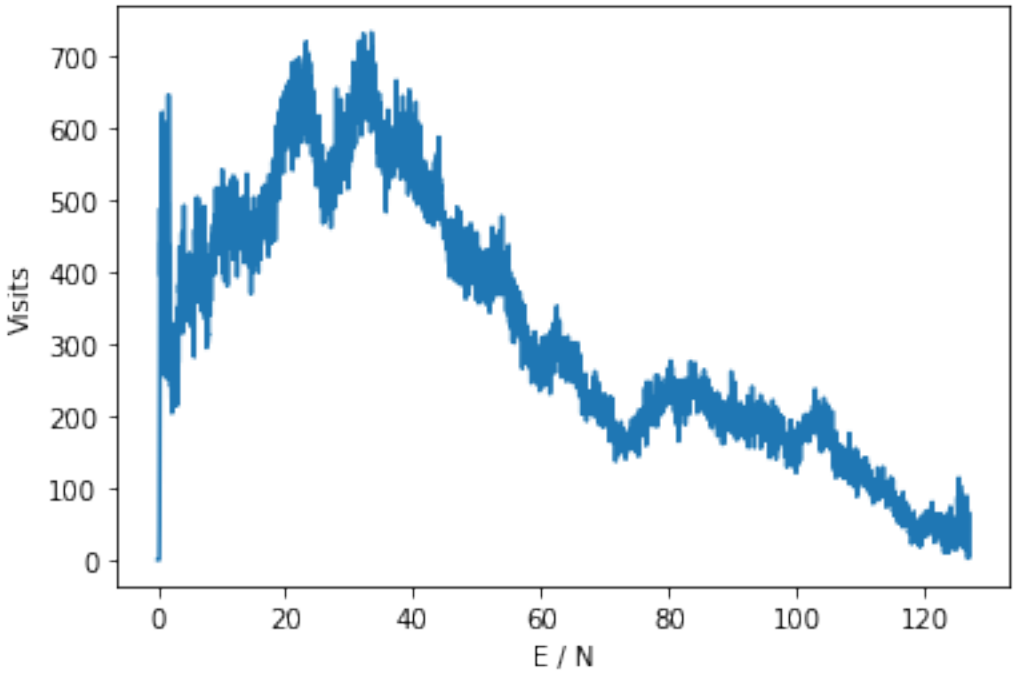
We plot the results for $L = 5$, since it is in between the "slow" behavior of $L = 3$ and the "fast" (phase transition?) behavior of $L = 7$.

```
L = Ls[2]
Es, S, H = wlresults[2]
```

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




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



1.2.1 Calculating canonical ensemble averages

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

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

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

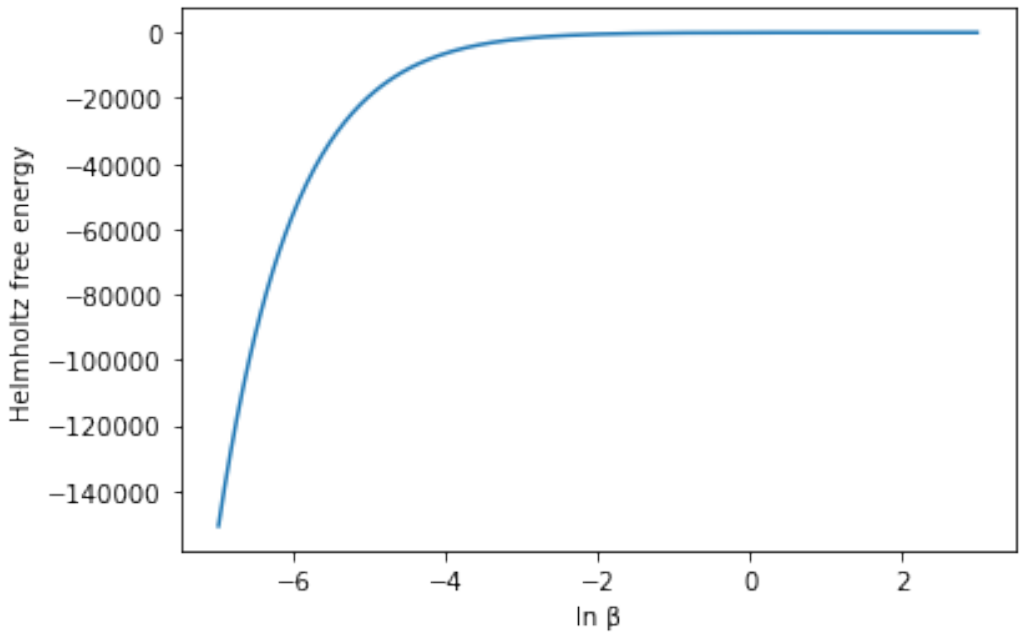
```
Z = lambda β: np.sum(gs * np.exp(-β * nEs))
```

Ensemble averages

```
βs = [np.exp(k) for k in np.linspace(-7, 3, 500)]
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))
```

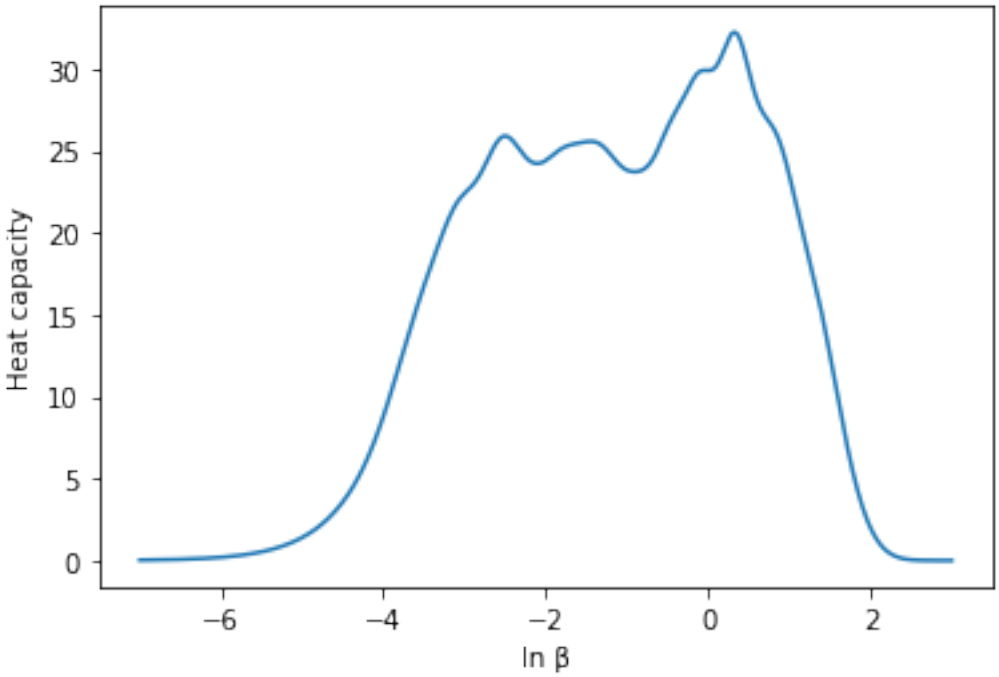
Helmholtz free energy

```
plt.plot(np.log(βs), [F(β) for β in βs])
plt.xlabel("ln β")
plt.ylabel("Helmholtz free energy")
plt.show()
```



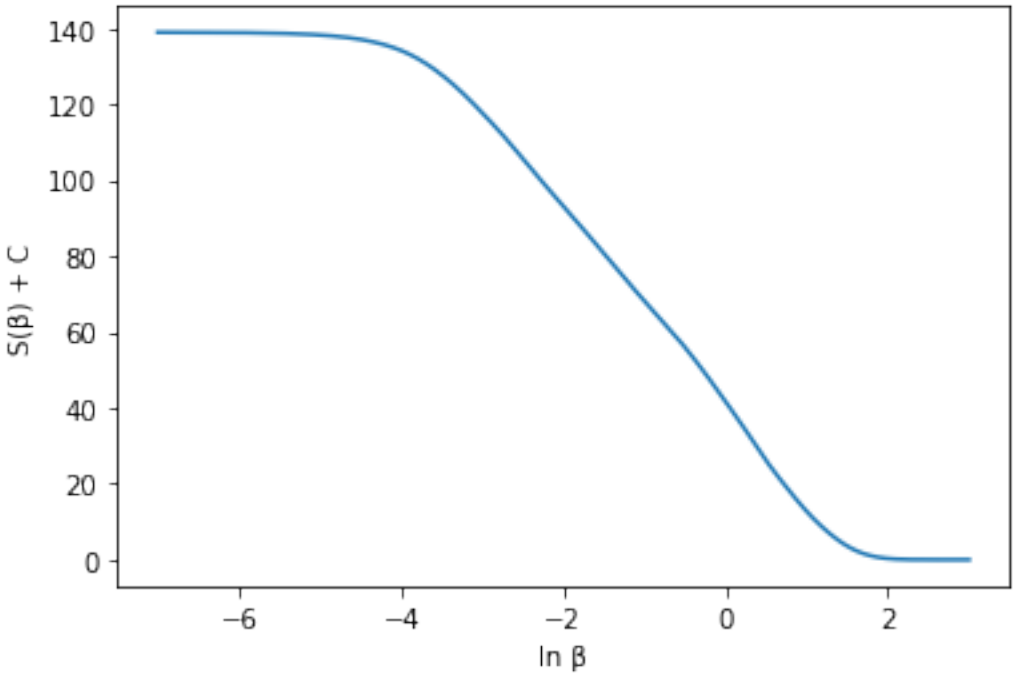
Heat capacity

```
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()
```


140
120
100
80
60
40
20
0
S(β) + C
−6
−4
−2
0
2
ln β