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.

TODO: - Compare different flatness functions - Profile using `binindex` instead of assuming linear system energy bins.

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

Utility functions.

```python
import bisect
```

```python
def binindex(Es, E):
    return bisect.bisect(Es, E, hi=len(Es) - 1) - 1
```

```python
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)
```

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.

```python
def wanglandau(system,
               Es,           # The energy bins
               M = 1_00_000, # Monte carlo step scale
               ε = 1e-8,     # f tolerance
               logf0 = 1,    # Initial log f
               logging = True # Log progress of f-steps
              ):
```

```
    # Initial values
    E0 = Es[0]
    Ef = Es[-1]
    ΔE = Es[1] - E0
    N = len(Es) - 1
    logf = logf0
    logftol = np.log(1 + ε)
    S = np.zeros(N) # Set all initial g's to 1
    H = np.zeros(N, dtype=int)
    i = binindex(Es, system.E)

    if logging:
        mciters = 0
        fiter = 0
        fiters = int(np.ceil(np.log2(logf0) - np.log2(logftol)))
        print("Wang-Landau START:")
        print("\t|Es| = {}\n\tM = {}\n\tε = {}\n\tlog f0 = {}".format(len(Es), M, ε, logf0))

    while logftol < logf:
        H[:] = 0
        logf /= 2
        iters = 0
        niters = int((M + 1) * np.exp(-logf / 2))
        if logging:
            fiter += 1
        while not flat(H) and iters < niters:
            system.propose()
            Ev = system.Ev
            j = binindex(Es, Ev)
#             if E0 ≤ Ev ≤ Ef and (
            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
        if logging:
            mciters += iters
            print("f: {} / {}\t({} / {})".format(fiter, fiters, iters, niters))

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

1.0.1 Parallel construction of the density of states

```python
from multiprocessing import Pool
import copy
```

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

```python
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))
```

Try monotonic instead of Wang-Landau steps

```python
def find_bin_systems(sys, Es, Ebins, N = 1_000_000):
    """Find systems with energies in the bins given by `Es` by stepping `sys`."""
#     S = np.zeros(len(Es), dtype=int)
    systems = [None] * (len(Ebins) - 1)
    n = 0
    i = binindex(Es, sys.E)
    while any(system is None for system in systems) and n < N:
        for s in range(len(systems)):
            if systems[s] is None and Ebins[s] ≤ sys.E < Ebins[s + 1]:
                systems[s] = copy.deepcopy(sys)

        sys.propose()
        j = binindex(Es, sys.Ev)
        if sys.E < sys.E\nu:
            sys.accept()
#         if S[j] < S[i]:
#             i = j
#             sys.accept()
#         S[i] += 1
        n += 1

    if N ≤ n:
        raise ValueError('Could not find bin systems after {} iterations.'.format(N))
    return systems
```

Now we can construct our parallel systems.

```python
def parallel_systems(system, Es, n = 8, k = 0.1, N = 1_000_000):
    Ebins = np.linspace(Es[0], Es[-1], n + 1)
    systems = find_bin_systems(system, Es, Ebins, N)
    binEs = [(lambda E0, Ef: Es[(E0 <= Es) & (Es <= Ef)])(*extend_bin(Ebins, i, k))
             for i in range(len(Ebins) - 1)]
    return zip(systems, binEs)
```

We also need a way to reset the random number generator seed in a way that is time-independent and different for each process.

```python
import os, struct
```

```python
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
```

Once we have parallel results, we stitch the pieces of $\ln g(E)$ together.

```python
def stitch_results(wlresults):
    E0, S0, _ = wlresults[0]
    E, S = E0, S0
    for i in range(1, len(wlresults)):
        Ev, Sv, _ = wlresults[i]
        # Assumes overlap is at end regions
        _, i0s, ivs = np.intersect1d(E0[:-1], Ev[:-1], return_indices=True)
        # Simplest: join middles of overlap regions
        l = len(i0s)
        m = l // 2
#         print(l, m, i0s, ivs, i0s[m], S0, Sv)
        Sv -= Sv[ivs[m]] - S0[i0s[m]]
        # Simplest: average the overlaps to produce the final value
        E = np.hstack((E, Ev[l+1:]))
        S[-l:] = (Sv[ivs] + S0[i0s]) / 2
        S = np.hstack((S, Sv[l:]))
        E0, S0 = Ev, Sv
    return E, S
```

1.1 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 `wanglandau`.

1.1.1 Simulation

```
isingn = 32
sys = Ising(isingn)
```

The Ising energies over the full range, with correct end bin. We remove the penultimate energies since $E = 2$ or $E_{\max} - 2$ cannot happen.

```
isingE0 = -2 * isingn**2
isingEf = 2 * isingn**2
isingΔE = 4
Es = np.arange(isingE0, isingEf + isingΔE + 1, isingΔE)
Es = np.delete(np.delete(Es, -3), 1)
```

```
psystems = parallel_systems(sys, Es, n = 16, k = 0.5, N = 10_000_000)
```

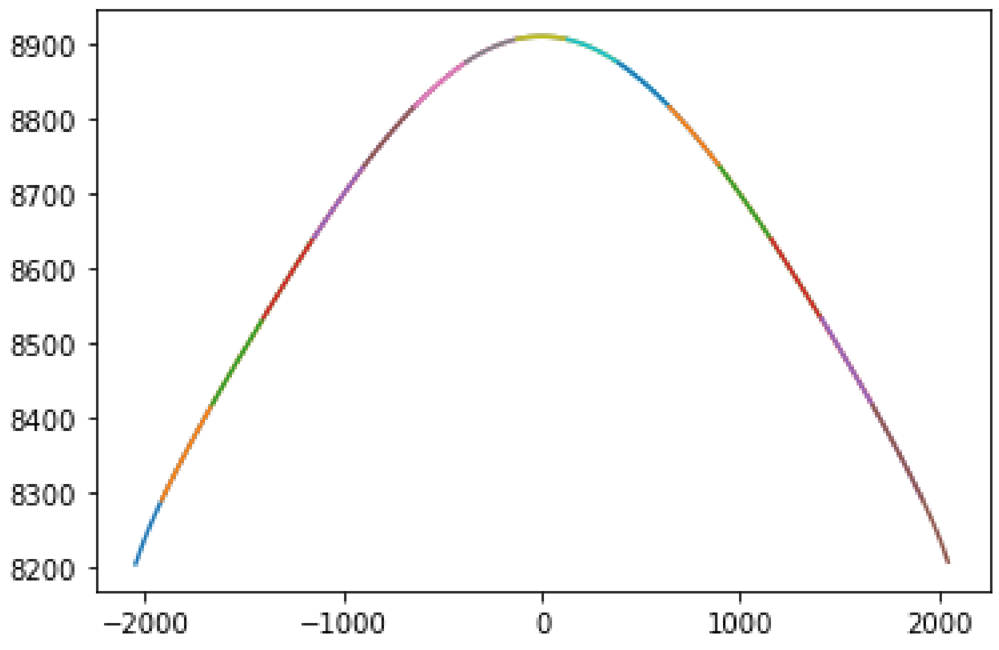
```
def parallel_wanglandau(subsystem): # Convenient form for `Pool.map`
    urandom_reseed()
    results = wanglandau(*subsystem, M = 1_000_000, logging=False)
    print('*', end='', flush=True)
    return results
```

```
with Pool() as pool:
    wlresults = pool.map(parallel_wanglandau, psystems)
```

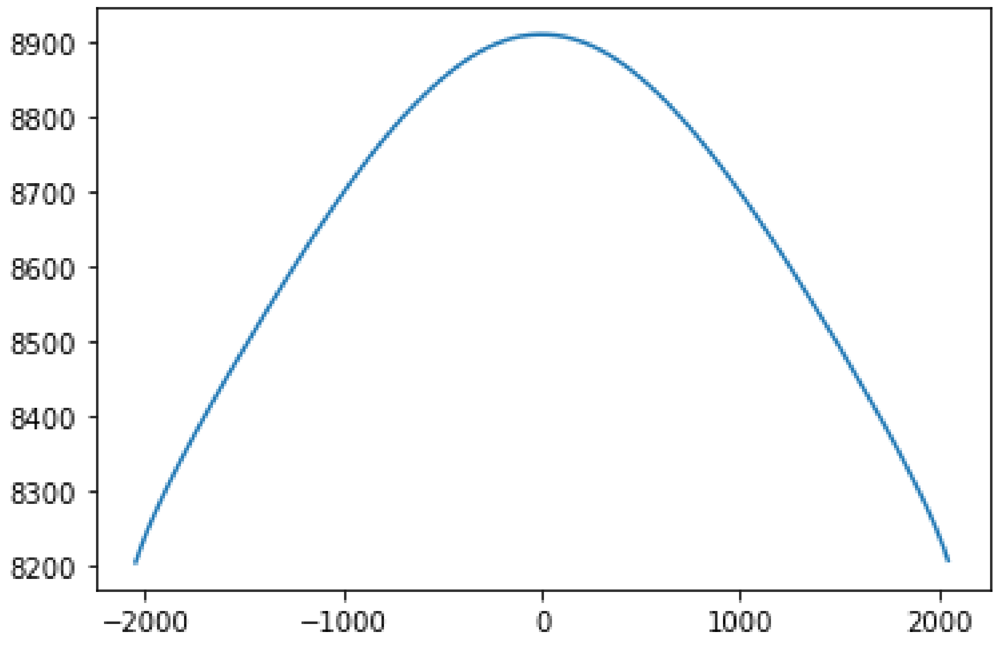
****

```
sEs, sS = stitch_results(wlresults)
```

```
for Es, S, H in wlresults:
    plt.plot(Es[:-1], S)
```



```
plt.plot(sEs[:-1], sS);
```



```python
import os, tempfile, pickle
```

```python
with tempfile.NamedTemporaryFile(mode='wb', prefix='wlresults-ising-', suffix='.pickle',
↪  dir='data', delete=False) as f:
    print(os.path.basename(f.name))
    pickle.dump(wlresults, f)
    pickle.dump(sEs, f)
    pickle.dump(sS, f)
```

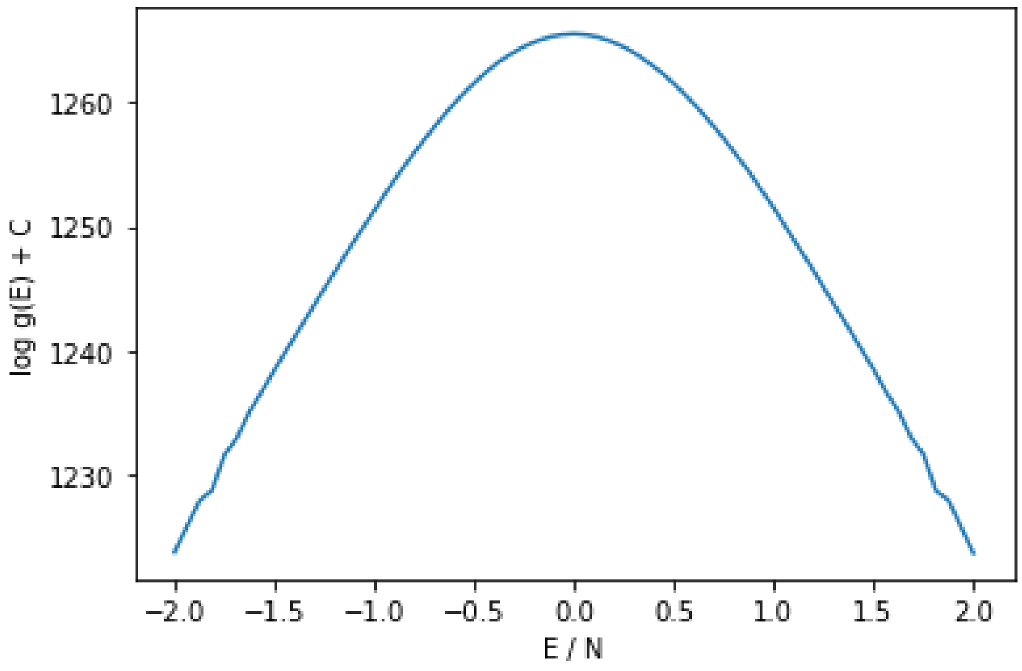
```
wlresults-ising-1m2_zvey.pickle
```

```python
Es, S, H = wanglandau(sys, Es, M = 200_000);
Es = Es[:-1] # Use the actual energy levels instead of the bins
```

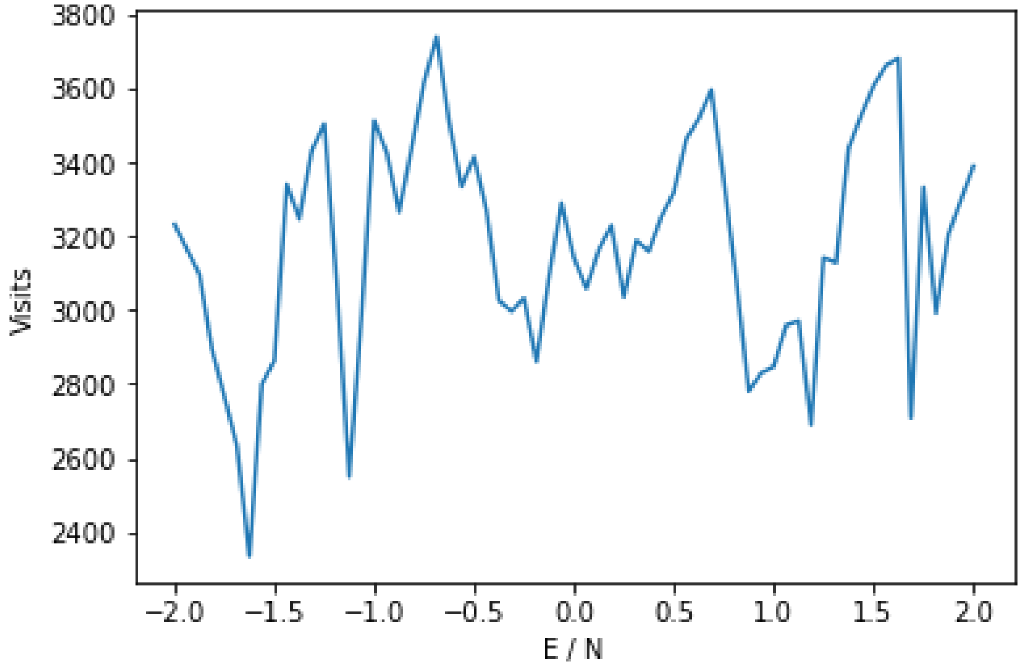
```
Wang-Landau START:
    |Es| = 64
    M = 200000
    ε = 1e-08
    log f0 = 1
f: 1 / 27   (84478 / 155760)
f: 2 / 27   (52028 / 176500)
f: 3 / 27   (71821 / 187883)
```

```
f: 4 / 27   (91508 / 193847)
f: 5 / 27   (152513 / 196900)
f: 6 / 27   (85943 / 198444)
f: 7 / 27   (190471 / 199221)
f: 8 / 27   (110618 / 199610)
f: 9 / 27   (199805 / 199805)
f: 10 / 27  (174248 / 199903)
f: 11 / 27  (199952 / 199952)
f: 12 / 27  (199976 / 199976)
f: 13 / 27  (199988 / 199988)
f: 14 / 27  (199994 / 199994)
f: 15 / 27  (199997 / 199997)
f: 16 / 27  (199999 / 199999)
f: 17 / 27  (200000 / 200000)
f: 18 / 27  (200000 / 200000)
f: 19 / 27  (200000 / 200000)
f: 20 / 27  (200000 / 200000)
f: 21 / 27  (200000 / 200000)
f: 22 / 27  (200000 / 200000)
f: 23 / 27  (200000 / 200000)
f: 24 / 27  (200000 / 200000)
f: 25 / 27  (200000 / 200000)
f: 26 / 27  (200000 / 200000)
f: 27 / 27  (200000 / 200000)
Done: 4613339 total MC iterations.
```

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



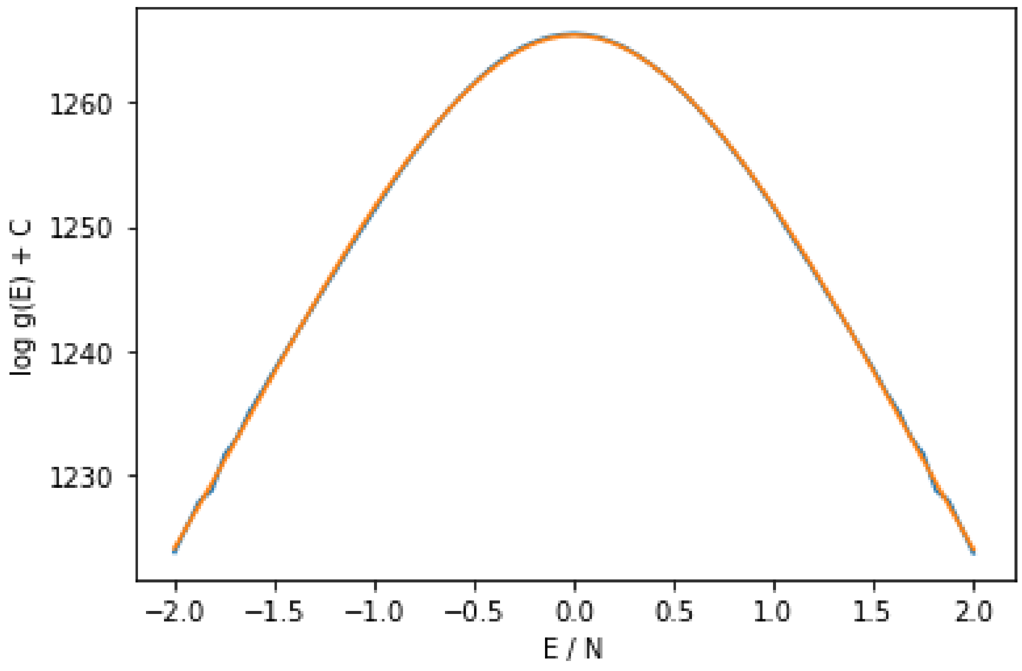
```python
plt.plot(Es / isingn**2, H)
plt.xlabel("E / N")
plt.ylabel("Visits");
```



1.1.2 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)
```

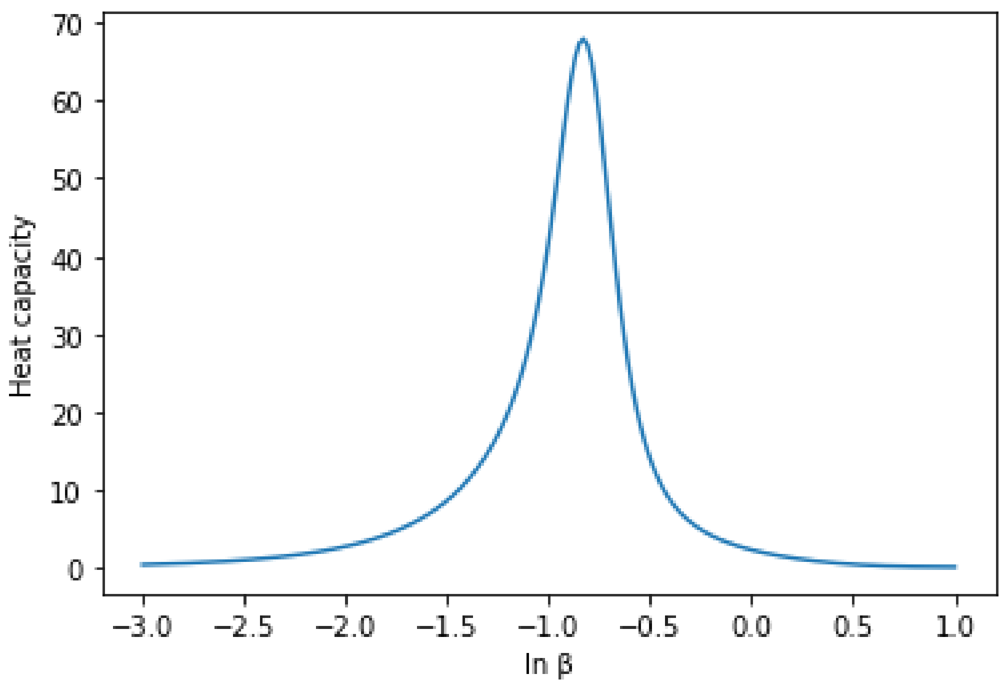
```
Z = lambda β: np.sum(gs * np.exp(-β * nEs))
```

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))
```

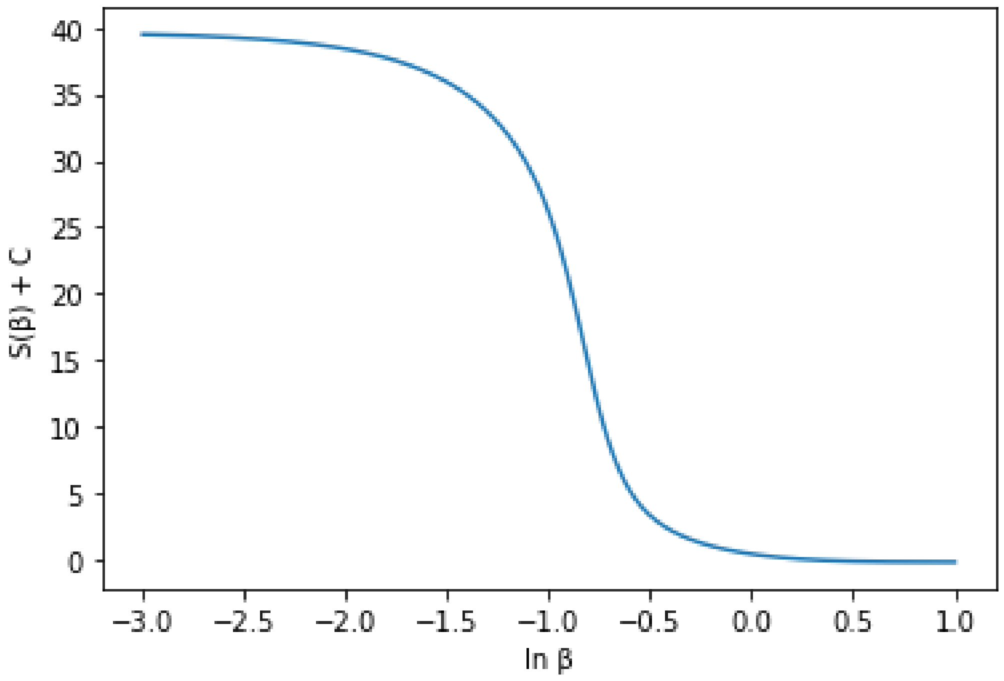
Heat capacity

```
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
```

1.2.1 Simulation

```
Ls = range(1, 11, 2)
wlresults = [wanglandau(StatisticalImage(128 * np.ones((L, L), dtype=int)),
                        Es = np.arange(0, 127*L**2 + 1),
                        M=1_000_000)
             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)
```

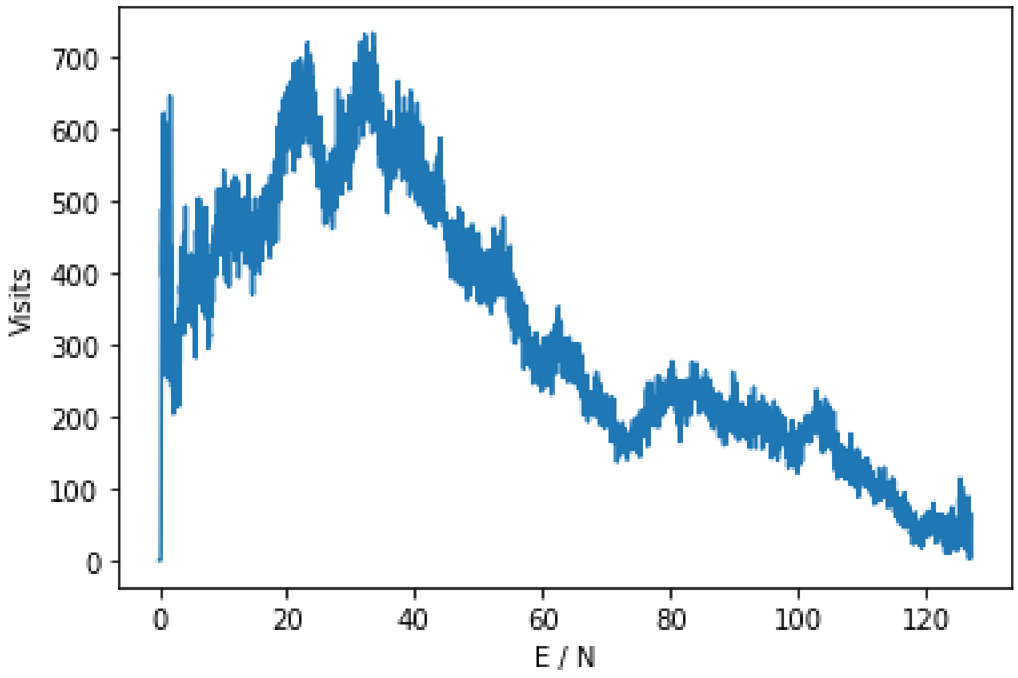
```
f: 24 / 27  (1000000 / 1000000)
f: 25 / 27  (1000000 / 1000000)
f: 26 / 27  (1000000 / 1000000)
f: 27 / 27  (1000000 / 1000000)
Done: 26538865 total MC iterations.
```

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

```
5
```

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

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



1.2.2 Parallel

```python
L = 3
sys = StatisticalImage(np.zeros((L, L), dtype=int))
Es = np.arange(0, (2**8 - 1)*L**2 + 1)
psystems = parallel_systems(sys, Es, n = 8, k = 0.25, N = 1_000_000)
```

```python
def parallel_wanglandau(subsystem): # Convenient form for `Pool.map`
    urandom_reseed()
    results = wanglandau(*subsystem, M = 10_000_000, logging=False)
    print('*', end='', flush=True)
    return results
```

```python
with Pool() as pool:
    wlresults = pool.map(parallel_wanglandau, psystems)
```

```
********
```

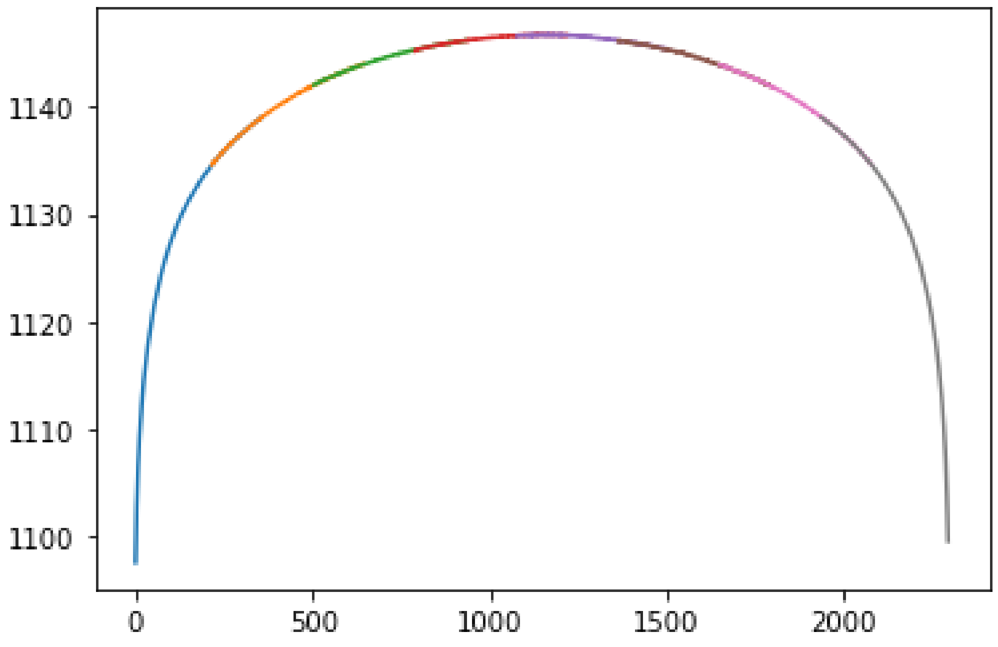
```python
sEs, sS = stitch_results(wlresults)
```

```python
import os, tempfile, pickle
```

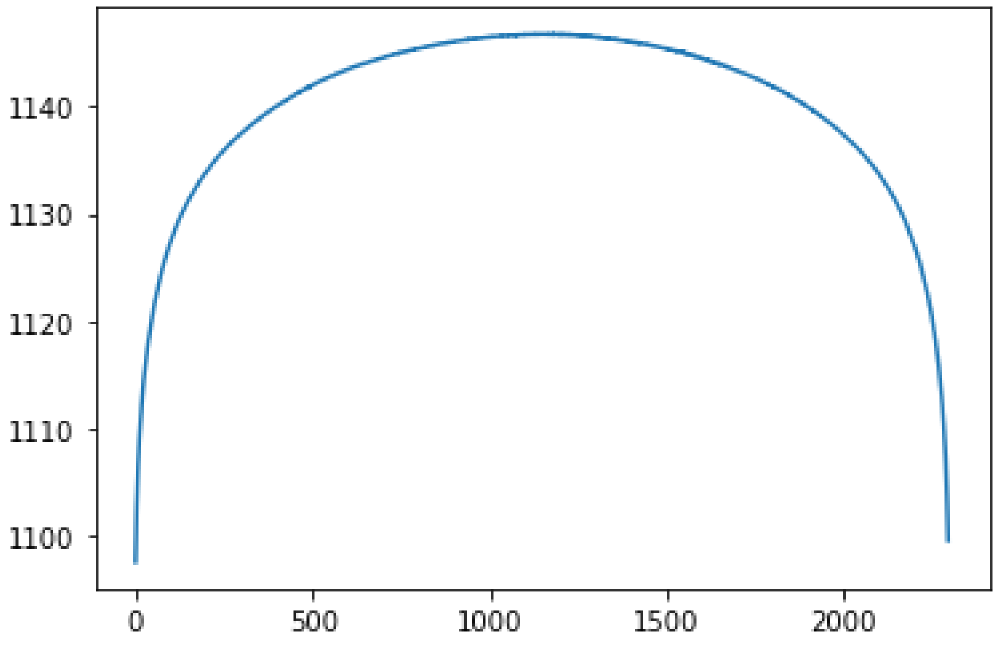
```python
with tempfile.NamedTemporaryFile(mode='wb', prefix='wlresults-image-', suffix='.pickle',
↪ dir='data', delete=False) as f:
    print(os.path.basename(f.name))
    pickle.dump(list(Ls), f)
    pickle.dump(wlresults, f)
```

```
wlresults-image-7hwobriy.pickle
```

```python
for Es, S, H in wlresults:
    plt.plot(Es[:-1], S)
```



```
plt.plot(sEs[:-1], sS);
```



```python
wlEs, S = sEs[:-1], sS
```

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

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

1.2.3 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.

```python
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 gray_g(E, N, M, exact=True):
    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.

```python
def reflect(a):
    return np.hstack([a[:-2], a[-1], a[-2::-1]])
def 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)
```

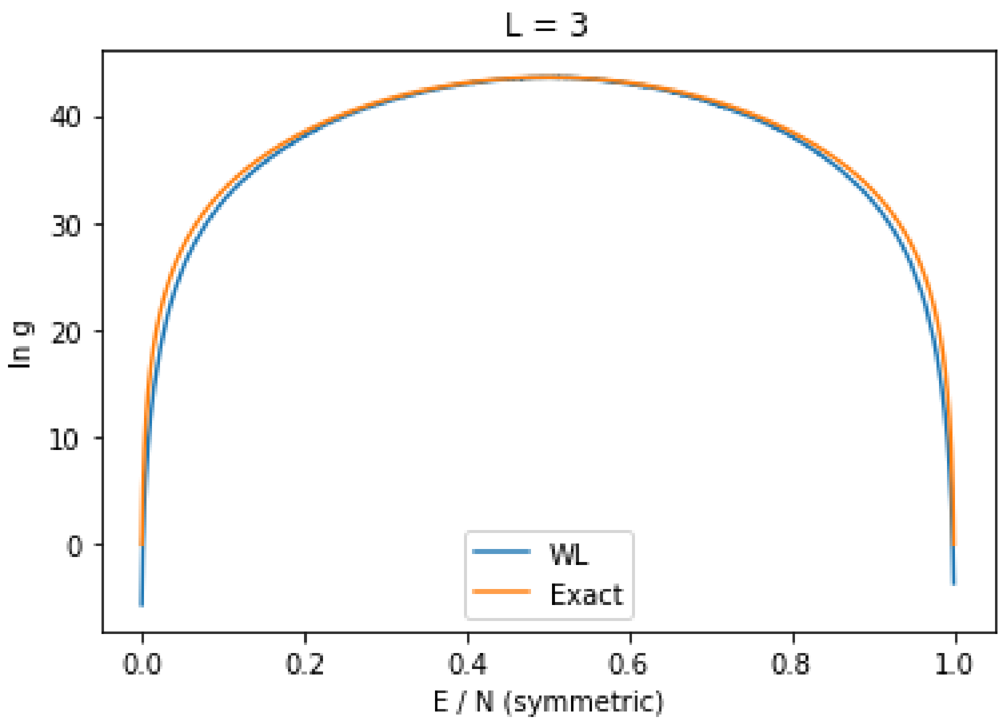
```python
# Gray
# Es, gs = gray_gs(N=L**2, M=2**7 - 1)
# Black
Es, gs = gray_gs(N=L**2, M=2**8 - 1)
gs /= 2
```

Renormalize the WL result

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

Compare the exact result to the WL result.

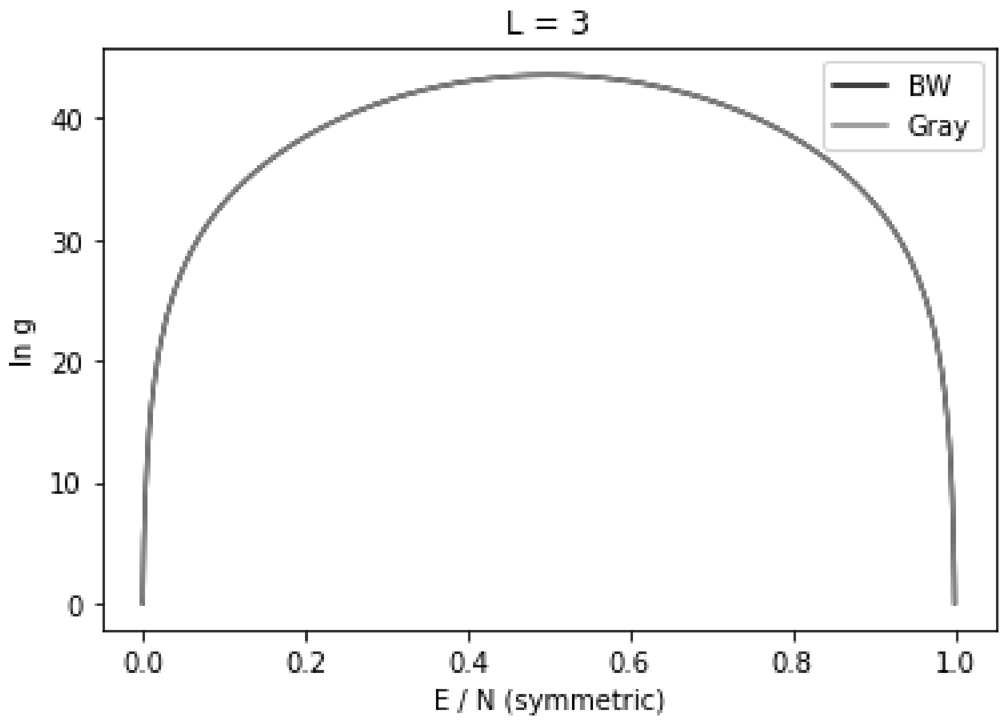
```python
plt.plot(wlEs / len(wlEs), np.log(wlgs), label='WL')
plt.plot(Es / len(Es), np.log(gs), label='Exact')
plt.xlabel('E / N (symmetric)')
plt.ylabel('ln g')
plt.title('L = {}'.format(L))
plt.legend();
```



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

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

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



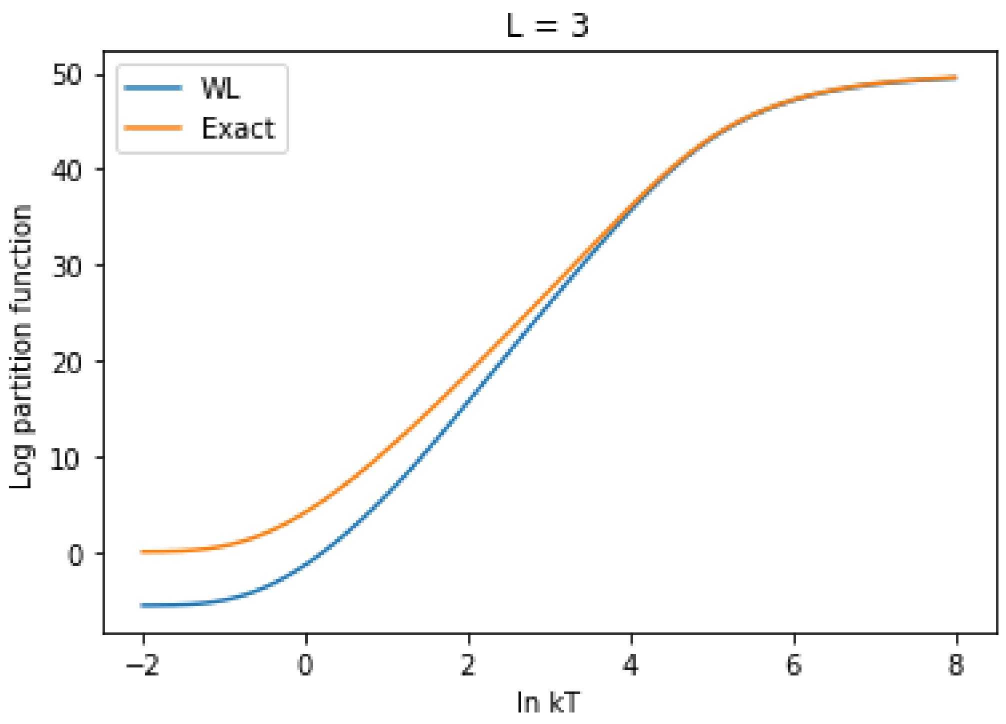
1.2.4 Calculating canonical ensemble averages

```python
class CanonicalEnsemble:
    def __init__(self, Es, gs, name):
        self.Es = Es
        self.gs = gs
        self.name = name
    def Z(self, β):
        return np.sum(self.gs * np.exp(-β * self.Es))
    def average(self, f, β):
        return np.sum(f(self) * self.gs * np.exp(-β * self.Es)) / self.Z(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))
```

```python
βs = [np.exp(k) for k in np.linspace(-8, 2, 500)]
wlens = CanonicalEnsemble(wlEs, wlgs, 'WL') # Wang-Landau results
xens = CanonicalEnsemble(Es, gs, 'Exact') # Exact
ensembles = [wlens, xens]
```

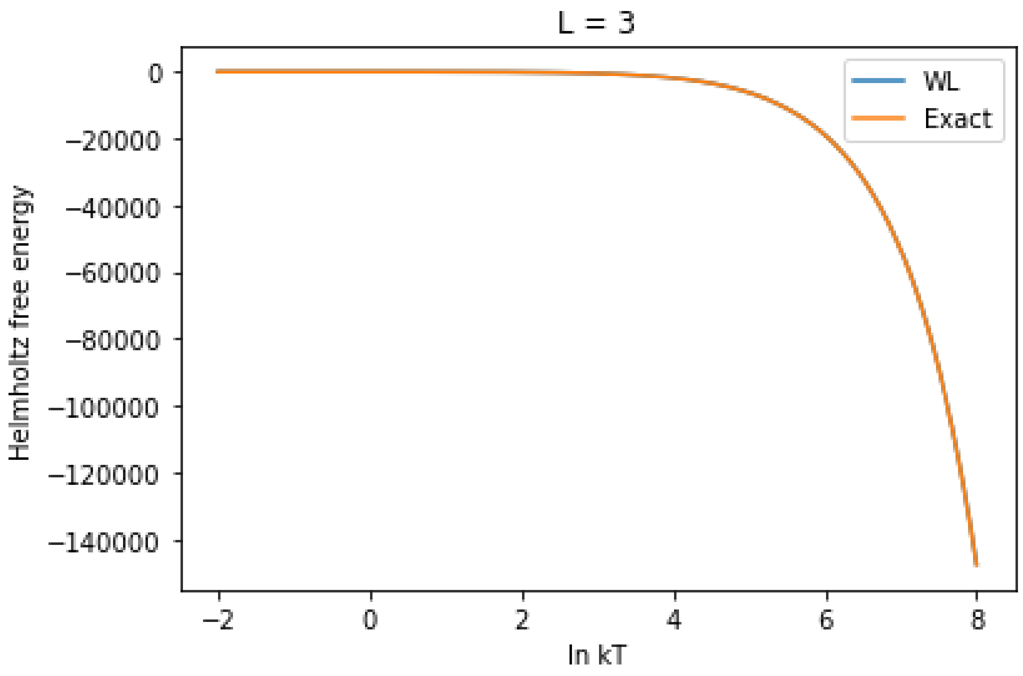
Partition function

```python
for ens in ensembles:
    plt.plot(-np.log(βs), np.log(np.vectorize(ens.Z)(βs)), label=ens.name)
plt.xlabel("ln kT")
plt.ylabel("Log partition function")
plt.title('L = {}'.format(L))
plt.legend();
```



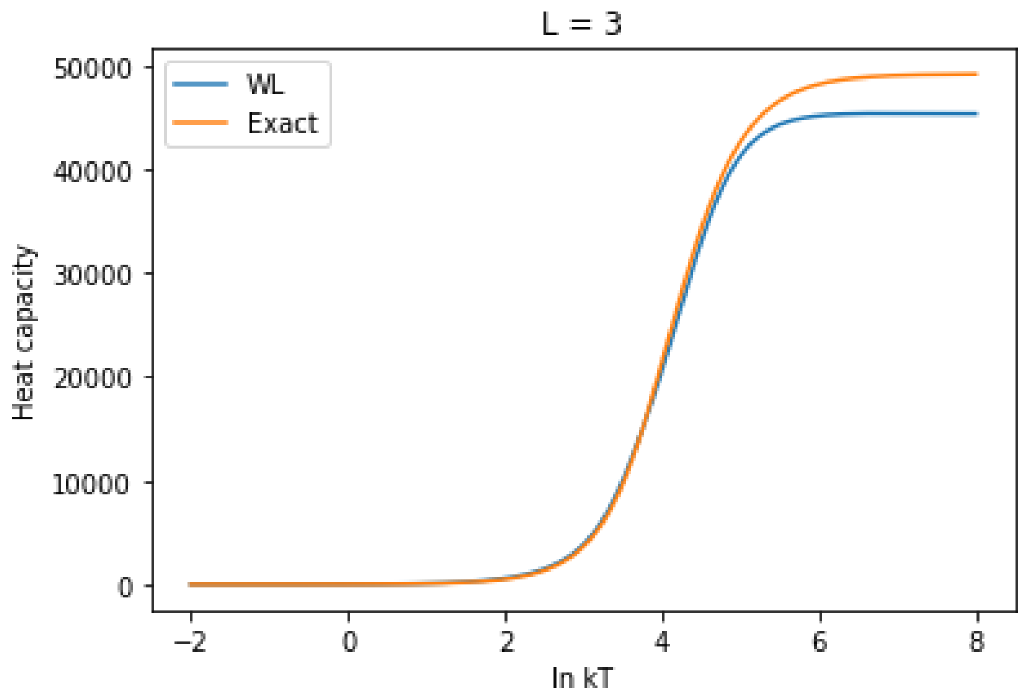
Helmholtz free energy

```python
for ens in ensembles:
    plt.plot(-np.log(βs), np.vectorize(ens.free_energy)(βs), label=ens.name)
plt.xlabel("ln kT")
plt.ylabel("Helmholtz free energy")
plt.title('L = {}'.format(L))
plt.legend();
```



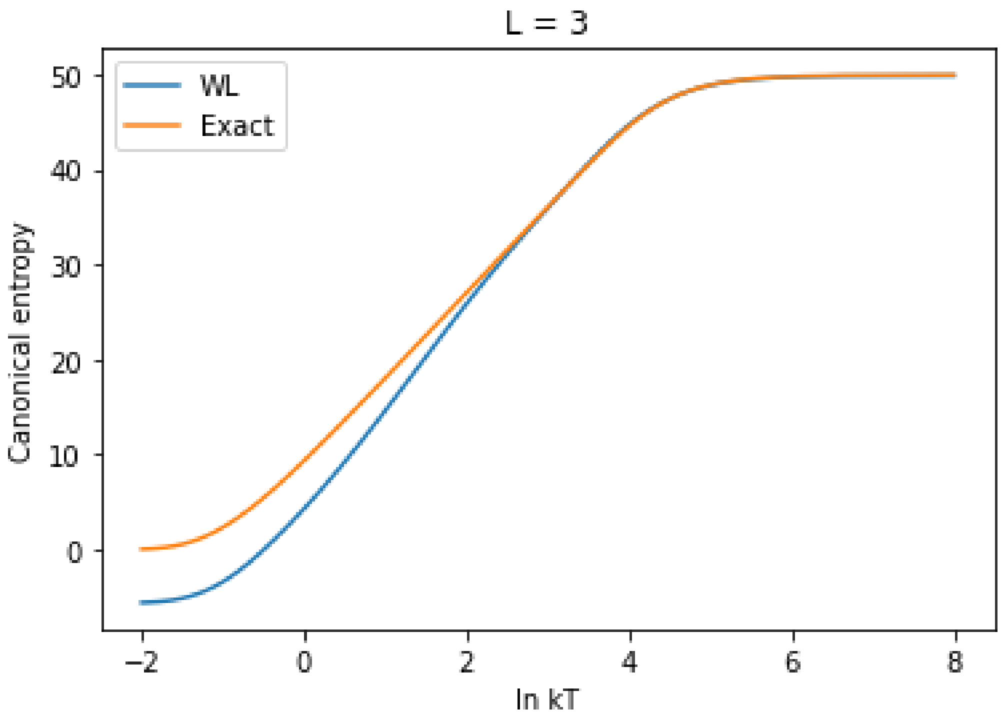
Heat capacity

```python
for ens in ensembles:
    plt.plot(-np.log(βs), np.vectorize(ens.heat_capacity)(βs), label=ens.name)
plt.xlabel("ln kT")
plt.ylabel("Heat capacity")
plt.title('L = {}'.format(L))
plt.legend();
```



Entropy

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


L = 3
WL
Exact
Canonical entropy
ln kT
50
40
30
20
10
0
−2
0
2
4
6
8