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: * Improve stitching of parallel solutions

```
from numba import jit
nopython = True
```

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

Utility functions.

```
from bisect import bisect
```

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

```
# @jit
# 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)
@jit
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)
```

1.1 Algorithm

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

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

```
@jit(forceobj=True)
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
               flatness = 0.2  # Desired histogram flatness
              ):
    # 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, flatness) 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
```

1.1.1 Parallel construction of the density of states

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

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

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

Try monotonic instead of Wang-Landau steps

```
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.Ev:
            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.

```
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.

```
import os, struct
```

```
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.

```
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
```

1.2 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 `wanglandau`.

1.2.1 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 = 50_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);
```

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

```
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)
```

1.2.2 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()
```

1.3 Thermal calculations on images

```
# @jitclass
class StatisticalImage:
    def __init__(self, I0, M = 2**8 - 1):
        self.I0 = I0
        self.I = I0.copy()
        self.N = len(I0)
        self.M = M
        self.E = self.energy()
        self.Ev = self.E
    def energy(self):
        return np.sum(np.abs(self.I - self.I0))
    def propose(self):
        i = np.random.randint(self.N)
        self.i = i
        x0 = self.I0[i]
        x = self.I[i]
        r = np.random.randint(2)
        if x == 0:
            dx = r
        elif x == self.M:
            dx = -r
        else:
            dx = 2*r - 1
        dE = dx
        self.dx = dx
        self.dE = dE
        self.Ev = self.E + dE
    def accept(self):
        self.I[self.i] += self.dx
        self.E = self.Ev
```

```
N = 3
M = 3
sys = StatisticalImage(np.zeros(N, dtype=int), M)
Es = np.arange(0, N*M + 1 + 1)
exactS = np.log(exact_bw_gs(N, M)[1])
```

```
Es, S, H = wanglandau(sys, Es, M = 100_000, ε = 1e-8, flatness = 0.01, logging=False)
S -= np.min(S)
plt.plot(Es[:-1], S)
plt.plot(Es[:-1], exactS);
```

1.3.1 Parallel Simulation

```
N = 16
M = 2**5 - 1
sys = StatisticalImage(np.zeros(N, dtype=int), M) # BW
Es = np.arange(N*M + 1 + 1)
psystems = parallel_systems(sys, Es, n = 8, k = 0.5, N = 1_000_000)
```

```
def parallel_wanglandau(subsystem): # Convenient form for `Pool.map`
    urandom_reseed()
    results = wanglandau(*subsystem, M = 10_000_000, ε = 1e-10, logging=False)
    print('*', end='', flush=True)
    return results
```

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

****

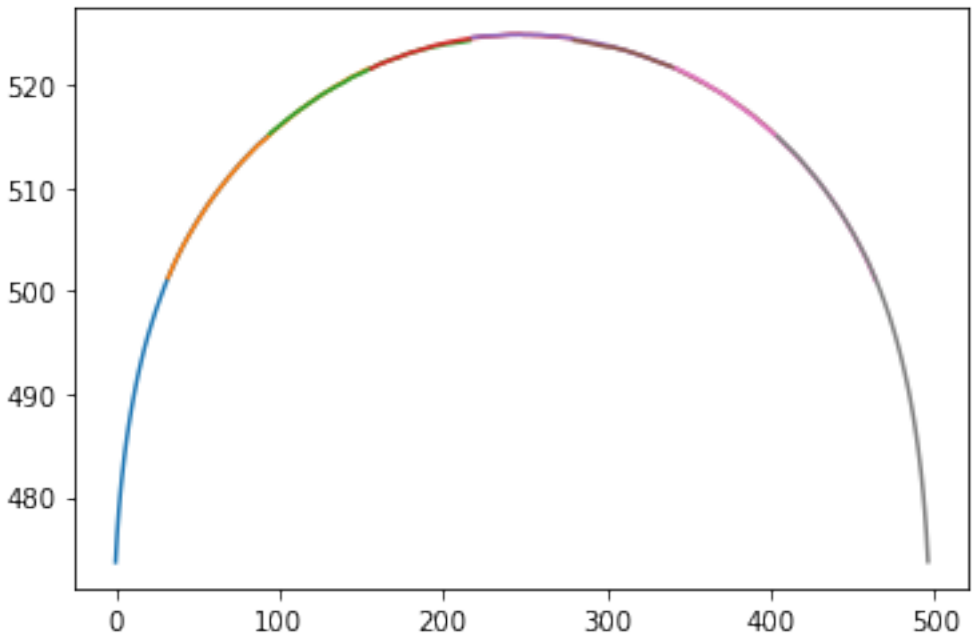
```
sEs, sS = stitch_results(wlresults)
```

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

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

wlresults-image-x7_3j6tp.pickle

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




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

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

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

1.3.2 Exact solution

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

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

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

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

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

Expected results for black/white and gray.

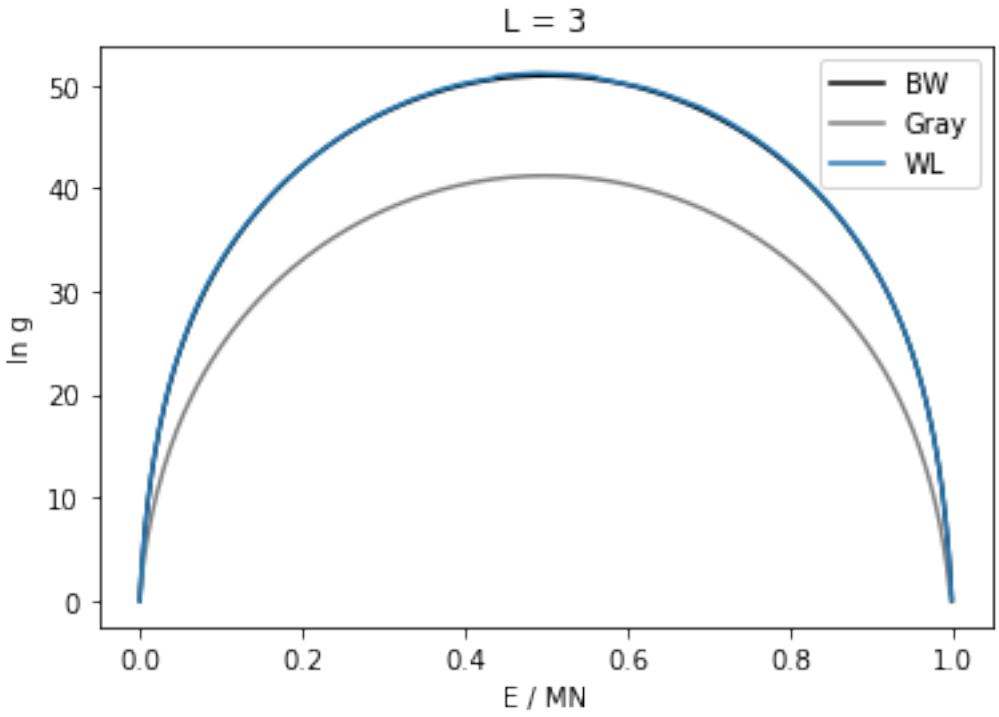
```
bw_Es, bw_gs = exact_bw_gs(N=N, M=M)
gray_Es, gray_gs = exact_gray_gs(N=N, M=-1 + (M + 1) // 2)
```

Choose what to compare to.

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

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

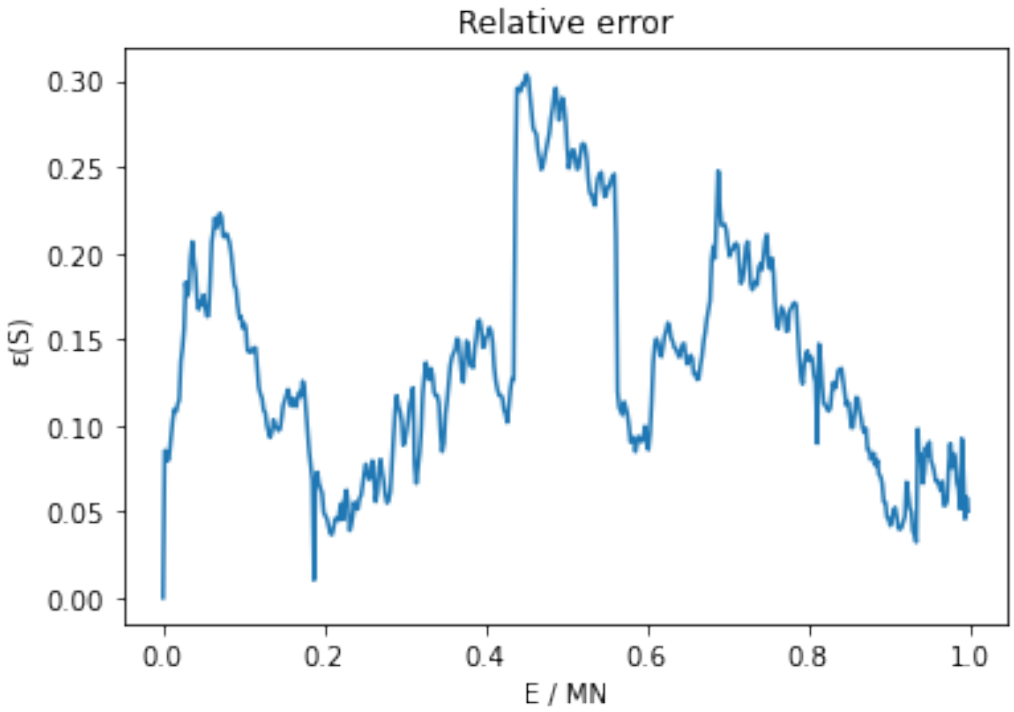
```
plt.plot(bw_Es / len(bw_Es), np.log(bw_gs), 'black', label='BW')
plt.plot(gray_Es / len(gray_Es), np.log(gray_gs), 'gray', label='Gray')
plt.plot(wlEs / len(wlEs), np.log(wlgs), label='WL')
plt.xlabel('E / MN')
plt.ylabel('ln g')
plt.title('L = {}'.format(L))
plt.legend();
```




```
wlgs[[0,-1]]
```

```
array([1.        , 1.05022784])
```

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



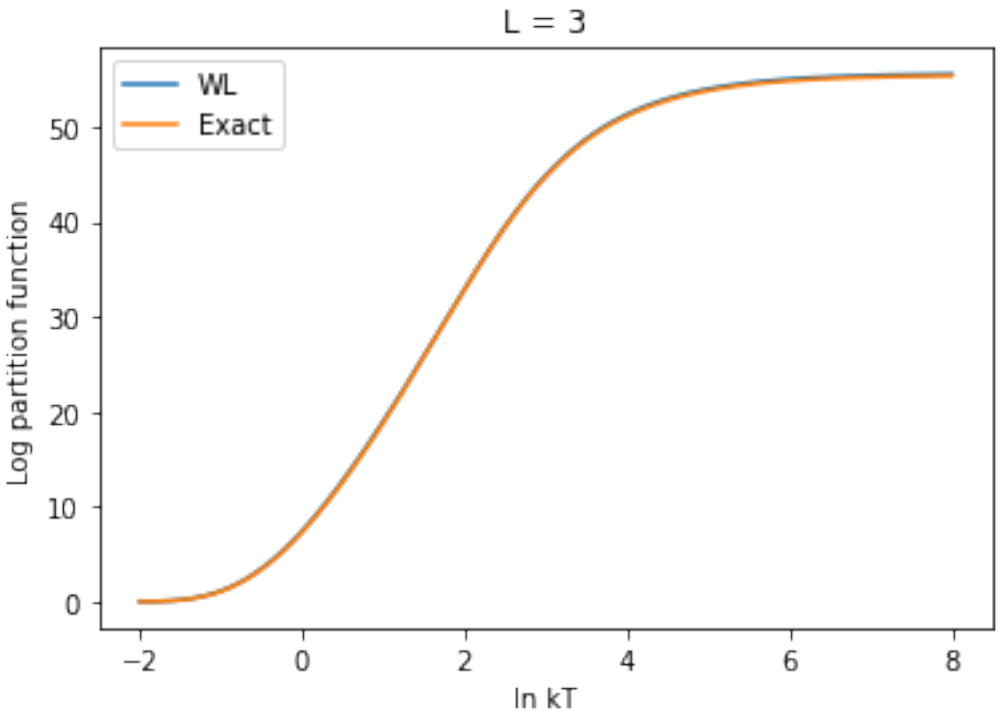
1.3.3 Calculating canonical ensemble averages

```
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))
```

```
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]
```

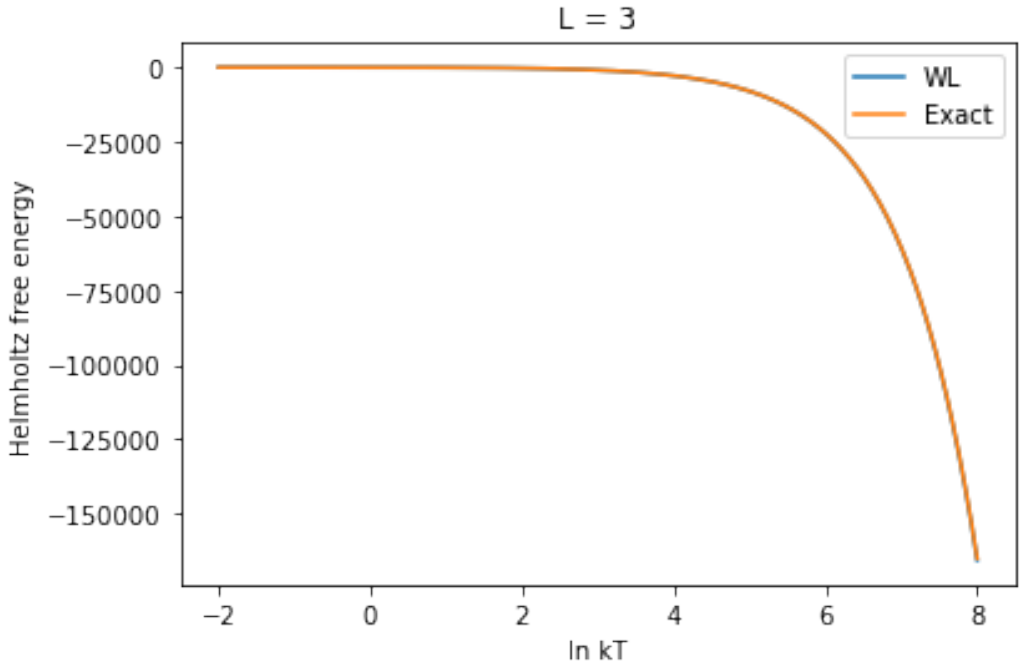
Partition function

```
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();
```



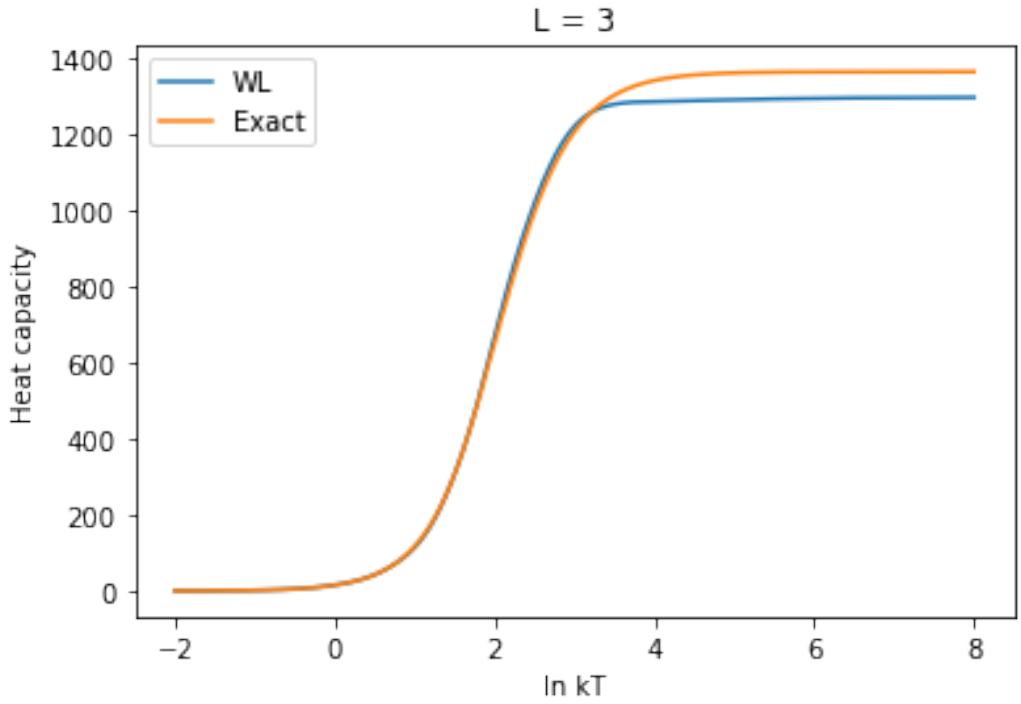
Helmholtz free energy

```
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();
```



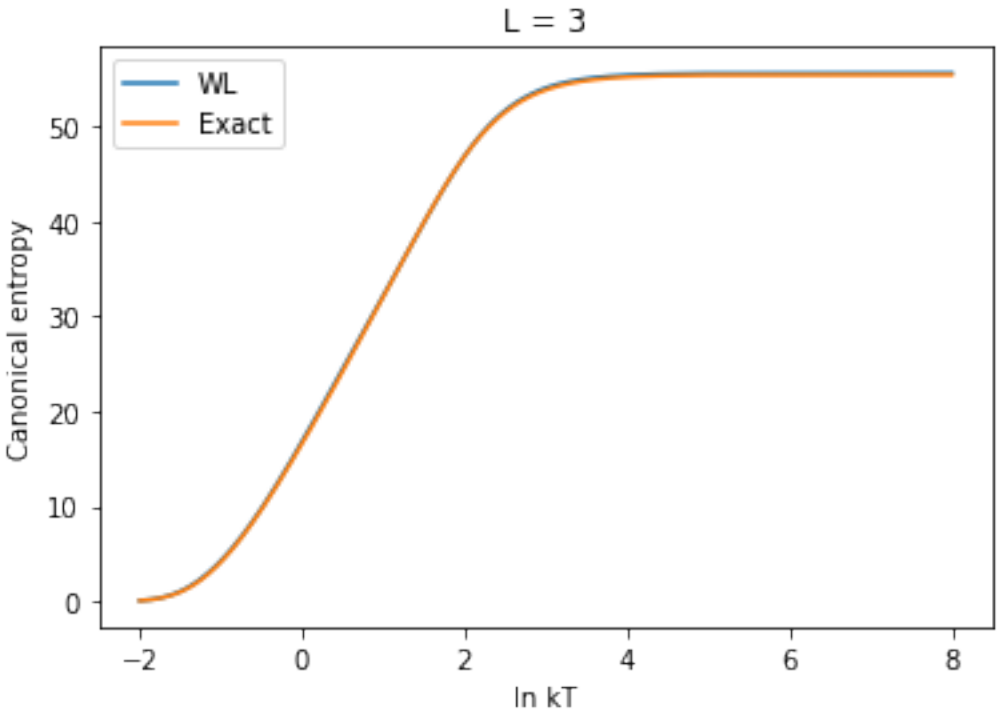
Heat capacity

```
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

```
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
0
10
20
30
40
50
−2
0
2
4
6
8
ln kT