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.

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

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 montecarlo steps proportional to $f^{-1/2}$ (a "Zhou-Bhat schedule").

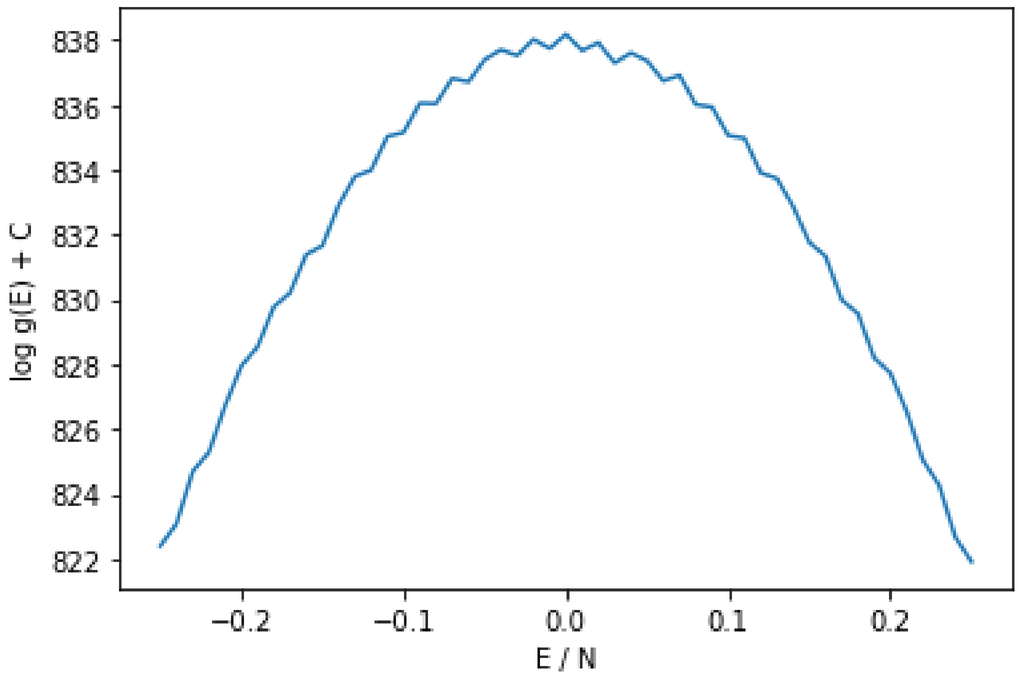
```python
def flat(H, tol = 0.1):
    """Determines if an evenly-spaced histogram is approximately flat."""
    Hμ = np.mean(H)
    Hf = np.max(H)
    H0 = np.min(H)
    return Hf / (1 + tol) < Hμ < H0 / (1 - tol)
```

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

    # Parameters
    M = 50_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   (38940 / 38940)
f: 2 / 20   (44125 / 44125)
f: 3 / 20   (46971 / 46971)
f: 4 / 20   (47725 / 48462)
f: 5 / 20   (49225 / 49225)
f: 6 / 20   (49611 / 49611)
f: 7 / 20   (49806 / 49806)
f: 8 / 20   (49903 / 49903)
f: 9 / 20   (49952 / 49952)
f: 10 / 20  (49976 / 49976)
f: 11 / 20  (49988 / 49988)
f: 12 / 20  (49994 / 49994)
f: 13 / 20  (49997 / 49997)
f: 14 / 20  (49999 / 49999)
f: 15 / 20  (50000 / 50000)
f: 16 / 20  (50000 / 50000)
f: 17 / 20  (50000 / 50000)
f: 18 / 20  (50000 / 50000)
f: 19 / 20  (50000 / 50000)
f: 20 / 20  (50000 / 50000)
Done: 976212 total MC iterations.
```

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



1.1 Calculating canonical ensemble averages

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

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

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

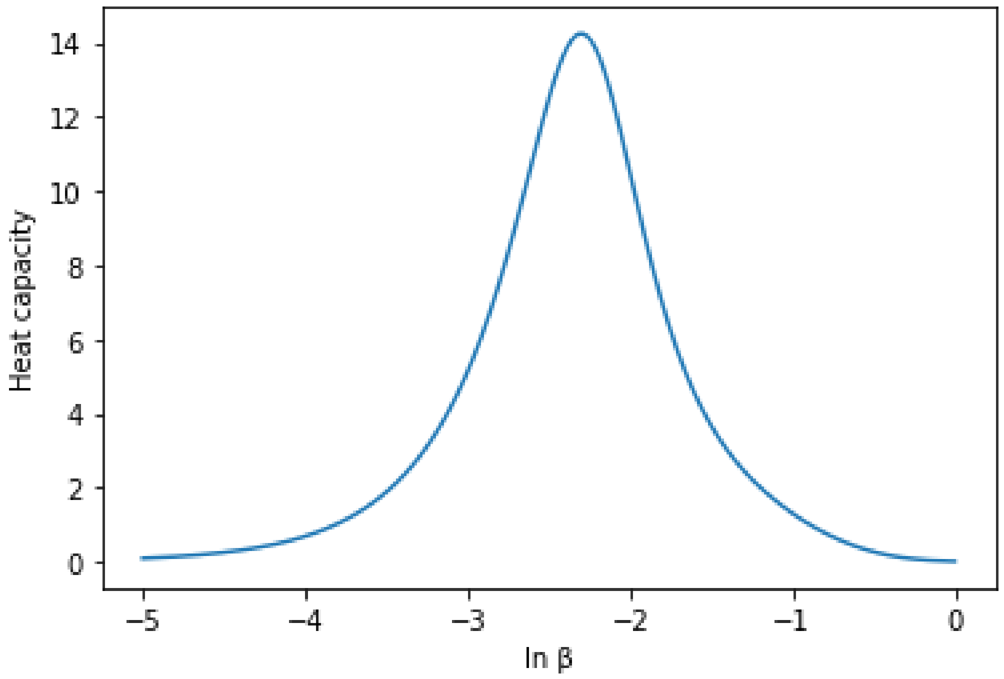
```
Z = lambda β: np.sum(gs * np.exp(-β * nEs))
```

Ensemble averages

```
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))
```

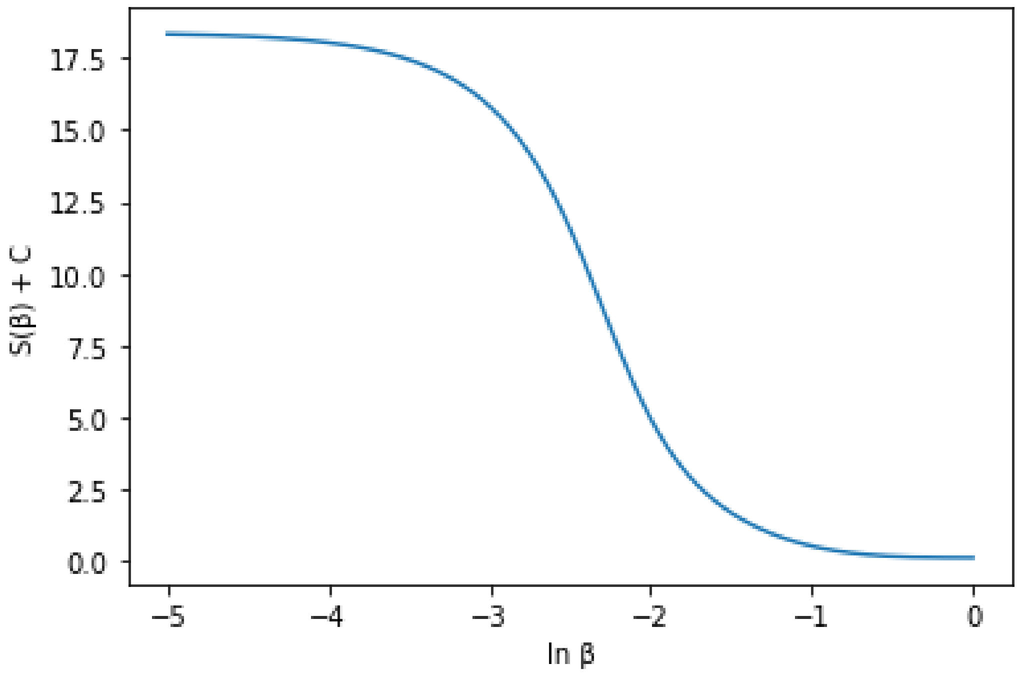
Heat capacity

```python
βs = [np.exp(k) for k in np.linspace(-5, 0, 200)]
plt.plot(np.log(βs), [CV(β) for β in βs])
plt.xlabel("ln β")
plt.ylabel("Heat capacity")
plt.show()
```



Entropy

```python
βs = [np.exp(k) for k in np.linspace(-5, 0, 200)]
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