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.

```
from numba import jit, jit_module
```

```
import numpy as np
import copy # for parallel systems
import os, struct # for using `urandom`
```

Utility functions.

```
@jit(nopython=True, cache=True)
def bisect_right(a, x, lo=0, hi=None):
    if lo < 0:
        raise ValueError('lo must be non-negative')
    if hi is None:
        hi = len(a)
    while lo < hi:
        mid = (lo + hi) // 2
        if x < a[mid]:
            hi = mid
        else:
            lo = mid + 1
    return lo

@jit(nopython=True, cache=True)
def binindex(a, x):
    return bisect_right(a, x, hi=len(a) - 1) - 1
```

```
@jit(nopython=True, cache=True)
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(nopython=True)
def wanglandau(system,
               Es,             # The energy bins
               M = 1_00_000,   # Monte carlo step scale
               ε = 1e-10,      # f tolerance
               logf0 = 1,      # Initial log f
               logging = True, # Log progress of f-steps
               flatness = 0.1  # 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=np.int32)
    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| = ", len(Es),
              "\n\tM = ", M,
              "\n\tε = ", ε,
              "\n\tlog f0 = ", 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: ", fiter, " / ", fiters, "\t(", iters, " / ", niters, ")")

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

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

<ipython-input-1-040fba0c4d8b> in <module>
----> 1 @jit(nopython=True)
      2 def wanglandau(system,
      3                 Es,             # The energy bins
      4                 M = 1_00_000,   # Monte carlo step scale
      5                 ε = 1e-10,      # f tolerance


NameError: name 'jit' is not defined
```

1.1.1 Parallel construction of the density of states

```
@jit(nopython=True)
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=np.int32)
#     systems = [None] * (len(Ebins) - 1)
    n = 0
    l = len(Ebins) - 1
    systems = [sys] * l
    empty = np.repeat(True, l)
    i = binindex(Es, sys.E)
    while np.any(empty) and n < N:
        for s in range(l):
            if empty[s] and Ebins[s] ≤ sys.E < Ebins[s + 1]:
                systems[s] = sys.copy()
                empty[s] = False
```

```
#     while np.any(np.array([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] = sys.copy()

        sys.propose()
        j = binindex(Es, sys.Ev)
        # Monotonic steps (not always applicable)
#         if sys.E < sys.Ev:
#             sys.accept()
        # Wang-Landau steps
        if S[j] < S[i]:
            i = j
            sys.accept()
        S[i] += 1
        n += 1

    if N ≤ n:
        raise ValueError('Could not find bin systems (hit step limit).')
    return systems
```

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))
```

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)
    states = [s.state() for s in systems]
    binEs = [(lambda E0, Ef: Es[(E0 ≤ Es) & (Es ≤ Ef)])(*extend_bin(Ebins, i, k))
             for i in range(len(Ebins) - 1)]
    return zip(states, 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.

```
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
```