0.1 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 njit
import numpy as np
```

```
import sys
if 'src' not in sys.path: sys.path.append('src')
import simulation as sim
```

Utility functions for the simulation.

```
@njit(inline='always')
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

@njit
def binindex(a, x):
    return bisect_right(a, x, lo=0, hi=len(a) - 1) - 1
```

```
@njit
def flat(H, ε = 0.2):
    """Determines if a histogram is approximately flat to within ε of the mean height."""
    return not np.any(H < (1 - ε) * np.mean(H)) and np.all(H ≠ 0)
```

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.

```
def system_prep(system):
    return system, system.energy_bins()
```

```
@njit
def simulation(system, Es,
               max_sweeps = 1_000_000,
               flat_sweeps = 1,
               eps = 1e-8,
               logf0 = 1,
```

```
                flatness = 0.2,
                log = False
               ):
    """
    Run a Wang-Landau simulation on system with energy bins Es to determine
    the system density of states g(E).

    Args:
        system: The system to perform the simulation on (see systems module).
        Es: The energy bins of the system to access. May be a subset of all bins.
        max_sweeps: The scale for the maximum number of MC sweeps per f-iteration.
            The actual maximum iterations may be fewer, but approaches max_sweeps
            exponentially as the algorithm executes.
        flat_sweeps: The number of sweeps between checks for histogram flatness.
            In AJP [10.1119/1.1707017], Landau et. al. use 10_000 sweeps.
        eps: The desired tolerance in f. Wang and Landau [WL] use 1e-8 in the original
            paper [10.1103/PhysRevLett.86.2050].
        logf0: The initial value of ln(f). WL set to 1.
        flatness: The desired flatness of the histogram. WL set to 0.2 (80% flatness).
        log: Whether or not to print results of each f-iteration.

    Returns:
        A tuple of results with entries:
        Es: The energy bins the algorithm was passed.
        S: The logarithm of the density of states (microcanonical entropy).
        H: The histogram from the last f-iteration.
        converged: True if each f-iteration took fewer than the maximum sweeps.

    Raises:
        ValueError: One of the parameters was invalid.
    """
    if (max_sweeps ≤ 0
        or flat_sweeps ≤ 0
        or eps ≤ 1e-16
        or not (0 < logf0 ≤ 1)
        or not (0 ≤ flatness < 1)):
        raise ValueError('Invalid Wang-Landau parameter.')

    # Initial values
    M = max_sweeps * system.sweep_steps
    flat_iters = flat_sweeps * system.sweep_steps
    logf = 2 * logf0 # Compensate for first loop iteration
    logftol = np.log(1 + eps)
    converged = True
    steps = 0

    E0 = Es[0]
    Ef = Es[-1]
    N = len(Es) - 1
    S = np.zeros(N) # Set all initial g's to 1
    H = np.zeros(N, dtype=np.int32)
    i = binindex(Es, system.E)

    if log:
        fiter = 0
        print("Wang-Landau START")
        print("fiter\t steps\t\t max steps")
```

```
        print("-----\t -----\t\t ---------")

    while logftol < logf:
        H[:] = 0
        logf /= 2
        iters = 0
        niters = int((M + 1) * np.exp(-logf / 2))
        if log:
            fiter += 1
        while (iters % flat_iters != 0 or not flat(H, flatness)) and iters < niters:
            system.propose()
            Ev = system.Ev
            j = binindex(Es, Ev)
            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
        steps += iters
        if niters ≤ iters:
            converged = False
        if log:
            print(fiter, "\t", iters, "\t", niters)

    if log:
        print("Done: ", steps, " total MC iterations;",
              "converged." if converged else "not converged.")
    return Es, S, H, steps, converged

def wrap_results(results):
    return {k: v for k, v in zip(('Es', 'S', 'H', 'steps', 'converged'), results)}
```

0.1.2 Parallel decomposition

```
@njit
def find_bin_systems(system, Es, Ebins, sweeps = 1_000_000, method = 'wl'):
    """
    Find systems with energies in the bins given by `Es` by stepping `sys`.

    Args:
        system: The initial system to search from. This is usually a ground state.
        Es: The energies of the system.
        Ebins: The energy bins to find systems for.
        sweeps: The maximum number of MC sweeps to try.
        method: The string name of the search method to try.
            'wl': Wang-Landau steps where we prefer energies we have not visited
            'increasing': Only accept increases in energy. This only works for
                steps that are not trapped by local maxima of energy.

    Returns:
        A list of independent systems with energies in Ebins.

    Raises:
        ValueError: The method argument was invalid.
        RuntimeError: Bin systems could not be found after N steps.
```

```
    """
    if method == 'wl':
        S = np.zeros(len(Es), dtype=np.int32)
    systems = [None] * (len(Ebins) - 1)
    n = 0
    N = sweeps * system.sweep_steps
    l = len(Ebins) - 1
    systems = [system] * l
    empty = np.repeat(True, l)
    i = binindex(Es, system.E)
    while np.any(empty) and n < N:
        for s in range(l):
            if empty[s] and Ebins[s] ≤ system.E < Ebins[s + 1]:
                systems[s] = system.copy()
                empty[s] = False

        system.propose()
        j = binindex(Es, system.Ev)
        if method == 'wl':
            if S[j] < S[i]:
                i = j
                system.accept()
            S[i] += 1
        elif method == 'increasing':
            if system.E < system.Ev:
                system.accept()
        else:
            raise ValueError('Invalid method argument for finding bin systems.')
        n += 1

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

```
def psystem_prep(system, bins = 8, overlap = 0.1, sweeps = 1_000_000, method = 'wl', **kwargs):
    Es = system.energy_bins() # Intrinsic to the system
    Ebins = np.linspace(Es[0], Es[-1], bins + 1) # For parallel subsystems
    systems = find_bin_systems(system, Es, Ebins, sweeps, method)
    binEs = [(lambda E0, Ef: Es[(E0 ≤ Es) & (Es ≤ Ef)])(*sim.extend_bin(Ebins, i, overlap))
             for i in range(len(Ebins) - 1)]
    return zip(systems, binEs)
```

```
def run(params, **kwargs):
    return sim.run(params, simulation, system_prep, psystem_prep, wrap_results, **kwargs)
```

```
def join_results(results, *args, **kwargs):
    return sim.join_results(*zip(*[(r['Es'][:-1], r['S']) for r in results]), *args, **kwargs)
```