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
from numba import njit
import numpy as np
```

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

Utility functions for the simulation.

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

```python
@njit(cache=True)
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)
```

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.

```
1   @njit(cache=True)
2   def simulation(system,
3                   Es,             # The energy bins
4                   M = 1_00_000,   # Monte carlo step scale
5                   ε = 1e-10,      # f tolerance
6                   logf0 = 1,      # Initial log f
7                   flatness = 0.1, # Desired histogram flatness
8                   logging = False # Log progress of f-steps
9                   ):
10      if M ≤ 0 or ε ≤ 1e-16 or not (0 < logf0 ≤ 1) or not (0 ≤ flatness < 1):
11          raise ValueError('Invalid Wang-Landau parameter.')
12
13      # Initial values
14      E0 = Es[0]
15      Ef = Es[-1]
16      N = len(Es) - 1
17      logf = 2 * logf0
18      logftol = np.log(1 + ε)
19      S = np.zeros(N) # Set all initial g's to 1
20      H = np.zeros(N, dtype=np.int32)
21      i = binindex(Es, system.E)
22      converged = True
23
24      if logging:
25          mciters = 0
26          fiter = 0
27          fiters = int(np.ceil(np.log2(logf0) - np.log2(logftol)))
28          print("Wang-Landau START")
29
30      while logftol < logf:
31          H[:] = 0
32          logf /= 2
33          iters = 0
34          niters = int((M + 1) * np.exp(-logf / 2))
35          if logging:
36              fiter += 1
37          while not flat(H, flatness) and iters < niters:
38              system.propose()
39              Ev = system.Ev
40              j = binindex(Es, Ev)
41              if E0 ≤ Ev < Ef and (
42                  S[j] < S[i] or np.random.rand() < np.exp(S[i] - S[j])):
43                  system.accept()
44                  i = j
45              H[i] += 1
```

```
46              S[i] += logf
47              iters += 1
48          mciters += iters
49          if niters ≤ iters:
50              converged = False
51          if logging:
52              print("f: ", fiter, " / ", fiters, "\t(", iters, " / ", niters, ")")
53
54      if logging:
55          print("Done: ", mciters, " total MC iterations.")
56      return Es, S, H
```

1.1.1   Parallel construction of the density of states

```
1   @njit
2   def find_bin_systems(system, Es, Ebins, N = 1_000_000, method = 'wl'):
3       """
4       Find systems with energies in the bins given by `Es` by stepping `sys`.
5
6       Args:
7           system: The initial system to search from. This is usually a ground state.
8           Es: The energies of the system.
9           Ebins: The energy bins to find systems for.
10          N: The maximum number of steps to try.
11          method: The string name of the search method to try.
12              'wl': Wang-Landau steps where we prefer energies we have not visited
13              'increasing': Only accept increases in energy. This only works for
14                  steps that are not trapped by local maxima of energy.
15
16      Returns:
17          A list of independent systems with energies in Ebins.
18
19      Raises:
20          ValueError: The method argument was invalid.
21          RuntimeError: Bin systems could not be found after N steps.
22      """
23      if method == 'wl':
24          S = np.zeros(len(Es), dtype=np.int32)
25      systems = [None] * (len(Ebins) - 1)
26      n = 0
27      l = len(Ebins) - 1
28      systems = [system] * l
29      empty = np.repeat(True, l)
30      i = binindex(Es, system.E)
31      while np.any(empty) and n < N:
```

```
32          for s in range(l):
33              if empty[s] and Ebins[s] ≤ system.E < Ebins[s + 1]:
34                  systems[s] = system.copy()
35                  empty[s] = False
36
37          system.propose()
38          j = binindex(Es, system.Ev)
39          if method == 'wl':
40              if S[j] < S[i]:
41                  i = j
42                  system.accept()
43              S[i] += 1
44          elif method == 'increasing':
45              if system.E < system.Ev:
46                  system.accept()
47          else:
48              raise ValueError('Invalid method argument for finding bin systems.')
49          n += 1
50
51      if N ≤ n:
52          raise RuntimeError('Could not find bin systems (hit step limit).')
53      return systems


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