0.1 Organized parallel simulations

We need some utility functions that

- Manage passing simulation parameters to parallel processes
- Run the parallel processes with correct input and distinct random states
- Save parameters and results to files automatically (without overwriting previous results)
- Log results while running simulations
- Join parallel results back together

Since the simulation systems (Numba `jitclass` objects) are not picklable (Python's serialization used to communicate between processes by `multiprocessing`), the relevant parameters to reconstruct a system are passed between processes. We require that all systems be accessible from the `systems` module, so that we may pass and save the class name as a way of accessing the specification of the system.

```
import numpy as np
from multiprocessing import Pool
from scipy.signal import windows
import sys
import time
import os, struct # for `urandom`
import pprint # for parameters
import tempfile
import h5py, hickle
```

```
if 'src' not in sys.path: sys.path.append('src')
import systems
```

```
def make_params(system):
    return {system.__class__.__name__:
            {k: v for k, v in zip(system.state_names(), system.state())}}

def make_system(system_params, system_prep = lambda x:x):
    return system_prep([getattr(systems, cls)(**state)
                        for cls, state in system_params.items()][0])

def make_psystems(params, psystem_prep): #:: params -> (system -> [system]) -> [params]
    log = params.get('log', False)
    if log:
        print('Finding parallel bin systems ... ', end='', flush=True)
    psystems = psystem_prep(make_system(params['system']), **params['parallel'])
    if log:
        print('done.')
    return [(make_params(s), *r) for s, *r in psystems]

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

def worker(simulation, psystem, params):
    log = params.get('log', False)
```

```
    urandom_reseed()
    psystem_params, *args = psystem
    system = make_system(psystem_params)
    if log:
        print('(', end='', flush=True)
    # Individual simulation output is too much when running parallel simulations.
    params['simulation'].update({'log': False})
    results = simulation(system, *args, **params['simulation'])
    if log:
        print(')', end='', flush=True)
    return results

def show_params(params):
    print('Run parameters')
    print('--------------')
    pprint.pp(params, sort_dicts=False)
    print()

def save_results(results, params, log=False, prefix='simulation-', dir='data'):
    with tempfile.NamedTemporaryFile( # Note: dir shadows dir()
        mode='wb', prefix=prefix, suffix='.h5', dir=dir, delete=False) as f:
        with h5py.File(f, 'w') as hkl:
            if log:
                print('Writing results ... ', end='', flush=True)
            hickle.dump({
                'parameters': params,
                'results': results
            }, hkl)
            relpath = os.path.relpath(f.name)
            if log:
                print('done: ', relpath)
    return relpath

def run(params, simulation, system_prep,
        psystem_prep = lambda x:x, result_wrapper = lambda x:x, **kwargs):
    params.update(kwargs)
    parallel = 'parallel' in params
    log = params.get('log', False)
    if log:
        show_params(params)

    if parallel:
        psystems = make_psystems(params, psystem_prep)
    else:
        psystem = make_system(params['system'], system_prep)

    if log:
        if parallel:
            print('Running || ', end='', flush=True)
        else:
            print('Running ...')
        start_time = time.time()

    if parallel:
        with Pool() as pool:
            results = pool.starmap(worker, ((simulation, args, params) for args in psystems))
        results = [result_wrapper(r) for r in results]
```

```
    else:
        results = result_wrapper(simulation(*psystem, **params['simulation'], **kwargs))

    if log:
        seconds = int(time.time() - start_time)
        if parallel:
            print(' || done in', seconds, 'seconds.')
        else:
            print('... done in', seconds, 'seconds.')

    # Save single-shot results in a singleton list so that we can analyze parallel and
    # single results the same way.
    rdict = {'results': results if parallel else [results]}
    save_params = params.pop('save', False)
    if save_params:
        relpath = save_results(results, params, log, **save_params)
        rdict.update({'file': relpath})

    return rdict
```

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))
```

Often parallel results are the value of a real function on some grid or list of bins. Given that many of these pieces may overlap, we must combine them back together into a full solution. This requires first transforming the results so that they are comparable, and then performing the combination. The most common case is repetition of the same real-valued experiment. No transformation is required, and we simply average all the results. Even better, we may assign the values within each piece a varying credence from 0 to 1 and perform weighted sums.

We must join results that are only known up to an additive constant. We must then assign an offset to each solution, where one of the offsets may be taken to be zero without loss of generality. The algorithm below implements the offsets that minimize the weighted mean-squared error of the overlap regions

$$\varepsilon = \sum_{i<j,\,k} ((y_{ik} + c_i) - (y_{jk} + c_j))^2 w_{ik} w_{jk},$$

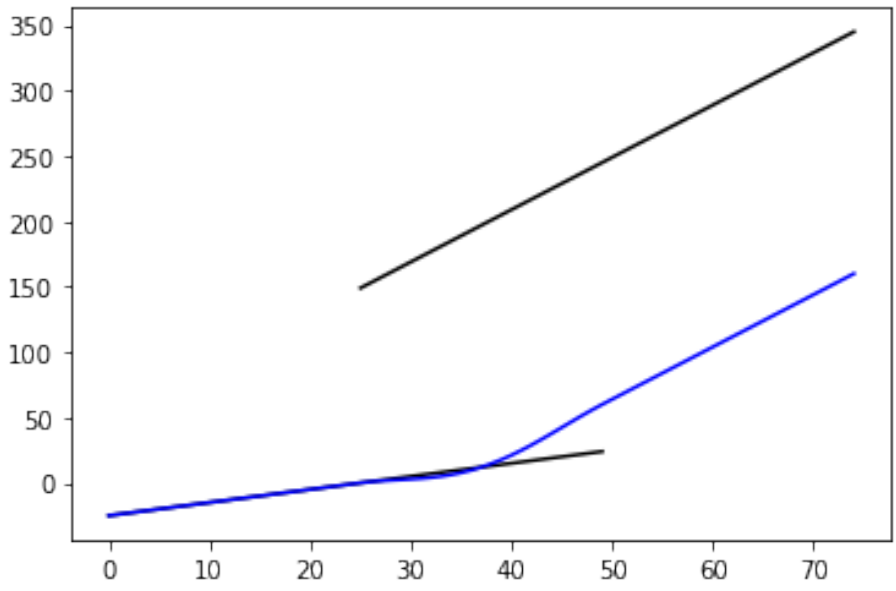
where i and j index different (overlapping) results, k indexes the full grid, and w_{ik} is the nonnegative weight assigned to index k of result i. Since we usually have results defined on a subset of the whole grid, we assign weights to the subset and set the other weights to be zero, so that the missing values y_{ik} outside the known subset are irrelevant.

```
def join_results(xs, ys, wf = windows.hann):
    xf = np.array(sorted(set().union(*xs)))
```

```
    xi = [np.intersect1d(xf, x, return_indices=True)[1] for x in xs]

    n, m = len(xf), len(xs)
    ws = np.zeros((m, n))
    wc = np.zeros((m, n))
    for i in range(m):
        l = len(xs[i])
        ws[i, xi[i]] = wf(l)
        wc[i, xi[i]] = np.ones(l)
    unweighted = np.sum(wc, 0) ≤ 1

    Δys = np.zeros(m)
    for i in range(m):
        Σc = Σw = 0
        for j in range(i):
            a = Δys[j] * np.ones(n)
            a[xi[j]] += ys[j]
            a[xi[i]] -= ys[i]
            w = ws[i,:] * ws[j,:]
            Σc += np.dot(a, w)
            Σw += np.sum(w)
        Δys[i] = Σc / Σw if i > 0 else 0

    yf = np.zeros(n)
    for i in range(m):
        w = ws[i, xi[i]]
        # The weights are meaningful only as relative weights at overlap points.
        # We must avoid division by zero at no-overlap points with weight zero.
        # Note that overlap points with all weights zero will be an issue, as
        # the weights in that situation are meaningless.
        w[(w == 0) & unweighted[xi[i]]] = 1
        yf[xi[i]] += (ys[i] + Δys[i]) * w
        ws[i, xi[i]] = w
    Σws = np.sum(ws, 0)
    yf /= Σws

    return xf, yf, Δys
```

Demonstration of joining overlapping results.

```
if __name__ == '__main__':
    from matplotlib import pyplot as plt

    testn = 50
    xs = [np.arange(testn), np.arange(testn // 2, testn + testn // 2)]
    ys = [np.arange(testn) - testn // 2, 4*np.arange(testn // 2, testn + testn // 2) + (testn - 1)]
    axs, ays, _ = join_results(xs, ys)

    for x, y in zip(xs, ys):
        plt.plot(x, y, 'black')
    plt.plot(axs, ays, 'blue');
```


350
300
250
200
150
100
50
0
0
10
20
30
40
50
60
70