1 Organized parallel simulations

```python
import numpy as np
from multiprocessing import Pool
from scipy.signal import windows
from functools import partial
import sys
import time
import os, struct # for `urandom`
import pprint
```

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

```python
def params_to_system(system_params):
    return [getattr(systems, cls)(**state)
            for cls, state in system_params.items()][0]

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

```python
def make_psystems(psystem_func, params): #:: params -> (system -> [system]) -> [params]
    logging = params['parallel']['logging']
    if logging:
        print('Finding parallel bin systems ... ', end='', flush=True)
    psystems = psystem_func(params_to_system(params['system']), **params['parallel'])
    if logging:
        print('done.')
    return [(system_to_params(s), *r) for s, *r in psystems]
```

```python
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):
        logging = params['parallel']['logging']
        urandom_reseed()
        psystem_params, *args = psystem
        system = params_to_system(psystem_params)
        if logging:
            print('(', end='', flush=True)
        results = simulation(system, *args, **params['simulation'])
        if logging:
            print(')', end='', flush=True)
        return results

def run_parallel(simulation, arguments, params):
    logging = params['parallel']['logging']
    if logging:
        print('Running | ', end='', flush=True)
        start_time = time.time()
    with Pool() as pool:
        results = pool.starmap(worker, ((simulation, args, params) for args in arguments))
    if logging:
        print(' | done in', int(time.time() - start_time), 'seconds.')
    return results
```

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.

```
def join_results(results):
    x0, y0 = results[0]
    x, y = x0, y0
    for i in range(1, len(results)):
        xv, yv = results[i]
        # Assumes overlap is at end regions
        _, i0s, ivs = np.intersect1d(x0[:-1], xv[:-1], return_indices=True)
        # Simplest: join middles of overlap regions
        l = len(i0s)
        m = l // 2
        yv -= yv[ivs[m]] - y0[i0s[m]]
        # Simplest: average the overlaps to produce the final value
        x = np.hstack((x, xv[l+1:]))
        y[-l:] = (yv[ivs] + y0[i0s]) / 2
        y = np.hstack((y, yv[l:]))
        x0, y0 = xv, yv
    return x, y
```

```
def align_results(xs, ys, wf = partial(windows.tukey, alpha=0.1)):
    xf = sorted(set().union(*xs))
    xi = [np.intersect1d(xf, x, assume_unique=True, return_indices=True)[1] for x in xs]

    # TODO: Implement offset of different pieces on top of one another.
    raise NotImplementedError()
```

```
def sum_results(n, results, weights):
    yf = np.zeros(n)
    wf = np.zeros(n)
    for (x, y), w in zip(results, weights):
        yf[x] += w * y
        wf[x] += w
    return yf / wf
```