Noisy park bench

April 15, 2019

1 About this notebook

The Python 3 code in this Jupyter notebook reproduces all simulation results, and two of the associated plots, for the paper "Minimal descriptions of cyclic memories" by Joseph D. Paulsen and Nathan C. Keim.

In [1]:
```
import sys
import itertools, collections

import numpy as np
import scipy.stats
import matplotlib
from matplotlib import pyplot as pl

import numba  # Not required but makes a VERY big difference in speed.
```

In [2]:
```
# Show versions of these dependencies
print('Python', sys.version)
print('numpy', np.__version__)
print('scipy', scipy.__version__)
print('matplotlib', matplotlib.__version__)
print('numba', numba.__version__)
```

```
Python 3.5.6 |Anaconda custom (x86_64)| (default, Aug 26 2018, 16:30:03)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)]
numpy 1.14.2
scipy 0.17.0
matplotlib 1.5.1
numba 0.24.0
```

2 Functions to drive and read the system

In [3]:
```
# If you don't have numba you can just comment out lines like the next one.
# The simulation will just run MUCH slower.
@numba.jit
def cut_grass(values, ampl):
    """Cut grass in one cycle of driving.
    Note that 'values' is modified by this operation."""
    for i in range(ampl):
        if values[i] > 0:
            values[i] -= 1
    return values
```

```
In [4]: def read_mem_values(values, h_init, threshold=1):
            """Identify memories in a sequence of grass heights.

            'threshold' sets the step size in the grass height required
            for detection.
            """
            values = np.array([h_init] + list(values) + [h_init,])
            return np.nonzero(np.diff(values) >= threshold)[0]
```

3 Detailed examination of $N = 3, h_{\text{init}} = 10$

```
In [5]: N = 3
        h_init = 10

        # The starting configuration is always with the grass at full height
        state0 = np.asarray((h_init,) * N)
```

3.1 Test with no noise

20 cycles with a 1-2 pattern → only largest value retained.

```
In [6]: state = state0.copy()
        for ampl, cyc in zip(itertools.cycle([2, 1]), range(20)):
            state = cut_grass(state, ampl)

        state

Out[6]: array([ 0,  0, 10])
```

3.2 Add noise

Shorter grass grows faster (on average), linearly.

For grass at site i with height h_i and max. height h_{init}, the probability of increasing grass height by 1 is

$$\alpha \frac{h_{\text{init}} - h_i}{h_{\text{init}}}$$

where α is given by the `alpha` argument:

```
In [7]: @numba.jit(nopython=True)
        def add_noise_linear(values, alpha, h_init):
            probs = alpha * (h_init - values) / h_init
            for i in range(len(values)):
                if random.random() < probs[i]:
                    values[i] += 1
            #values += (np.random.random(len(values)) < probs).astype(int)
            # Returns an ndarray instead of a list, but that's OK!
            return values  #.astype(int)

        @numba.jit(nopython=True)
        def init_random(seed=42):
            """Initialize the random number generator to a known state,
            for reproducibility."""
            np.random.seed(seed)
```

```
In [8]: @numba.jit
        def steady_states(state0, ampl_pattern, h_init,
                          ncycles=1e7, ncycles_transient=1e6, alpha=1.0):
            """Simulate many cycles with noise, and store history."""
            state = state0.copy()
            init_random()
            if h_init >= 65536:
                raise ValueError('h_init is too large to store states as 16-bit unsigned '
                                 'integers. Use a different dtype in steady_states().')
            history = np.zeros((int(ncycles), len(state)), dtype=uint16)

            training_sequence = itertools.cycle(ampl_pattern)  # Repeat indefinitely

            # Run the simulation for a transient period, but do not record the states.
            for cyc, ampl in zip(range(int(ncycles_transient)), training_sequence):
                state = add_noise_linear(state, alpha, h_init)
                state = cut_grass(state, ampl)

            # Store history in the steady state.
            # training_sequence will pick up where we left off at the end of the transient;
            # it is not reset.
            for cyc, ampl in zip(range(int(ncycles)), training_sequence):
                state = add_noise_linear(state, alpha, h_init)
                state = cut_grass(state, ampl)
                history[cyc,:] = state

            return history
```

If we are mowing the first site on every cycle, then $\alpha \sim \mathcal{O}(1)$ is appropriate.

To demonstrate this, we run the simulation with $\alpha = 1$, trying to store two memories in the steady state.

```
In [9]: history = steady_states(state0, [2, 1], h_init)
```

3.2.1 Steady-state probability of each state

```
In [10]: # Build a histogram of how much time we spend in each state.
         counts = collections.defaultdict(int)
         for state in history:
             counts[tuple(state)] += 1

In [11]: for st, count in sorted(counts.items()):
             print('%s\t%.6g' % (st, count / len(history)))
```

```
(0, 0, 10)        6.1e-06
(0, 1, 10)        0.0005366
(0, 2, 10)        0.0108742
(0, 3, 10)        0.0777627
(0, 4, 10)        0.23849
(0, 5, 10)        0.343886
(0, 6, 10)        0.238789
(0, 7, 10)        0.0781824
(0, 8, 10)        0.0109421
(0, 9, 10)        0.0005259
(0, 10, 10)        5.3e-06
```

3.2.2 Steady-state probabilities at the end of each training pattern

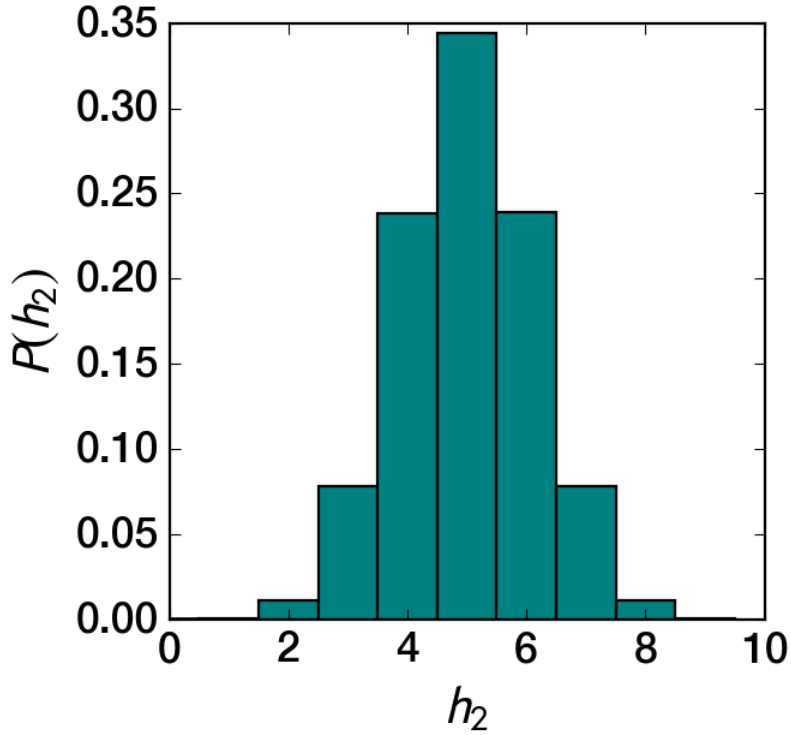
Compare these with the results of the Markov chain analysis in the paper.

```
In [12]: counts_wholepattern = collections.defaultdict(int)
         # The 2nd state (index 1) is after application of the 2nd training amplitude.
         for state in history[1::2, :]:
             counts_wholepattern[tuple(state)] += 1

         # States sorted by grass height
         for st, count in sorted(counts_wholepattern.items()):
             print('%s\t%.6g' % (st, count / (len(history) / 2)))
```

```
(0, 1, 10)        0.0001096
(0, 2, 10)        0.0043278
(0, 3, 10)        0.0465952
(0, 4, 10)        0.19065
(0, 5, 10)        0.343964
(0, 6, 10)        0.286431
(0, 7, 10)        0.109399
(0, 8, 10)        0.0175658
(0, 9, 10)        0.0009456
(0, 10, 10)        1.06e-05
```

3.2.3 Statistics of the 2nd site

```
In [13]: pl.figure(figsize=(4, 4))
         # Histogram
         with pl.style.context('presentation'):
             pl.hist(history[:,1], bins=np.arange(0, 11) - 0.5, color='teal',
                    normed=True);
             pl.xlim(0, 10)
             pl.xlabel('$h_2$')
             pl.ylabel('$P(h_2)$')
```




```
In [14]: # 1st and 2nd moments
         np.mean(history[:,1]), np.std(history[:,1])

Out[14]: (5.0012958, 1.1465991108065452)

In [15]: # 3rd and 4th moments
         scipy.stats.skew(history[:,1]), scipy.stats.kurtosis(history[:,1])

Out[15]: (-0.00026875437899970663, -0.09638389924079416)
```

4 Compute evolution for various h_{init} and $\langle D \rangle$

```
In [16]: histories = {}
         N = 3

         # Show dependence on h_init for D=0.5
         ampl_pattern = [2, 1]
         D = 0.5  # For filing results
         for h_init in (5, 10, 30, 100, 300, 1000):
             state0 = np.asarray((h_init,) * N)
             # Store only the middle site
             histories[(D, h_init)] = steady_states(state0, ampl_pattern, h_init)[:,1]

         ampl_pattern = [2, 1, 1]
         D = 1/3
         for h_init in [5, 10, 100, 1000]:
```

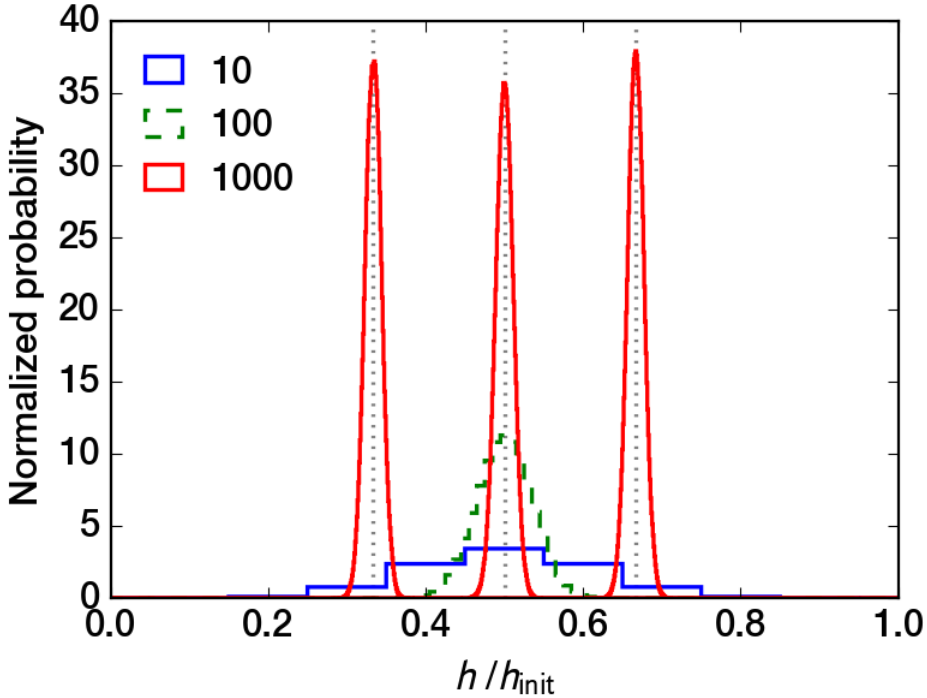
```
    state0 = np.asarray((h_init,) * N)
    histories[(D, h_init)] = steady_states(state0, ampl_pattern, h_init)[:,1]

ampl_pattern = [2, 2, 1]
D = 2/3
for h_init in [5, 10, 100, 1000]:
    state0 = np.asarray((h_init,) * N)
    histories[(D, h_init)] = steady_states(state0, ampl_pattern, h_init)[:,1]
```

4.1 Histograms for various conditions

In [17]:
```
# Verify that there is a bin centered on each possible grass height.
h_init = 10
np.linspace(0, (h_init + 1) / h_init, h_init + 2, endpoint=True) - 1 / h_init / 2
```

Out[17]:
```
array([-0.05,  0.05,  0.15,  0.25,  0.35,  0.45,  0.55,  0.65,  0.75,
        0.85,  0.95,  1.05])
```

In [18]:
```
pl.figure(figsize=(5, 3/4*5))
pl.axes((0.13, 0.15, 0.82, 0.8))

def grass_histogram(D, h_init, **kwargs):
    """Plot histogram of h/h_init with bins that correspond to integer grass heights."""
    pl.hist((histories[(D, h_init)] / h_init),
            bins=np.linspace(0, (h_init + 1) / h_init, h_init + 2,
                             endpoint=True) - 1 / h_init / 2,
            normed=True, histtype='step',
            **kwargs)

grass_histogram(0.5, 10, ls='-', lw=1.5, label=r'10')
grass_histogram(0.5, 100, ls='--', lw=1.5, label=r'100')
grass_histogram(0.5, 1000, ls='-', color='r', lw=1.5, label=r'1000')

grass_histogram(1/3, 1000, ls='-', lw=1.5, color='r')
grass_histogram(2/3, 1000, ls='-', lw=1.5, color='r')

axvline(0.5, color='0.5', ls=':', lw=1.5, zorder=0)
axvline(1/3, color='0.5', ls=':', lw=1.5, zorder=0)
axvline(2/3, color='0.5', ls=':', lw=1.5, zorder=0)

pl.xlim(0, 1)
pl.xlabel(r'$h\ / h_\mathrm{init}$', size=16)
pl.ylabel('Normalized probability')
pl.legend(frameon=False, handlelength=1, fontsize=14, loc='upper left')
pl.savefig('histograms_combined.pdf')
```



4.2 Standard deviation as a function of h_{init}

```
In [19]: for h_init in (5, 10, 30, 100, 300, 1000):
             print(h_init, '%.6g' % np.std(histories[(0.5, h_init)]))
```

```
5 0.832922
10 1.1466
30 1.95267
100 3.54318
300 6.13089
1000 11.157
```

4.3 Standard deviation as a function of $\langle D \rangle$ at $h_{\text{init}} = 1000$

```
In [20]: for D in (1/3, 1/2, 2/3):
             print('%.4g' % D, '%.6g' % np.std(histories[(D, 1000)]))
```

```
0.3333 10.55
0.5 11.157
0.6667 10.5456
```

5 Extended training patterns

5.1 $h_{\text{init}} = 1000$, 4-cycle pattern

```
In [21]: N = 3
         h_init = 1000
```

```
state0 = np.asarray((h_init,) * N)
history = steady_states(state0, [2, 2, 1, 1], h_init)
```

Statistics of the 2nd site

```
In [22]: # 1st and 2nd moments
         np.mean(history[:,1]), np.std(history[:,1])
```

```
Out[22]: (500.126768, 11.163344376761657)
```

```
In [23]: # 3rd and 4th moments
         scipy.stats.skew(history[:,1]), scipy.stats.kurtosis(history[:,1])
```

```
Out[23]: (-0.036416237461162515, 0.031853456843801364)
```

5.2 $h_{\text{init}} = 1000$, 100-cycle pattern

```
In [24]: N = 3
         h_init = 1000

         state0 = np.asarray((h_init,) * N)
         history = steady_states(state0, [2] * 50 + [1] * 50, h_init)
```

Statistics of the 2nd site

```
In [25]: # 1st and 2nd moments
         np.mean(history[:,1]), np.std(history[:,1])
```

```
Out[25]: (500.1238877, 13.296558341833745)
```

```
In [26]: # 3rd and 4th moments
         import scipy.stats
         scipy.stats.skew(history[:,1]), scipy.stats.kurtosis(history[:,1])
```

```
Out[26]: (-0.022061043920072774, -0.08792740311959246)
```

6 Simulations of a larger system with various α

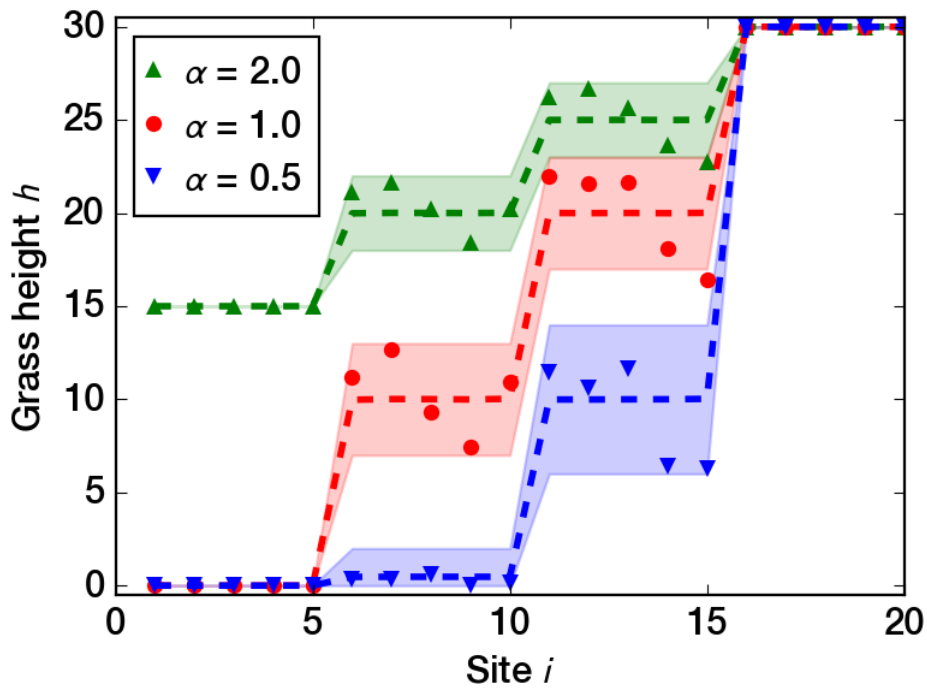
```
In [27]: N = 20
         h_init = 30

         state0 = np.asarray((h_init,) * N)

         pattern3 = [15, 10, 5]
```

Run for three different α

```
In [28]: history05 = steady_states(state0, pattern3, h_init, alpha=0.5, ncycles=1e6)
         history10 = steady_states(state0, pattern3, h_init, alpha=1.0, ncycles=1e6)
         history20 = steady_states(state0, pattern3, h_init, alpha=2.0, ncycles=1e6)
```

Gather and plot statistics about grass heights at each α, including the average over the last `n_sample` cycles.

```
In [29]: n_sample = 12

         means05 = np.mean(history05, axis=0)
         lower05 = np.percentile(history05, 5, axis=0)
         upper05 = np.percentile(history05, 95, axis=0)
         means_sample05 = np.mean(history05[-n_sample:], axis=0)

         means10 = np.mean(history10, axis=0)
         lower10 = np.percentile(history10, 5, axis=0)
         upper10 = np.percentile(history10, 95, axis=0)
         means_sample10 = np.mean(history10[-n_sample:], axis=0)

         means20 = np.mean(history20, axis=0)
         lower20 = np.percentile(history20, 5, axis=0)
         upper20 = np.percentile(history20, 95, axis=0)
         means_sample20 = np.mean(history20[-n_sample:], axis=0)

In [30]: sites = np.arange(len(state0)) + 1

         markersize = 6
         linewidth = 2.5

         pl.figure(figsize=(5, 3/4*5))
         pl.axes((0.13, 0.15, 0.82, 0.8))
         pl.fill_between(sites, lower20, upper20, color='g', alpha=0.2)
         pl.plot(sites, means_sample20, 'g^', label=r'$\alpha$ = 2.0', ms=markersize, mew=0)
         pl.plot(sites, means20, 'g--', linewidth=linewidth)

         pl.fill_between(sites, lower10, upper10, color='r', alpha=0.2)
         pl.plot(sites, means_sample10, 'ro', label=r'$\alpha$ = 1.0', ms=markersize, mew=0)
         pl.plot(sites, means10, 'r--', linewidth=linewidth)

         pl.fill_between(sites, lower05, upper05, color='b', alpha=0.2)
         pl.plot(sites, means_sample05, 'bv', label=r'$\alpha$ = 0.5', ms=markersize, mew=0)
         pl.plot(sites, means05, 'b--', linewidth=linewidth)

         pl.xlabel('Site $i$')
         pl.ylabel('Grass height $h$')
         pl.ylim(-0.5, h_init + 0.5)
         pl.legend(loc='upper left', numpoints=1, handlelength=0, borderpad=0.5,
                   prop={'size': 14})

         pl.savefig('noisy-steady-state.pdf')
```



6.1 Save histories (takes a few minutes)

```
In [31]: np.savetxt('N=20,h_init=30,alpha=0.5.gz', history05, fmt='%i')
         np.savetxt('N=20,h_init=30,alpha=1.0.gz', history10, fmt='%i')
         np.savetxt('N=20,h_init=30,alpha=2.0.gz', history20, fmt='%i')
```