Collective behaviour Indices

Some metrics aimed at quantifying bee group behaviour are implemented in the ‘cbtb‘ python package. This document describes these metrics and provides code examples.

1 (Group) Volatility index

In a collective binary choice assay:

Intuition: Quantify the number of “total population swings”, i.e., the number of times a group moves entirely from one decision to the other decision, per time unit. The metric is fractional, i.e., it can count partial group moves.

Computation Given n bees and the counts of bees on the left/right side at timestep t are given by $L(t), \in [0, n]$; $R(t), \in [0, n]$, s.t. $L(t) + R(t) = n, \quad \forall t$, we compute the volatility score $\upsilon(t)$ for one timestep t

$$\upsilon(t) = \frac{\|L(t) - L(t+1)\|}{n}, \tag{1}$$

and for a whole experiment / analysis window,

$$V = \frac{r}{t_{max}} \sum_{t=0}^{t_{max}-1} \upsilon(t), \tag{2}$$

where r is the rate of samples per analysis window, e.g., if the volatility per minute is desired, and the analysis timesteps are 1 s apart, set $r = 60$.

code

```
import numpy as np

def vol_step(X, t, n):
    '''
    compute the volatility score at one step t,
    among the data set X, which is assumed to be 1 dimensional info
    about a population of size n.
    '''
    _X = np.array(X) # X should be list-like but work on a numpy array
    if t >= _X.shape[0]-1:
        raise RuntimeError, "cannot compute index past end of array"

    v = np.abs(_X[t] - _X[t+1]) / float(n)
    return v

def volatility(X, n):
    '''
    compute volatility index for entire experimental series X
    '''
    _X = np.array(X)
    t_max = _X.shape[0]
    _V = np.zeros(t_max-1)

    for t in xrange (t_max-1):
        _V[t] = vol_step(_X, t, n=n)
        #print _V[t], _X[t], _X[t+1]

    return (1.0 / (t_max-1.0) ) * _V.sum()

def _volatility_onefunc(X,n):
    ''' explicit form above; this does the same thing but harder to read! '''
    return np.abs(np.ediff1d(X)).sum() / float(n * len(X)-1)


# define some examples
n = 12

# oscillating signal -- maximal
Losc1 = [n if i % 2 ==0 else 0 for i in xrange(18)]
# oscillating more slowly
Losc2 = [n if i % 4 < 2 else 0 for i in xrange(50)]

# Smaller oscillations, every 2
Losc3 = [n if i % 4 < 2 else n/2 for i in xrange(20)]

# a single transition, either slowly...
L1ramp = n * np.ones(50,)
L1ramp[0:n+1] = range(0,n+1)
# or quickly...
L1step = n * np.ones(50,)
L1step[25:] = 0

# two transitions
```

```
L2 = np.zeros(50,)
L2[0:20] = n
L2[20:25] = 0
L2[25:] = n

# constant readings
Cn = n * np.ones(50,)
Cn2 = n/2 * np.ones(50,)
C0 = 0  * np.ones(50,)

print "======\nMany changes in population decision give high metric values:"
print "{:.4f}\t".format(volatility(Losc1, n)), "n 0 n 0 ...    : Full swing every step"
print "{:.4f}\t".format(volatility(Losc2, n)), "n n 0 0 n n ...: Full swing every other step
print "{:.4f}\t".format(volatility(Losc3, n)), "n n n/2 n/2 n n ... "
print "\n======\nThe form of change is not relevant, only the magnitude:"
print "{:.4f}\t".format(volatility(L1step, n)), "One 100%% step change in 50 steps"
print "{:.4f}\t".format(volatility(L1ramp, n)), "One 100%% ramp change over 50 steps"
print "{:.4f}\t".format(volatility(L2, n)), "Two full transitions in 50 steps"

print "\n=======\nConstant situations should measure zeros:"
print "{:.4f}\t".format(volatility(Cn, n)), "Constant =n"
print "{:.4f}\t".format(volatility(C0, n)), "Constant =0"
print "{:.4f}\t".format(volatility(Cn2, n)), "Constant =n/2"
```

2 Collective decision-making index

Intuition: This metric is based on a time budget of how much time a group spends in a significantly unlikely aggregation. For a binary decision in an unbiased environment, we define the null model as the binomial distribution, such that a large number of the group need to have made the same decision (shown through their position) for the configuration to be considered as "significantly unlikely".

The binomial distribution tells us what would happen if all the actors moved randomly, and independently from one another.

For a confidence level of $p = 0.05$, we find that among groups of $n = 12$, aggregations of $a \geq 10$ actors on the same side are considered significant.

Computation Given n bees and the counts of bees on the left/right side at timestep t are given by $L(t), \in [0, n]$; $R(t), \in [0, n]$, s.t. $L(t) + R(t) = n, \quad \forall t$, and a threshold for significant decision d that is derived from the binomial distribution.

We can compute the majority side $M(t)$ for each timestep as

$$M(t) = \max\left(L(t), R(t)\right). \tag{3}$$

Then we simply produce a binary value c, for each timestep indicating whether the population has significantly decided or not,

$$c(t) = \begin{cases} 1 \text{ if } M(t) \geq d, \\ 0 \text{ otherwise.} \end{cases} \tag{4}$$

Finally, we produce a mean value of these per-step decisions such to give a fractional time budget, in the range $[0, 1]$,

$$C = \frac{1}{t_{max}} \sum_{t=0}^{t_{max}} c(t). \tag{5}$$

Code It should be noted that the implementation below uses an upper and a lower threshold in order to compute the value for $c(t)$ directly from L rather than first computing M. I have no idea which is computationally more efficient.

```
import numpy as np
from scipy import stats

def compute_threshold(n, onesided=True, pval=0.05, prob=0.5):
    '''
    for binomial test, at what point is it considered a significant
    cllective decision?
    '''
    # compute in a bit of a clunky way - compute all values until we find
    # the first level that exceeds the desired p value.
    ev = np.zeros((n+1,))
    for i, L in enumerate(xrange(n+1)):
        if onesided:
            p = stats.binom.sf(L-1, n, p=prob)
        else:
            raise NotImplementedError
        if p < pval:
            ev[i] = True

    nz = ev.nonzero()[0]
    if len(nz):
        return nz[0]
    else:
        return -1

def cdi(X, n):
    '''
    for a 1d data series X, presenting the number of a total of n animals
    on one side of a binary collective choice assay:
    compute the collective decision index

    '''
    # compute upper and lower thresholds.
    thr_upper = compute_threshold(n)
    thr_lower = n - thr_upper

    _X = np.array(X) # work on a numpy array for logic operators
    coll_decns = ((_X >= thr_upper) | (_X <= thr_lower))

    return coll_decns.mean()



# === now consider some examples === #
n = 12


# oscillating signal -- maximal
Losc1 = [n if i % 2 ==0 else 0 for i in xrange(18)]
# oscillating more slowly
Losc2 = [n if i % 4 < 2 else 0 for i in xrange(50)]

# Smaller oscillations, every 2
Losc3 = [n if i % 4 < 2 else n/2 for i in xrange(20)]
```

```
# a single transition, either slowly...
L1ramp = n * np.ones(50,)
L1ramp[0:n+1] = range(0,n+1)
# or quickly...
L1step = n * np.ones(50,)
L1step[25:] = 0

# two transitions
L2 = np.zeros(50,)
L2[0:20] = n
L2[20:25] = 0
L2[25:] = n

# constant readings
Cn = n * np.ones(50,)
thr_u = compute_threshold(n)
Csub_th = np.ones(50,) * thr_u - 1
Cn2 = n/2 * np.ones(50,)
C0 = 0  * np.ones(50,)

print "\n=======\nextremes - strong decision either side; or even split"
print "{:.4f}\t".format(cdi(Cn, n)), "Constant =n. Stroing decision"
print "{:.4f}\t".format(cdi(C0, n)), "Constant =0"
print "{:.4f}\t".format(cdi(Cn2, n)), "Constant =n/2"
print "{:.4f}\t".format(cdi(Csub_th, n)), "Constant just under threshold"

print "======\nRapid oscillations don't reduce the index, so long as"
print "  the whole population moves together (not likely in reality with bees)"
print "{:.4f}\t".format(cdi(Losc1, n)), "n 0 n 0 ...    : Full swing every step"
print "{:.4f}\t".format(cdi(Losc2, n)), "n n 0 0 n n ...: Full swing every other step"
print "{:.4f}\t".format(cdi(Losc3, n)), "n n n/2 n/2 n n ... "

print "\n======\nIf the whole population changes, the decision index is high"
print "{:.4f}\t".format(cdi(L1step, n)), "One 100%% step change in 50 steps"
print "{:.4f}\t".format(cdi(L1ramp, n)), "One 100%% ramp change over 50 steps"
print "{:.4f}\t".format(cdi(L2, n)), "Two step transitions in 50 steps"


# fast transition to decision
Dfast = n/2 * np.ones(50,)
Dmid  = n/2 * np.ones(50,)
Dslow = n/2 * np.ones(50,)
Dfast[5:] = n
Dmid[25:] = n
Dslow[40:] = n
```

```
print "\n======\nSpeed of decision influences score:"
print "{:.4f}\t".format(cdi(Dfast, n)), "Decision onset at 5/50 steps"
print "{:.4f}\t".format(cdi(Dmid,  n)), "Decision onset at 25/50 steps"
print "{:.4f}\t".format(cdi(Dslow, n)), "Decision onset at 40/50 steps"

# random independent actions

print "\n======\nRandom independent behaviour gives very low scores (should be ~5%)"
for i in xrange(5):
    R1 = np.random.binomial(n, 0.5, 6000)
    print "{:.4f}\t".format(cdi(R1, n)), "6000 samples, p=0.5"
print "\n======\nBut a bias in that behaviour makes the index less useful."
for pv in [0.5, 0.6, 0.7, 0.8, 0.9, 0.95]:
    _c80 = np.zeros(20)
    for i in xrange(20):
        R1 = np.random.binomial(n, pv, 6000)
        _c80[i] = cdi(R1, n)
    print "{:.4f}\t".format(_c80.mean()), "mean cdi from 20 replicates, p={:.1f}".format(pv
```