Deliberation and the Wisdom of Crowds

Supplementary Material: Code for Simulations

For Journal Website

For each parameter constellation under consideration, our Monte Carlo simulation generates many independent outcomes of the pre- and post-deliberation opinion structure, and then derives estimates of relevant quantities by taking appropriate averages across simulation rounds. The simulations were run in Python 3. We first coded an explicit routine that represented the opinion structure in lists (not shown here). To validate and to improve speed, we took a different coding approach by vectorizing with numpy arrays. This speed-optimized, documented code is shown below.

```
import numpy as np
from itertools import combinations
rng = np.random.default_rng()

def share_absorb(rounds, n, S_size, sd, p_acc, p_shares, p_absorbs):
    """Monte Carlo simulation with rounds, group size, source size,
    standard deviation, and the access, shareing, and receiving parameters"""
    # list to track pre-deliberation majority
    pre_result_tracker = []
    # list to track post-deliberation majority
    post_result_tracker = []
    # list to track pre-deliberation spread imbalance
    si_tracker = []
    # list to track post-deliberation spread imbalance
    si_plus_tracker = []
    # list to track pre-deliberation interpersonal imbalance
    ii_tracker = []
    # list to track post-deliberation interpersonal imbalance
    ii_plus_tracker = []

    for r in range(rounds):
        # create boolean array with access S_i based on access parameter
        # Each [i,s] entry is whether individual i has access to source s
        access = p_acc - np.random.rand(n, S_size) > 0
        # calculate spread imbalance index
        si_sum = 0
        # for all spread pairs (combination), for each count spread on axis 0
        for tup in combinations(np.sum(access, axis=0), 2):
            #calculate spread imbalance
```

```
        s1 = abs(tup[0] - tup[1])
        s2 = (tup[0] + tup[1]) / 2
        #sum up if denominator is not 0 (otherwise assume result is 0)
        if s2 !=0 :
            si_sum += s1/s2
    si_sum = si_sum * 2 / (S_size * (S_size-1))
    si_tracker.append(si_sum)
    # create evidences based on normal distribution
    e_s = rng.normal(1, scale = sd, size = S_size)
    # limit evidences to each individual with access only
    # each [i,s] entry is the evidence individual i has from source s
    e_s_i = e_s * access
    # calculate interpersonal imbalance index pre-deliberation
    ii_sum = 0
    # for all individual evidence pairs (combination)
    # calculate absolute total evidence by summing an axis 1
    for tup in combinations(abs(np.sum(e_s_i, axis=1)), 2):
        #calculate interpersonal imbalance index
        s1 = abs(tup[0] - tup[1])
        s2 = (tup[0] + tup[1]) / 2
        #sum up if denominator is not 0
        if s2 !=0 :
            ii_sum += s1/s2
    ii_sum = ii_sum * 2 / (n * (n-1))
    ii_tracker.append(ii_sum)
    # votes as the sum of evidences each individual has access to
    pre_votes = np.sign(np.sum(e_s_i, axis=1)) # per individual
    pre_result = np.sign(np.sum(pre_votes)) # as group
    pre_result_tracker.append(pre_result)
    # Sharing
    # check who reaches random prob to share,
    # but only among those with access
    who_shares = access * (p_shares - np.random.rand(n, S_size) > 0)
    # determine which sources have been shared at least once
    sent = np.any(who_shares, axis=0)
    # determine who reaches random prob to absorb,
    # but only among the sources shared
    # join these new links with the existing links
    S_i_plus = sent * (p_absorbs - np.random.rand(n, S_size) > 0) + access
    # evidences for each individual with post-deliberation sources
    e_s_i_plus = e_s * S_i_plus
    # take post-deliberation votes
    post_votes = np.sign(np.sum(e_s_i_plus, axis=1))
    post_result = np.sign(np.sum(post_votes))
    post_result_tracker.append(post_result)
    # calculate post-deliberation spread imbalance index
    si_sum = 0
    for tup in combinations(np.sum(S_i_plus, axis=0), 2):
        s1 = abs(tup[0] - tup[1])
        s2 = (tup[0] + tup[1]) / 2
        if s2 !=0 :
            si_sum += s1/s2
    si_sum = si_sum * 2 / (S_size * (S_size-1))
    si_plus_tracker.append(si_sum)
```

```
    # calculate post-deliberation interpersonal imbalance index
    ii_sum = 0
    for tup in combinations(abs(np.sum(e_s_i_plus, axis=1)), 2):
        s1 = abs(tup[0] - tup[1])
        s2 = (tup[0] + tup[1]) / 2
        if s2 !=0 :
            ii_sum += s1/s2
    ii_sum = ii_sum * 2 / (n * (n-1))
    ii_plus_tracker.append(ii_sum)
# summarize aggregate estimates
predelib_comp = sum([x==1 for x in pre_result_tracker]) / rounds
postdelib_comp = sum([x==1 for x in post_result_tracker]) / rounds
si_mean = np.mean(si_tracker)
si_plus_mean = np.mean(si_plus_tracker)
ii_mean = np.mean(ii_tracker)
ii_plus_mean = np.mean(ii_plus_tracker)
comp_diff = postdelib_comp - predelib_comp
si_change = (si_plus_mean - si_mean) / si_mean
ii_change = (ii_plus_mean - ii_mean) / ii_mean
return (predelib_comp, postdelib_comp,si_mean, si_plus_mean, ii_mean,
        ii_plus_mean, comp_diff, si_change, ii_change)
```