COMP6026 - Assignment 2 - Group Selection

Henry Lovett - hl13g10

School of Electronics and Computer Science,
Faculty of Physics and Applied Sciences,
University of Southampton

6th January, 2014

1 Introduction

Group selection encounters many problems, one of which is that selfish behaviour is more commonly prefered (Powers et al., 2012). Selfish behaiour is overall detrimental to a group. The Prisoner's dilemma is an example of how selfish behaviour benefits an individual (Axelrod, 1987). However, in nature, cooperation is common (Szathmary and Maynard Smith, 1995). This then raises the question as to why cooperation exists.

In Powers et al. (2007), a situation was set up with selfish and cooperative individuals. Each individual also has a preference of being in a small or large group. Resources were allocated to the groups and the population increased depending on the amount of resource the genotype had. Selfish individuals had a higher growth, however had a higher consumption of the resource than the cooperative. A small group had less resource per captia than a large group.

In each generation, the pool was split into as many small and large groups as possible and allocated resources. The numbers of genotypes were then allowed to grow. This method has been shown by Wilson (1975) to purge selfish individuals. Powers et al. (2007) showed that the genotype of small cooperators flourished and became the only genotype in the population.

This paper discusses the reimplementation of the experiment Powers et al. (2007) and a comparison of results in sections 2 and 3. An extension to this work is covered in section 4, the results of which are shown in section 5 and section 6 concludes the paper. All code implemented is the writer's own and can be seen in Appendix A.

2 Reimplementation

The parameters used were taken from Powers et al. (2007) and can be seen in table 1. To implement the experiment, the following psuedocode was used:

- Initialise
- for number of generations:

- Make groups
- for timesteps:
  - Allocate resources
  - Grow populatiosn
- Reform migrant pool
- Scale migrant pool

This experiment used individuals with two genotypes, giving four possible combinations of individual. The genotypes included whether the individual preferred small or large groups, and whether it was selfish or cooperative. The four possible combinations therefore were: cooperative & small, selfish & small, cooperative & large, selfish & large.

In the reimplementation, an individual was not explicitly represented. Instead, a list of four values was used to store the total number of each genotype. This was used for both the migrant pool and the groups.

Initialisation was done exactly proportionately. Each genotype was assigned $\frac{N}{4}$ number of individuals. This was done so that no single genotype would gain an initial advantage from the beginning.

During each generation, the migrant pool was split down into as many small and large groups as possible. Individuals were split into groups depending on their preference in their genotype. Groups were made to represent the proportions of the global migrant pool. Only full groups were allowed and any members left over from group allocation were removed from the population. These groups were then allocated resources and allowed to grow.

Resources were allocated to each genotype proportionately depending on their genotype. This was done using equation (1). It is biased to allocate more resource to the selfish genotype ($[0.02 \times 0.2] > [0.018 \times 0.1]$). R also changes depending on the group size - small groups have limited resources to encourage cooperation, and large groups have more resource per captia, which encourages the selfish population.

$$r_i = \frac{n_i.G_i.C_i}{\sum_j (n_j.G_j.C_j)}.R \tag{1}$$

TABLE 1: Parameters used in the reimplementation

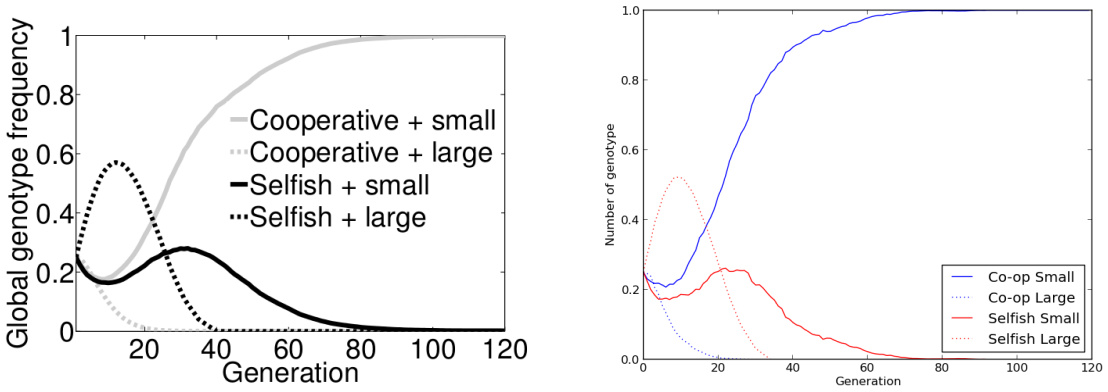
Parameter, *symbol*	value
Cooperative consumption rate, C_c	0.1
Selfish consumption rate, C_s	0.2
Cooperative growth rate, G_c	0.018
Selfish growth rate, G_s	0.02
Population size, N	4000
Small group size, N_{small}	4
Large group size, N_{large}	40
Number of generations, T	120
Number of timesteps, t	4
Resource for small groups, R_{small}	4
Resource for large groups, R_{large}	50
Death rate, K	0.1

Once the resources are allocated, the groups are then grown. The new population size is calculated by three terms, seen in equation (2). The first is the current size and the third is a constant death rate to all genotypes. The middle term uses the resources allocated and the consumption rate of the genotype. As the consumption for a cooperative genotype is lower, this is biased to allow the cooperators to grow more quickly.

$$n_i(t+1) = n_i(t) + \frac{r_i}{C_i} - K.n_i(t) \tag{2}$$

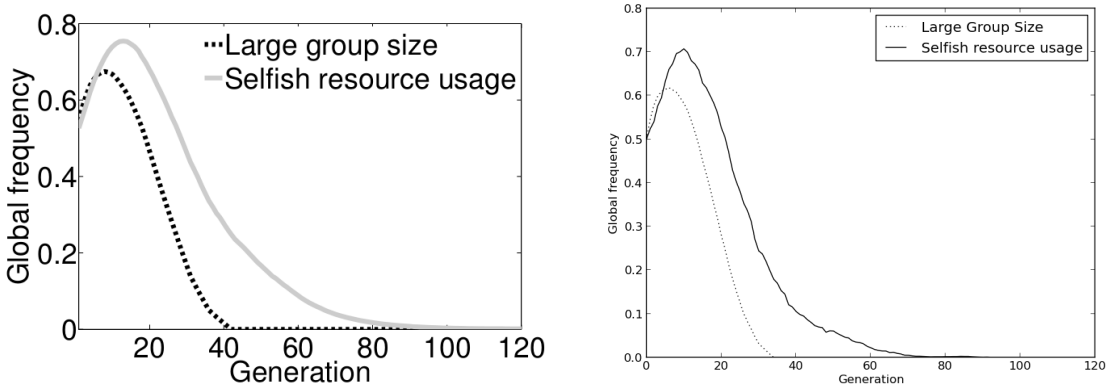
3 Comparison of results

The reproduced results were found to be very close to the original data. Figure 1 shows the proportions of each genotype in the population. In both graphs, the cooperators in the large group get immediately out competed by the selfish, and are then pushed to extinction. The numbers of large selfish then begin to diminish and both small genotypes increase before the cooperative small genotype excels



(a) The original results from Powers et al. (2007). (b) Reproduced results.

FIGURE 1: Change in genotype frequencies over time.



(a) The original results from Powers et al. (2007). (b) Reproduced results.

FIGURE 2: Average environment and strategy through time.

and results in being the entire of the population. The population reached a steady state by 100 generations.

Figure 2 shows the proportions of the strategies. Both results show that the large populations reach 0 first and the selfish gene takes a little longer to be removed from the population.

The results obtained from the extension proved to be a very close replication to the original data, and therefore can be used for an extension of this work.

4 Extension

Discrete groups do not always occur in nature. The extension covered here adds a third middle group to the experiment. The middle group contains all genotypes in similar proportions of the genotypes in the pool. As before, the individuals may only exist in one group during the group phase.

The main parameters remain unchanged (apart from the number of generations, which was increased to 200) from Powers et al. (2007). This experiment set out to find when, if at all, the small cooperators were out competed by another genotype. It is predicted that the large selfish genotype will take advantage of this middle group once a large enough proportion of the population is placed in this.

Some extra parameters were added to characterise the middle group. The size of this group, and the resources allocated, was made to be the average of the small and large group's size and resources. This was done to keep the same amount of resource per captia in the group. The final parameter was the parameter under test - the proportion of the population that was placed in the medium sized group. The parameters are summarised in table 2.

5 Results

A sweep of the middle proportions was done from 0 to 0.24. At each value of $M_{proportion}$, the simulation was run 10 times. After each simulation, the genotype with the largest population was deemed to be the 'winner' and a tally was kept. The number of wins of each genotype was plotted against the value of $M_{proportion}$ and can be seen in figure 3.

The results show that the small cooperators win consistently until around 0.03. From this point, the selfish large genotype starts to win some of the simulations. By 0.06, large selfish starts to win the majority of the simulations.

Both small groups win the occasional game in the higher proportions. This is assumed to be noise as no explicit checking was done to verify the populations had reached fixation.

The graphs in figure 4 show some of the populations in some of the simulations. The two graphs, 4(b) and 4(c) show two different outcomes for the same $M_{proportion}$ value.

TABLE 2: The extra parameters used to implement the extension

Parameter, *symbol*	value
Proportion of population in middle group, $M_{proportion}$	$0.0 - 0.24$
Medium group size, N_{med}	22
Medium group resource, R_{med}	27

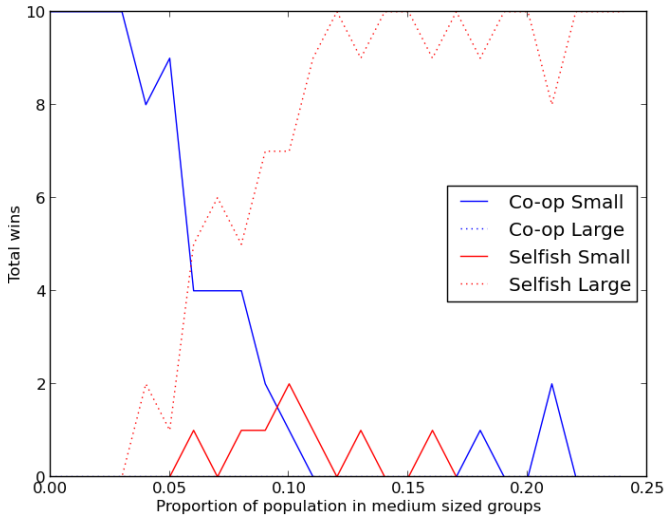
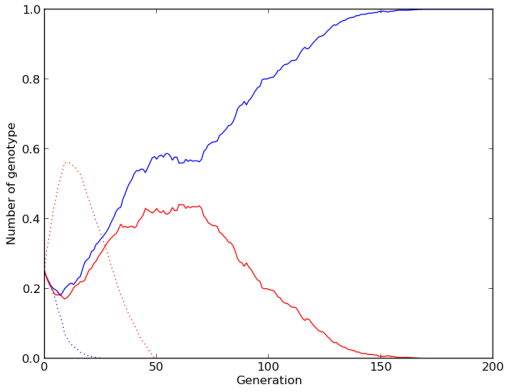
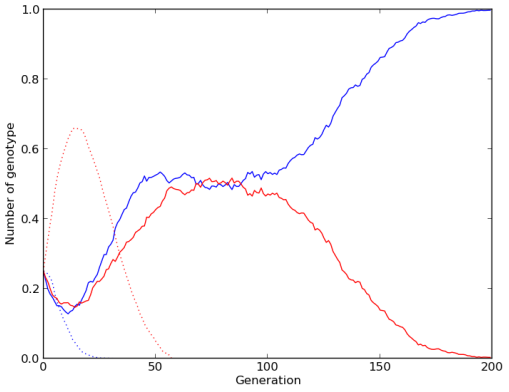


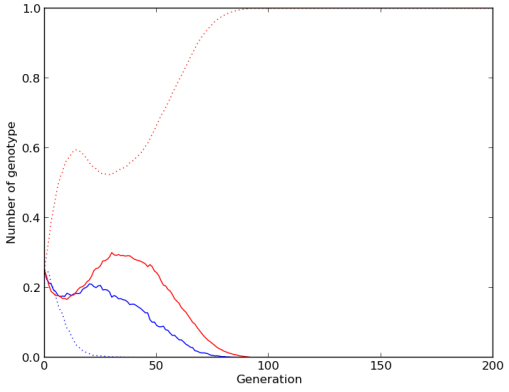
FIGURE 3: A sweep of the proportion of the population allocated into medium sized groups.



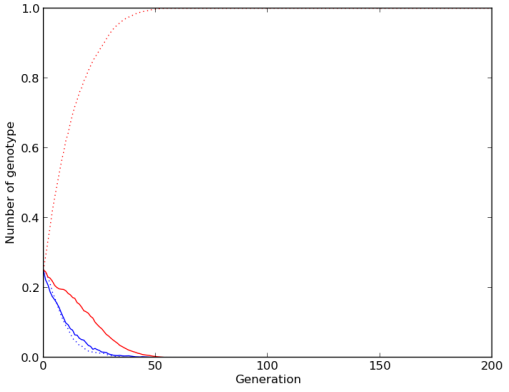
(a) Middle Proportion = 0.05



(b) Middle Proportion = 0.06 where the small cooperative wins.



(c) Middle Proportion = 0.06 where the large selfish wins.



(d) Middle Proportion = 0.24. The large selfish genotype wins outright.

FIGURE 4: Proportion of genotypes in simulations with varying proportions in the middle group.

6 Conclusion

This paper covered the reimplementation of the work in Powers et al. (2007). The implemented code was then shown to match the results expected and this then provided a platform to extend this work. The extension investigated set out look at creating more groups as a step towards making a fully continuous simulation. This was done by adding a middle group, for all genotypes being able to be a part of. A proportion of the pool was then allocated into these middle sized groups before allocating to the small and large sized groups.

It was predicted that the selfish gene would eventually take over the population as it would make use of being a defector in the middle sized group. This was found to be correct, as at about 0.06% of the population in the middle group, the selfish genotype won the majority of the simulations.

However, this work still creates discrete groups. Future work could improve this by having more groups, or even implementing groups of many different sizes. Also, in this experiment, the individuals are not treated differently depending on their group size preference in the middle group. More investigation could also be done into this.

References

Axelrod, R. (1987). The evolution of strategies in the iterated prisoner's dilemma. *The dynamics of norms*, pages 1–16.

Powers, S. T., Penn, A. S., and Watson, R. A. (2007). Individual selection for cooperative group formation. In *Advances in Artificial Life*, pages 585–594. Springer.

Powers, S. T., Penn, A. S., and Watson, R. A. (2012). The efficacy of group selection is increased by coexistence dynamics within groups. *arXiv preprint arXiv:1208.0518*.

Szathmary, E. and Maynard Smith, J. (1995). The major transitions in evolution. *Nature*, 374:227–232.

Wilson, D. S. (1975). A theory of group selection. *Proceedings of the national academy of sciences*, 72(1):143–146.

A Code

A.1 main.py

```
#!/usr/bin/python
# COMP6026
# @author hl13g10
# @brief reimplementation of Powers (2007) work for COMP6026 coursework
import math
import random
import pylab
outfile = "pool.txt"
fig = pylab.figure()
## Globals
K = 0.1 ## Global Death rate
R_small = 4.0 ## Resources for a small group
R_large = 50.0 ## Resources for a large group
Gc = 0.018 ## Growth rate for a cooperator
Gs = 0.02 ## Growth rate for selfish
Cc = 0.1 ## Consumption rate for a cooperator
Cs = 0.2 ## Consumption rate for selfsih
N = 4000 ## Population size
N_large = 40 ## Number of individuals in a large group
N_small = 4 ## Number of individuals in a small group
T = 120 ## Number of generations
t = 4 ## Number of time steps in groups

#The numbers will be stored in a list, these are the Indexs for each
    genotype
NUM_GENO = 4
COOP_SM = 0
COOP_LG = 1
SELF_SM = 2
SELF_LG = 3

#Some global lists for storing some data in to plot later.
data_c_s = list()
data_c_l = list()
data_s_s = list()
data_s_l = list()
large = list()
selfish = list()

## @brief Calculated the resources allocated to each genotype.
#  @param - the group to use
#  @retval - The resources allocated
#  Resources are allocated using the following formula
#  \f[
# r_i = \frac{ n_i . G_i . C_i }{\sum\limits_j (n_j . G_j . C_j )} . R
#  \f]
def Resource(group, R):

    #calculates the resource allocated to each genotype
    #calculate the sum part
    den = ( group[COOP_SM] * Gc * Cc ) + ( group[COOP_LG] * Gc * Cc ) + (
    group[SELF_SM] * Gs * Cs ) + ( group[SELF_LG] * Gs * Cs )
```

```
51      den = R / den  #and times it by the constant
52      resources = [0] * NUM_GENO
53      resources[COOP_SM] = den * group[COOP_SM] * Gc * Cc
54      resources[COOP_LG] = den * group[COOP_LG] * Gc * Cc
55      resources[SELF_SM] = den * group[SELF_SM] * Gs * Cs
56      resources[SELF_LG] = den * group[SELF_LG] * Gs * Cs
57      return resources
58
59
60  ## @brief Calculate the growth of the population depending on the resource
        calculation
61  #  @param group - the group to use
62  #  @param resource - the resources consumed by the group
63  #  @retval - The resulting population
64  #  growth is calculated using the following formula
65  #  \f[
66  #   n_i (t + 1) = n_i (t) + \frac{r_i}{C_i} - K.n_i (t)
67  #  \f]
68  def GrowPopulation(group, resource):
69      group[COOP_SM] = (group[COOP_SM] + ( resource[COOP_SM] / Cc ) - K *
        group[COOP_SM])
70      group[COOP_LG] = (group[COOP_LG] + ( resource[COOP_LG] / Cc ) - K *
        group[COOP_LG])
71      group[SELF_SM] = (group[SELF_SM] + ( resource[SELF_SM] / Cs ) - K *
        group[SELF_SM])
72      group[SELF_LG] = (group[SELF_LG] + ( resource[SELF_LG] / Cs ) - K *
        group[SELF_LG])
73      return group
74  ## @brief Inialises the global lists and clears the output file
75  def InitWrite():
76      f = open(outfile, 'w')
77      f.write("COOP_SM,COOP_LG,SELF_SM,SELF_LG\n")
78      f.close()
79      data_c_s = list()
80      data_c_l = list()
81      data_s_s = list()
82      data_s_l = list()
83      large = list()
84      selfish = list()
85
86  ## @brief Writes the pool data to a text file and stores to list for
        plotting
87  #  @param pool - the pool to write
88  def WriteData(pool):
89      f = open(outfile, 'a')
90      f.write("%d,%d,%d,%d\n" % (pool[COOP_SM], pool[COOP_LG], pool[SELF_SM
        ], pool[SELF_LG]))
91      f.close()
92      data_c_s.append(pool[COOP_SM] / float(N))
93      data_c_l.append(pool[COOP_LG] / float(N))
94      data_s_s.append(pool[SELF_SM] / float(N))
95      data_s_l.append(pool[SELF_LG] / float(N))
96      large.append((pool[SELF_LG] + pool[COOP_LG] )/ float(N))
97      selfish.append((pool[SELF_LG] + pool[SELF_SM] )/ float(N))
98      pass
99
100
```

```
101 ## @brief plots the data in the global lists.
102 def PlotAll():
103         pylab.figure(fig.number)
104         pylab.xlabel("Generation")
105         pylab.ylabel("Number of genotype")
106 
107     x = range(T)
108         pylab.plot(x, data_c_s, 'b-', label="Co-op Small") # '.' is point,
    ',' is pixel
109         pylab.plot(x, data_c_l, 'b:', label="Co-op Large") # '.' is point,
    ',' is pixel
110         pylab.plot(x, data_s_s, 'r-', label="Selfish Small") # '.' is
   point, ',' is pixel
111         pylab.plot(x, data_s_l, 'r:', label="Selfish Large") # '.' is
   point, ',' is pixel
112     pylab.legend(loc='lower right')
113     pylab.show()
114         pylab.draw()
115 
116     #pylab.figure()
117     pylab.xlabel("Generation")
118     pylab.ylabel("Global frequency")
119     pylab.plot(x, large, 'k:', label="Large Group Size")
120     pylab.plot(x, selfish, 'k-', label="Selfish resource usage")
121     pylab.legend(loc='upper right')
122     pylab.show()
123     pylab.draw()
124     pass
125 
126 ## @brief some testing to check things work
127 def Test():
128     test = [6.0,8.0,12.0,14.0]
129     r = Resource(test, R_large)
130     print ("Group :")
131     print test
132     print("Resources: ")
133     print r
134     GrowPopulation(test, r)
135     print ("Group :")
136     print test
137     raw_input()
138     pass
139 ## @brief main function.
140 #  Executes the stages of the GA.
141 if "__main__" == __name__:
142     #initialise an equally distributed population
143     InitWrite()
144     pool = list()
145     for i in range(NUM_GENO):
146         pool.append( float(N / NUM_GENO ) )
147     print pool
148     #WriteData(pool)
149     #r = Resource(pool)
150     #print r
151     #pool = GrowPopulation(pool, r)
152     #print pool
153     for g in range(T):
```

```
154         print("GENERATION %d" % g)
155         WriteData(pool)
156         #Group formation
157         smallgroups = list()
158         largegroups = list()
159         #number of groups
160         sm = int((pool[COOP_SM] + pool[SELF_SM]) / N_small)
161         lg = int((pool[COOP_LG] + pool[SELF_LG]) / N_large)
162         #calculate proportions
163         if sm:#if we have any small groups to make
164             p_sm_coop = pool[COOP_SM] / ( pool[COOP_SM] + pool[SELF_SM])
165             for i in range(sm):
166                 group = [0.0] * NUM_GENO
167                 for i in range(N_small): #group size of n small
168                     if (random.random() < p_sm_coop):#choose a coop
169                         group[COOP_SM] += 1
170                     else:
171                         group[SELF_SM] += 1
172                 smallgroups.append(group)
173         if lg:#if we have any large groups to make
174             p_lg_coop = pool[COOP_LG] / ( pool[COOP_LG] + pool[SELF_LG])
175             for i in range(lg):
176                 group = [0.0] * NUM_GENO
177                 for i in range(N_large): #group size of n small
178                     if (random.random() < p_lg_coop):#choose a coop
179                         group[COOP_LG] += 1
180                     else:
181                         group[SELF_LG] += 1
182                 largegroups.append(group)
183 
184         #Reproduction and resource allocation for allowed timesteps
185         for group in largegroups:
186             for _t in range(t):
187                 rl = Resource(group, R_large)
188                 GrowPopulation(group, rl)
189         for group in smallgroups:
190             for _t in range(t):
191                 rs = Resource(group, R_small)
192                 GrowPopulation(group, rs)
193         #Migrant pool formation
194         pool = [0.0]*NUM_GENO#reset pool - will remove any that didn't
    make it to groups
195         for group in (largegroups + smallgroups):
196             for i in range(NUM_GENO):
197                 pool[i] += group[i]
198         print("Pool Size = %d" % sum(pool))
199         #reduce pool
200         scale = float(N) / float(sum(pool)) #scale so that have a
    population size of N
201         print("Scale = %f" % scale)
202         for i in range(NUM_GENO):
203             pool[i] = ((pool[i] * scale))
204         print("Pool Size after scale = %d" % sum(pool))
205 
206     #end for T
207     PlotAll()
208     print("DONE!")
```

```
209     raw_input()
210     pass
```

Listing 1: Reimplementation of Powers et al. (2007)

A.2 main.py

```
1  #!/usr/bin/python
2  # COMP6026
3  # @author hl13g10
4  # @brief reimplementation of Powers (2007) work for COMP6026 coursework
5  import math
6  import random
7  import pylab
8  import time
9
10 outfile = "pool.txt"
11 fig = pylab.figure()
12 ## Globals
13 K = 0.1 ## Global Death rate
14 R_small = 4.0 ## Resources for a small group
15 R_large = 50.0 ## Resources for a large group
16 Gc = 0.018 ## Growth rate for a cooperator
17 Gs = 0.02 ## Growth rate for selfish
18 Cc = 0.1 ## Consumption rate for a cooperator
19 Cs = 0.2 ## Consumption rate for selfsih
20 N = 4000 ## Population size
21 N_large = 40 ## Number of individuals in a large group
22 N_small = 4 ## Number of individuals in a small group
23 T = 200 ## Number of generations
24 t = 4 ## Number of time steps in groups
25
26
27 #Extension parameters
28 M_Proportion = 0.05
29 N_med = 22
30 R_med = 27
31 #The numbers will be stored in a list, these are the Indexs for each
       genotype
32 NUM_GENO = 4
33 COOP_SM = 0
34 COOP_LG = 1
35 SELF_SM = 2
36 SELF_LG = 3
37
38 #Some global lists for storing some data in to plot later.
39 data_c_s = [0]*T
40 data_c_l = [0]*T
41 data_s_s = [0]*T
42 data_s_l = [0]*T
43 large =    [0]*T
44 selfish =  [0]*T
45
46 ## @brief Calculated the resources allocated to each genotype.
47 #  @param - the group to use
48 #  @retval - The resources allocated
49 #  Resources are allocated using the following formula
```

```
50 #  \f[
51 # r_i = \frac{ n_i . G_i . C_i }{\sum\limits_j (n_j . G_j . C_j )} . R
52 #  \f]
53 def Resource(group, R):
54 
55     #calculates the resource allocated to each genotype
56     #calculate the sum part
57     den = ( group[COOP_SM] * Gc * Cc ) + ( group[COOP_LG] * Gc * Cc ) + (
   group[SELF_SM] * Gs * Cs ) + ( group[SELF_LG] * Gs * Cs )
58     den = R / den  #and times it by the constant
59     resources = [0] * NUM_GENO
60     resources[COOP_SM] = den * group[COOP_SM] * Gc * Cc
61     resources[COOP_LG] = den * group[COOP_LG] * Gc * Cc
62     resources[SELF_SM] = den * group[SELF_SM] * Gs * Cs
63     resources[SELF_LG] = den * group[SELF_LG] * Gs * Cs
64     return resources
65 
66 
67 ## @brief Calculate the growth of the population depending on the resource
       calculation
68 #  @param group - the group to use
69 #  @param resource - the resources consumed by the group
70 #  @retval - The resulting population
71 #  growth is calculated using the following formula
72 #  \f[
73 #   n_i (t + 1) = n_i (t) + \frac{r_i}{C_i} - K.n_i (t)
74 #  \f]
75 def GrowPopulation(group, resource):
76     group[COOP_SM] = (group[COOP_SM] + ( resource[COOP_SM] / Cc ) - K *
   group[COOP_SM])
77     group[COOP_LG] = (group[COOP_LG] + ( resource[COOP_LG] / Cc ) - K *
   group[COOP_LG])
78     group[SELF_SM] = (group[SELF_SM] + ( resource[SELF_SM] / Cs ) - K *
   group[SELF_SM])
79     group[SELF_LG] = (group[SELF_LG] + ( resource[SELF_LG] / Cs ) - K *
   group[SELF_LG])
80     return group
81 ## @brief Inialises the global lists and clears the output file
82 def InitWrite():
83     f = open(outfile, 'w')
84     f.write("COOP_SM,COOP_LG,SELF_SM,SELF_LG\n")
85     f.close()
86     global data_c_s
87     global data_c_l
88     global data_s_s
89     global data_s_l
90     global large
91     global selfish
92     data_c_s = [0]*T
93     data_c_l = [0]*T
94     data_s_s = [0]*T
95     data_s_l = [0]*T
96     large =    [0]*T
97     selfish =  [0]*T
98 
99 ## @brief Writes the pool data to a text file and stores to list for
       plotting
```

```
100 #  @param pool - the pool to write
101 def WriteData(pool, g):
102     f = open(outfile, 'a')
103     f.write("%d,%d,%d,%d\n" % (pool[COOP_SM], pool[COOP_LG], pool[SELF_SM
        ], pool[SELF_LG]))
104     f.close()
105     #data_c_s.append(pool[COOP_SM] / float(N))
106     global data_c_s
107     global data_c_l
108     global data_s_s
109     global data_s_l
110     global large
111     global selfish
112     data_c_s[g] = (pool[COOP_SM] / float(N))
113     data_c_l[g] = (pool[COOP_LG] / float(N))
114     data_s_s[g] = (pool[SELF_SM] / float(N))
115     data_s_l[g] = (pool[SELF_LG] / float(N))
116     large[g] = ((pool[SELF_LG] + pool[COOP_LG] )/ float(N))
117     selfish[g] = ((pool[SELF_LG] + pool[SELF_SM] )/ float(N))
118     pass
119 
120 
121 ## @brief plots the data in the global lists.
122 def PlotAll():
123         pylab.figure(fig.number)
124         pylab.xlabel("Generation")
125         pylab.ylabel("Number of genotype")
126 
127     x = range(T)
128         pylab.plot(x, data_c_s, 'b-', label="Co-op Small") # '.' is point,
        ',' is pixel
129         pylab.plot(x, data_c_l, 'b:', label="Co-op Large") # '.' is point,
        ',' is pixel
130         pylab.plot(x, data_s_s, 'r-', label="Selfish Small") # '.' is
       point, ',' is pixel
131         pylab.plot(x, data_s_l, 'r:', label="Selfish Large") # '.' is
       point, ',' is pixel
132     #pylab.legend(loc='lower right')
133     pylab.show()
134         pylab.draw()
135 
136 ## @brief some testing to check things work
137 def Test():
138     test = [6.0,8.0,12.0,14.0]
139     r = Resource(test, R_large)
140     print ("Group :")
141     print test
142     print("Resources: ")
143     print r
144     GrowPopulation(test, r)
145     print ("Group :")
146     print test
147     raw_input()
148     pass
149 ## @brief runs the GA with the parameters set
150 def Run(ShowGraph = False):
151     #initialise an equally distributed population
```

```
152     InitWrite()
153     pool = list()
154     for i in range(NUM_GENO):
155         pool.append( float(N / NUM_GENO ) )
156     #print pool
157     for g in range(T):
158         #print("GENERATION %d" % g)
159         WriteData(pool, g)
160         #Group formation
161         smallgroups = list()
162         largegroups = list()
163         middlegroups = list()
164         #do middle group first and remove from pool
165         nummiddle = N * M_Proportion
166         #number of middle groups
167         md = int(nummiddle / N_med)
168         if md:
169             #calculate the proportions
170             p_CS = pool[COOP_SM] / ( sum(pool) )
171             p_CL = pool[COOP_LG] / ( sum(pool) )
172             p_SS = pool[SELF_SM] / ( sum(pool) )
173             p_SL = pool[SELF_LG] / ( sum(pool) )
174             for i in range(md):
175                 group = [0]*NUM_GENO
176                 for i in range(N_med):
177                     r = random.random()
178                     if r < p_CS:#choose Coop small
179                         group[COOP_SM] += 1
180                         pool[COOP_SM] -= 1
181                     elif r < (p_CL + p_CS):#choose coop large
182                         group[COOP_LG] += 1
183                         pool[COOP_LG] -= 1
184                     elif r < (p_SS + p_CL + p_CS):#choose selfish small
185                         group[SELF_SM] += 1
186                         pool[SELF_SM] -= 1
187                     else: #must be S.L.
188                         group[SELF_LG] += 1
189                         pool[SELF_LG] -= 1
190                 middlegroups.append(group)
191 
192         #number of groups
193         sm = int((pool[COOP_SM] + pool[SELF_SM]) / N_small)
194         lg = int((pool[COOP_LG] + pool[SELF_LG]) / N_large)
195         #calculate proportions
196         if sm:#if we have any small groups to make
197             p_sm_coop = pool[COOP_SM] / ( pool[COOP_SM] + pool[SELF_SM])
198             for i in range(sm):
199                 group = [0.0] * NUM_GENO
200                 for i in range(N_small): #group size of n small
201                     if (random.random() < p_sm_coop):#choose a coop
202                         group[COOP_SM] += 1
203                     else:
204                         group[SELF_SM] += 1
205                 smallgroups.append(group)
206         if lg:#if we have any large groups to make
207             p_lg_coop = pool[COOP_LG] / ( pool[COOP_LG] + pool[SELF_LG])
208             for i in range(lg):
```

```
209                 group = [0.0] * NUM_GENO
210                 for i in range(N_large): #group size of n small
211                     if (random.random() < p_lg_coop):#choose a coop
212                         group[COOP_LG] += 1
213                     else:
214                         group[SELF_LG] += 1
215                 largegroups.append(group)
216 
217         #Reproduction and resource allocation for allowed timesteps
218         for group in largegroups:
219             for _t in range(t):
220                 rl = Resource(group, R_large)
221                 GrowPopulation(group, rl)
222         for group in middlegroups:
223             for _t in range(t):
224                 rm = Resource(group, R_med)
225                 GrowPopulation(group, rm)
226         for group in smallgroups:
227             for _t in range(t):
228                 rs = Resource(group, R_small)
229                 GrowPopulation(group, rs)
230         #Migrant pool formation
231         pool = [0.0]*NUM_GENO#reset pool - will remove any that didn't
    make it to groups
232         for group in (largegroups +middlegroups + smallgroups):
233             for i in range(NUM_GENO):
234                 pool[i] += group[i]
235         #~print("Pool Size = %d" % sum(pool))
236         #reduce pool
237         scale = float(N) / float(sum(pool)) #scale so that have a
    population size of N
238         #print("Scale = %f" % scale)
239         for i in range(NUM_GENO):
240             pool[i] = ((pool[i] * scale))
241         #print("Pool Size after scale = %d" % sum(pool))
242 
243     #end for T
244     if ShowGraph:
245         PlotAll()
246     #print("DONE!")
247     winner = max(pool)#return the winner
248     for i in range(len(pool)):
249         if pool[i] == winner:
250             return i
251     raise Exception("No winner found...")
252 
253 if "__main__" == __name__:
254     #Run control experiment
255     print("Running Control Experiment...")
256     global M_Proportion
257     M_Proportion = 0
258     Run(ShowGraph=True)
259 
260     print("Running Extension Experiment...")
261     result_filename = "results_" +  time.strftime("%Y_%m_%d_%H%M") + ".txt
    "
262     print result_filename
```

```
263     res_file = open(result_filename, 'w')
264     res_file.write("M_Prop,CS,CL,SS,SL\n")
265     results = list()
266     proportions = range(25)
267     for i in range(len(proportions)):
268         proportions[i] /= 100.0
269     print proportions
270     #raw_input()
271     for M_Proportion in proportions:
272         winners = [0]*NUM_GENO
273         for i in range(10):
274             winners[Run()] += 1
275         print("PROP = %f" % M_Proportion)
276         print winners
277         results.append(winners)
278         res_file.write("%f,%d,%d,%d,%d\n" % (M_Proportion, winners[COOP_SM
    ], winners[COOP_LG], winners[SELF_SM], winners[SELF_LG]))
279     print results
280     res_file.close()
281 
282     #sort the data into a usable format
283     cs = list()
284     cl = list()
285     ss = list()
286     sl = list()
287     for group in results:
288         cs.append(group[COOP_SM])
289         cl.append(group[COOP_LG])
290         ss.append(group[SELF_SM])
291         sl.append(group[SELF_LG])
292 
293     pylab.figure(fig.number)
294     pylab.xlabel("Proportion of population in medium sized groups")
295     pylab.ylabel("Total wins")
296 
297     x = range(T)
298     pylab.plot(proportions, cs, 'b-', label="Co-op Small") # '.' is point,
     ',' is pixel
299     pylab.plot(proportions, cl, 'b:', label="Co-op Large") # '.' is point,
     ',' is pixel
300     pylab.plot(proportions, ss, 'r-', label="Selfish Small") # '.' is
    point, ',' is pixel
301     pylab.plot(proportions, sl, 'r:', label="Selfish Large") # '.' is
    point, ',' is pixel
302     pylab.legend(loc='center right')
303     pylab.show()
304     pylab.draw()
305 
306     pass
```

Listing 2: Extension code