Completing the implementation of the genetic algorithm

In this worksheet, we are going to complete the implementation of the genetic algorithm. We shall implement a population model and selection as well as genetic manipulation. We have provided you with the starter code, which is the code as it was at the end of the previous lab.

This lab is quite long, so if you get stuck or you just want to skip to experimenting with the GA, you can use the complete code provided after the lab.

Objectives

- Implement roulette-wheel selection
- Implement functions to mutate and crossover genomes
- Create an iterated genetic algorithm that evolves a population of creatures

Selecting parents using Roulette-wheel selection

Roulette-wheel selection allows us to select creatures who will be the parents of the next generation. The algorithm takes account of the fitness of the creatures such that fitter creatures are more likely to become parents.

We will create two functions on the Population class: get_fitness_map and select_parent. get_fitness_map takes a list of fitness values (floating-point values) as input and provides at its output a kind of weighted distribution. select_parent samples from the weighted distribution.

Here is a test for get_fitness_map, to go into test_population.py:

```
def testFitmap(self):
    fits = [2.5, 1.2, 3.4]
    want = [2.5, 3.7, 7.1]
    fitmap = population.Population.get_fitness_map(fits)
    self.assertEqual(fitmap, want)
```

Can you implement get_fitness_map to pass this test? Hopefully, it is clear that it essentially sums the values in the input array and writes the sum at each step into another array.

Here are some tests for the select_parent function, to go into test_population.py

```
## check for a parent id in the range 0-2
def testSelPar(self):
    fits = [2.5, 1.2, 3.4]
    fitmap = population.Population.get_fitness_map(fits)
    pid = population.Population.select_parent(fitmap)
    self.assertLess(pid, 3)
```

```
## parent id should be 1 as the first fitness is zero
## second is 1000 and third is 0.1 , so second should
## almost always be selected
def testSelPar2(self):
    fits = [0, 1000, 0.1]
    fitmap = population.Population.get_fitness_map(fits)
    pid = population.Population.select_parent(fitmap)
    self.assertEqual(pid, 1)
```

Note that both the functions can be static methods, as they do not depend on the state of the Population object:

```
@staticmethod
def get_fitness_map(fits):
```

So now we have a roulette wheel!

Start on the GA: evaluate a population and select parents

Let's combine the roulette wheel with the simulation code from previous worksheets. Here is some starter code for that. Put this in a file called test_ga.py

```
import unittest
import population
import simulation

class TestGA(unittest.TestCase):
    def testBasicGA(self):
        pop = population.Population(pop_size=10,
                                    gene_count=3)
        sim = simulation.ThreadedSim(pool_size=4)
        sim.eval_population(pop, 2400)
        fits = [cr.get_distance_travelled()
                for cr in pop.creatures]
        self.assertNotEqual(fits[0], 0)

unittest.main()
```

Now to carry out selection - after calculating fits, add this code to select the parents.

```
...
fits = [cr.get_distance_travelled()
        for cr in pop.creatures]
fit_map = population.Population.get_fitness_map(fits)
new_creatures = []
for i in range(len(pop.creatures)):
    p1_ind = population.Population.select_parent(fit_map)
    p2_ind = population.Population.select_parent(fit_map)
```

```
    p1 = pop.creatures[p1_ind]
    p2 = pop.creatures[p2_ind]
    # now we have the parents!
    self.assertIsNotNone(p1)
    self.assertIsNotNone(p2)
```

Crossover

The next step is to combine the genomes of the parents using a crossover function.

We will write the crossover function into the Genome class. Here is a simple crossover algorithm. The algorithm is slightly different from the one in the videos, but it has more or less the same effect:

```
we have two genomes g_1 and g_2 of length n_1 and n_2
generate value x_1 between 0 and n_1 - 1 inclusive
generate value x_2 between 0 and n_2 - 1 inclusive
concatenate g1[x_1:end] to g2[x2:end]
```

Try and implement this function inside the genome class as a static method:

```
@staticmethod
def crossover(g1, g2):
```

Here is some example code to get you started.

```
import random
import numpy as np
g1 = [[1], [2], [3]]
g2 = [[4], [5], [6]]
x1 = random.randint(0, len(g1))
x2 = random.randint(0, len(g2))
g3 = np.concatenate((g1[x1:], g2[x2:]))
```

Does that code ever crash? Why? Carefully check if it implements the algorithm above. Can you stop it from crashing?

Here is a test that you can put in test_genome.py

```
def testCrossover(self):
    g1 = [[1], [2], [3]]
    g2 = [[4], [5], [6]]
    for i in range(10):
        g3 = genome.Genome.crossover(g1, g2)
        self.assertGreater(len(g3), 0)
```

Another issue with this crossover function is that it favours the increase of genome size over time as it adds part of parent 1 to part of parent 2. You might want to limit the size of the genome - this is simple but effective:

```
if len(g3) > len(g1):
```

```
    g3 = g3[0:len(g1)]
```

At this point, we should have a working crossover function that does not crash on repeated calls. As usual, refer to the example code following the lab for a solution.

Mutation

Now that we can combine parent genomes via crossover, we need to mutate the genomes to introduce further variation.

Point mutation

Point mutation involves adjusting individual values in the genome. Here is an algorithm for that:

```
genome is an array of arrays of floats
mutation rate is a value in the range 0-1
mutation amount is a value in the range 0-1 (or less)
Iterate over genes in the genome:
   Iterate over values in each gene:
      if random < mutation_rate:
         generate random value in range
         -mutation_amount to mutation amount
         add that value to the value at the current position
         in current gene
```

Here is some starter code (for genome.py):

```
import copy # put this line at the top!
@staticmethod
def point_mutate(genome, rate, amount):
    new_genome = copy.copy(genome)
    return new_genome
```

Here are two tests (you could write more!) for test_genome.py:

```
def test_point(self):
    g1 = np.array([[1.0], [2.0], [3.0]])
    g2 = genome.Genome.point_mutate(g1,
                                    rate=1,
                                    amount=0.25)
    self.assertFalse(np.array_equal(g1, g2))

def test_point_range(self):
    g1 = np.array([[1.0], [0.0], [1.0], [0.0]])
    for i in range(100):
        g2 = genome.Genome.point_mutate(g1,
                                        rate=1,
```

```
                                        amount=0.25)
        self.assertLess(np.max(g2), 1.0)
        self.assertGreaterEqual(np.min(g2), 0.0)
```

Make sure you do not mutate in place - apply mutations to a copy of the incoming genome. Also - note that the second test verifies that the values have not gone out of the range 0-1.

There are some considerations for point mutation. Some parameters are in the range 0-1, but they specify only one of two values. The waveform for a motor controller is an example of this with its two possible waveforms. Other parameters are more continuous, such as the position and rotation parameters. Some researchers implement parameter specific types of mutation to allow appropriate mutations.

Shrink mutation

Now onto the shrink and grow mutations. Shrink is simple: remove a randomly chosen gene if a random value is lower than the rate.

We can do that with some NumPy magic:

```
np.delete([1,2,3,4], 1, 0)
```

Here are some tests for you to pass, put them in test_genome.py

```
def test_shrink(self):
    g1 = np.array([[1.0], [2.0]])
    g2 = genome.Genome.shrink_mutate(g1, rate=1.0)
    # should def. shrink as rate = 1
    self.assertEqual(len(g2), 1)

def test_shrink2(self):
    g1 = np.array([[1.0], [2.0]])
    g2 = genome.Genome.shrink_mutate(g1, rate=0.0)
    # should not shrink as rate = 0
    self.assertEqual(len(g2), 2)

def test_shrink3(self):
    g1 = np.array([[1.0]])
    g2 = genome.Genome.shrink_mutate(g1, rate=1.0)
    # should not shrink if already len 1
    self.assertEqual(len(g2), 1)
```

Note the constraints specified in those tests.

Grow mutation

Grow mutate involves adding a new gene to the end of the genome. Here are some tests for your grow_mutate function. They should go in test_genome.py:

```
def test_grow1(self):
    g1 = np.array([[1.0], [2.0]])
    g2 = genome.Genome.grow_mutate(g1, rate=1)
    self.assertGreater(len(g2), len(g1))

def test_grow2(self):
    g1 = np.array([[1.0], [2.0]])
    g2 = genome.Genome.grow_mutate(g1, rate=0)
    self.assertEqual(len(g2), len(g1))
```

This number should give you some help:

```
np.append([1,2,3], [4], axis=0)
```

Go ahead and implement grow_mutate on the Genome class:

```
@staticmethod
def grow_mutate(genome, rate):
```

Integrate crossover and mutation to make the next generation

Here is a reminder of the genetic algorithm so far:

```
pop = population.Population(pop_size=10,
                            gene_count=3)
sim = simulation.ThreadedSim(pool_size=4)

sim.eval_population(pop, 2400)
fits = [cr.get_distance_travelled()
        for cr in pop.creatures]
fit_map = population.Population.get_fitness_map(fits)
new_creatures = []
for i in range(len(pop.creatures)):
    p1_ind = population.Population.select_parent(fit_map)
    p2_ind = population.Population.select_parent(fit_map)
    p1 = pop.creatures[p1_ind]
    p2 = pop.creatures[p2_ind]
```

let's add some lines to generate a new genome from the parents and then to insert it into a new creature:

```
dna = genome.Genome.crossover(p1.dna, p2.dna)
dna = genome.Genome.point_mutate(dna, rate=0.25, amount=0.25)
dna = genome.Genome.shrink_mutate(dna, rate=0.25)
dna = genome.Genome.grow_mutate(dna, rate=0.25)
cr = creature.Creature(1)
cr.update_dna(dna)
```

```
new_creatures.append(cr)
```

We need an update_dna function on the Creature to update the links and motors based on the new DNA. It should also reset the creature. In Creature.py

```
def update_dna(self, dna):
    self.dna = dna
    self.flat_links = None
    self.exp_links = None
    self.motors = None
    self.start_position = None
    self.last_position = None
```

You might want to tidy this code into a 'reset' function on the creature.

Print out stats and iterate

Now we just need to iterate the algorithm:

```
pop = population.Population(pop_size=10,
                            gene_count=3)
sim = simulation.ThreadedSim(pool_size=4)
# put the rest in a for loop:
for iteration in range(10):
    sim.eval_population(pop, 2400)
    fits = [cr.get_distance_travelled()
            for cr in pop.creatures]
    fit_map = population.Population.get_fitness_map(fits)
    new_creatures = []
    for i in range(len(pop.creatures)):
        p1_ind = population.Population.select_parent(fit_map)
        p2_ind = population.Population.select_parent(fit_map)
        p1 = pop.creatures[p1_ind]
        p2 = pop.creatures[p2_ind]
        # now we have the parents!
        dna = genome.Genome.crossover(p1.dna, p2.dna)
        dna = genome.Genome.point_mutate(dna, rate=0.25, amount=0.25)
        dna = genome.Genome.shrink_mutate(dna, rate=0.25)
        dna = genome.Genome.grow_mutate(dna, rate=0.25)
        cr = creature.Creature(1)
        cr.update_dna(dna)
        new_creatures.append(cr)
    pop.creatures = new_creatures
```

Add some lines after the fits list comprehension to print out some stats:

```
fits = [cr.get_distance_travelled()
        for cr in pop.creatures]
links = [len(cr.get_expanded_links())
```

```
        for cr in pop.creatures]
print(iteration, "fittest:", np.round(np.max(fits), 3),
        "mean:", np.round(np.mean(fits), 3), "mean links", np.round(np.mean(links)), "max li
```

You should see output like this when you run the GA code:

```
0 fittest: 4.832 mean: 2.234 mean links 6.0 max links 13
1 fittest: 4.709 mean: 2.075 mean links 5.0 max links 13
2 fittest: 4.428 mean: 2.197 mean links 4.0 max links 19
3 fittest: 5.573 mean: 2.304 mean links 4.0 max links 13
4 fittest: 5.196 mean: 2.143 mean links 3.0 max links 14
5 fittest: 4.036 mean: 2.196 mean links 4.0 max links 17
```

Link counts give you an insight into how complex the creatures are.

Some tweaks

At this point, you are running the full GA. Again, check the code after this lab if yours does not appear to be working. I had to fine-tune a few aspects to get things working nicely. We will cover those tweaks in the following few sections.

Fixing crashing simulation

Sometimes, large, highly recursive creatures seem to crash pybullet when it tries to call getBasePositionAndOrientation. To fix that, you can wrap up the run_creature function in a try-except block:

```
def run_creature(self, cr, iterations=2400):
    try:
        # all the sim code here
    except:
        print("sim failed cr links: ",
              len(cr.get_expanded_links()))
```

Elitism

Elitism means you keep the fittest genome each generation. Elitism is easy to implement - you just use np.max(fits) to find the highest score, then find the creature with that score in the pop.creatures array and put that at position 0 in the new_creatures array after generating all the other new creatures. See if you can implement that. One essential thing to remember with elitism is not just to store the fittest creature object. You need to reset it first. You might implement a reset function on the creature or create a new creature and use the update_dna function to trigger a reset.

Some starting parameter values

We will experiment with the parameters more later, but for now, try the following settings:

- population size of 200
- try reducing the recursion parameter in the genome spec to 2 or 3
- try increasing the link-length genome parameter in genome.py to 2 instead of 1 to generate more 'spread out' creatures.

And then the cheaters

A common problem with genetic algorithms, and all machine learning algorithms, is cheating. The algorithm exploits inadequacies in your dataset or simulation environment. In creatures, this manifests as impossible flying creatures, creatures that grow really tall, so they fall over a long way from the origin and so on. You can increase the complexity of your fitness function to combat cheating. For example, do not allow creatures that go over a certain height off the ground from being parents.

Once you experiment with the full system, you will probably encounter some cheating! The best thing to do is to discuss this in the forums.

Saving fit creatures and viewing them in the pybullet GUI

At this point, you are probably quite keen to see the evolving creatures. This involves running them in real-time in GUI mode. To do that, we need to be able to save genomes to CSV and import them back in again.

Saving to CSV

Here are some tests for the to_csv function. Put them into test_genome.py.

```
def test_tocsv(self):
    g1 = [[1,2,3]]
    genome.Genome.to_csv(g1, 'test.csv')
    self.assertTrue(os.path.exists('test.csv'))

def test_tocsv_content(self):
    g1 = [[1,2,3]]
    genome.Genome.to_csv(g1, 'test.csv')
    expect = "1,2,3,\n"
    with open('test.csv') as f:
        csv_str = f.read()
    self.assertEqual(csv_str, expect)

def test_tocsv_content2(self):
    g1 = [[1,2,3], [4,5,6]]
```

```
    genome.Genome.to_csv(g1, 'test.csv')
    expect = "1,2,3,\n4,5,6,\n"
    with open('test.csv') as f:
        csv_str = f.read()
    self.assertEqual(csv_str, expect)
```

How did you get on?

Here is my to_csv function:

```
@staticmethod
def to_csv(dna, csv_file):
    csv_str = ""
    for gene in dna:
        for val in gene:
            csv_str = csv_str + str(val) + ","
        csv_str = csv_str + '\n'

    with open(csv_file, 'w') as f:
        f.write(csv_str)
```

Reading from CSV

Now to read the genome back in again. Here are some tests for that:

```
def test_from_csv(self):
    g1 = [[1,2,3]]
    genome.Genome.to_csv(g1, 'test.csv')
    g2 = genome.Genome.from_csv('test.csv')
    self.assertTrue(np.array_equal(g1, g2))

def test_from_csv2(self):
    g1 = [[1,2,3], [4,5,6]]
    genome.Genome.to_csv(g1, 'test.csv')
    g2 = genome.Genome.from_csv('test.csv')
    self.assertTrue(np.array_equal(g1, g2))
```

Have a crack at those. Here is an example solution:

```
@staticmethod
def from_csv(filename):
    csv_str = ''
    with open(filename) as f:
        csv_str = f.read()
    dna = []
    lines = csv_str.split('\n')
    for line in lines:
        vals = line.split(',')
        gene = [float(v) for v in vals if v != '']
```

```
        if len(gene) > 0:
            dna.append(gene)
    return dna
```

Add code to the GA to save fit genomes to CSV

You can insert a couple of lines to the genetic algorithm code in test_ga.py after the elitism line:

```
...
# elitism
max_fit = np.max(fits)
for cr in pop.creatures:
    if cr.get_distance_travelled() == max_fit:
        new_creatures[0] = cr
        # ---> here you can use to_csv:
        filename = "elite_"+str(iteration)+".csv"
        genome.Genome.to_csv(cr.dna, filename)
        #
        break
...
```

That will write the elite genome to disk with the number of the iteration in the filename.

Complete code to load from CSV and run in simulation

We now have the fit genomes written to disk. Let's load them into a real-time simulation and see how the creatures operate.

It would be neat to be able to quickly load in any CSV file and play it from the command line. This code will get you started with that. Create a file called playback_test.py and put this code into it:

```
import os
import genome
import sys

def main(csv_file):
    assert os.path.exists(csv_file), "Tried to load " + csv_file + " but it does not exists'
    dna = genome.Genome.from_csv(csv_file)
    cr = creature.Creature(gene_count=1)
    cr.update_dna(dna)
    ... now put your pybullet setup code here, along with
    ... the motor driving code from motor_test.py

if __name__ == "__main__":
    assert len(sys.argv) == 2, "Usage: python playback_test.py csv_filename"
```

```
main(sys.argv[1])
```

Now put in the code you have used before to render the creature to URDF, then to update the motors over time.

Mini challenge: about the genome spec

One problem with the CSV code is that it does not store the genome parameter settings with the CSV data. If you try and reload later with different recursion or geometry settings, the creatures will not be the same as those in the original evolution run. The answer is to save the genome spec to JSON and load it back in again when running the creatures. A challenge to you is to implement this code.

Experiment with parameters

You can adjust any of the parameters you like in the system, and we mentioned some earlier. We recommend starting with these:

- Population size
- Initial genome size
- Mutation rates and amounts for different types of mutation
- Recursion level in the genome
- Length of simulation run (2400 frames is 10 seconds)

See which settings give you a gradual increase in fitness. For example, if you have a high recursion parameter, e.g. 4, and you have large genomes (4+), you will see very complex creatures appearing which are hard to optimise. If you set the recursion lower (2) and start with a small number of genes (2), you might see a better search.

Share your settings in the forum with other students!

Challenge

There are other genome manipulations you could implement. Translation, where you move a gene to a different position, is one example. Another is duplication, where instead of growing by adding a random gene, you grow by making a copy of an existing gene and appending it. Try experimenting with different genomic manipulations.

Read up on genetic algorithm tournament selection - can you implement this as an alternative to roulette-wheel selection?

Review the objectives

Wow, that was a long lab. Congratulations on reaching the end. Here are the objectives - how did you do?

- Implement roulette-wheel selection
- Implement functions to mutate and crossover genomes
- Create an iterated genetic algorithm that evolves a population of creatures