# Evolutionary Robotics Exercise

## Motivation

From neural networks, to dynamical systems analysis to numerical integration and genetic algorithms, the *evolutionary robotics* methodology involves a wide variety of skill sets, each of which can be challenging to learn. Given the limited amount of time we have available, I have put together a (mostly) complete evolutionary robotics project, with the hope that it will allow you to “get your hands dirty” and learn from experience while investigating and experimenting with the software.

Below are instructions and questions to guide your investigation. It is my hope that you will not just stick to this guide, but that you will also follow your nose a bit, and look at parts of the code that you find interesting. As with the Robot Psychoanalysis Workshop, it is my hope that you focus on exploring, see yourself as a kind of detective, and enjoy the learning process.

## **Installation**

This exercise is based upon a few Python scripts. I have tested them in Linux and in Windows (Anaconda and ipython) and both worked without hassle. Your mileage may vary though, as this is all new code (this year!). Get in touch right away if you have installation issues and I’ll try to help. I recommend discussing technical issues (installation troubles etc.) on piazza so that insights are shared widely and quickly.

You may need to install a few packages (if they are not already installed) including **matplotlib**, **cython**, **numpy** and **scipy**. If you find you need to install any other packages, please let me know so that I can update these instructions accordingly. All of the necessary packages were already present on a new installation of Anaconda on Windows.

Artificial evolution is a computationally expensive process. To help speed things along I have used a basic form of **cython** acceleration, which compiles elements of the Python scripts to C. (Cython is why some of the files have the ending .pyx instead of .py). I have also used the **multiprocessing** package, which lets my program evaluate the fitness of several robots in parallel. Both significantly speed up the scripts on my machine, but may cause teething issues in getting it all to work on yours. Again, let me know and I’ll try to help debug any issues you have.

## Overview

This project is based upon the paper we have already read.

Seth, Anil K. (1998) *"Evolving action selection and selective attention without actions, attention, or selection."* From Animals to Animats – Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior. Vol. 5.

It also builds upon elements of the first six chapters of Valentino Braitenberg’s book *Vehicles* (which you read after the Robot Psychoanalysis workshop).

The code I have provided you with uses an evolutionary algorithm to train two-wheeled robots to seek out two different types of resource: food (green) and water (blue), while avoiding traps (red).

Each agent has two batteries, one that is replenished when the agent encounters food, and the other when it encounters water. The agent “dies” when it encounters a trap or when either of its batteries goes empty.

Each agent also has 6 sensors: 3 on each side, corresponding to a food-detector, a water-detector, and a trap-detector. For each sensor there is a single, (always ipsilateral) ‘link’ to a motor (this is simpler than Seth’s model). Each link maps a sensory state to a motor output. As in Seth’s paper, the form of this map is defined by a piecemeal-linear functions where the parameters (the control points for the piecemeal function) are determined by artificial evolution. The actual motor state is the sum of all incoming link outputs (as each motor has three incoming links). Also like in Seth’s paper, the piecemeal functions are scaled and translated in a genetically specified way by one of the two battery levels.

Given a robot with 6 sensors (food, water, and trap on the LEFT; food, water and trap on the RIGHT) then we have six links. But (!) but if we assume that the robots links have bilateral symmetry (i.e. that all of the links on the right are the same as on the left) then we only have to evolve 3 links.

**0.5 marks**

Draw here a quick sketch that shows how the sensors, motors and links relate to each other. It doesn’t need to be pretty, but it should be easy to read / interpret.

The scripts are...

* **robot.pyx** – Simulates a two wheeled robot with directional sensors. The sensors respond to the presence of ‘lights’ in the robot’s environment. Each light has a position in 2D space and a ‘type’ (e.g. ‘FOOD’). The type allows the robot to have different sensors that respond only to lights of a given type. For instance, the robot could have two sensors that only respond to the ‘FOOD’ lights; and an additional two sensors that only respond to the ‘WATER’ lights etc. The motors work similar to the robots we investigated in the last workshop – two independent motors can cause the robot to move forwards or backwards, pivot on the spot, move while turning, etc.
* **seth\_controller.pyx** – An implementation of the controller described in the paper *Evolving Action Selection and Selective Attention Without Actions, Attention, or Selection* by Anil Seth*.*
* **evolve.py** – Uses a genetic algorithm to tune the parameters of a SethController so that it keeps its two batteries (‘food’ and ‘water’) high, while avoiding traps.
* **animate\_best.py** – After having evolved a robot, this script can be run to watch an animation of the best (so far!) evolved agent in a random environment.

## **Part I: Develop an understanding of the robot simulation**

1. Open **robot.pyx** and briefly look through the code. This script can be run on its own to test that it is working, without getting into the complexities of the SethController. Note that it is not yet functional though! You need to first fix the code (as described below) to make it work.
2. Take what you have learned recently about Euler integration and add code into the labelled section of the **euler\_update()** method to make this robot simulation work properly. You will likely need to refer to the materials from Tuesday’s lecture.

All of your code for this answer should go between these two comments.   
  
 #### INSERT EULER INTEGRATION HERE (START)

#### INSERT EULER INTEGRATION HERE (END)

Note that you are already given the **calculate\_derivative()** method. This is always called before **euler\_update()** and it calculates the rate at which all of the robot’s state variables are changing and stores them in **self.dx**, **self.dy**, and **self.da**. Your code needs to use these values to update **self.x**, **self.y** and **self.a** appropriately according to the Euler integration method.

Recall that Euler integration assumes that the rate of change is constant for short periods of time. This short period of time is called the “time-step,” and is written ‘Δt’ implying a small, but finite change in time. In the **euler\_update()** method, this time step is specified by the argument **DT**, which has a default value of **0.02**. Make sure to use **DT** as part of your solution.

When you have done this correctly, and run this script (I run **`python3 robots.pyx`** from the command line to do so), you should see 10 robots, each going around in a circle. These robots are NOT being controlled by a SethController – but much more simply, their motor values are set to fixed values by the following lines near the bottom of **robot.pyx**.

## NOT PARTICULARLY INTERESTING ROBOT  
 robot.lm = 0.4   
 robot.rm = 0.5

**1 mark**

Copy / paste your Euler integration code answer here. It should probably be 3 lines long.

1. Time to make the robots more interesting. Find the comment

## BRAITENBERG AGGRESSION  
  
and add code after this comment that connects the sensors to the motors to make the “AGGRESSION Vehicle” that Braitenberg describes. This should be 2 lines of code. It should **not** involve any conditional statements (like IF...ELSE). Make sure that the “NOT PARTICULARLY INTERESTING” motor-setting instructions are overwritten or commented out so that you can test and observe how your AGGRESSION robots behave.

**0.5 marks**

Paste your AGGRESSION solution (code) here.

Also paste here a screen capture of your AGGRESSION vehicle’s behaviour as displayed when you run the script.

1. Q. Find the comment

## BRAITENBERG LOVE  
  
and after it add code to connect the sensors to the motors to make the “LOVE Vehicle” that Braitenberg describes. This should be 2 lines of code. It should **not** involve any conditional statements (like IF...ELSE). Again make sure your AGGRESSION controller code does not interfere with your LOVE controller code.

**0.5 marks**

Paste your LOVE solution here.

Also paste here a screen capture of your LOVE vehicle’s behaviour as displayed when you run the script.

1. Once you are confident you have the robots working, experiment with time-step. What are the advantages and disadvantages of low and high time-step sizes?

**0.5 marks**

Write your answer here.

## **Part II: Develop an understanding of seth\_controller.pyx**

Open **seth\_controller.pyx** and briefly look through the code. I don’t expect you to study and come to understand every detail of how this class works (though if you choose to use it in your group projects, then you’ll probably need to learn more than what I summarize here).

There are two classes in this file. **SMLink** describes a single ‘sensor→motor link’. It has two important methods.

* **set\_genome()** takes a list of numbers (each between 0 and 1) and uses those numbers to create the piece-wise function that specifies the output of the link for a given input
* **output()** takes a sensory state and the state of the batteries, and feeds them through the link’s piece-wise function to calculate its output

If you run this script (**`python3 seth\_controller.pyx`**) it plots a randomly generated SMLink.

To confirm that you understand how a genotype (a string of values) is turned into the phenotype (an actual SMLink that can transform sensory input into motor output), hand-design a genome that specifies the ipsilateral connection for a Braitenberg LOVE vehicle. All that matters is the overall shape of the green curve that is plotted—i.e. you can ignore the battery influence, and it is the overall shape of the curve that is important, not the specific values. To be able to do this, you will need to read the docstring (i.e. the long comment that explains the genotype of the SMLink) of the **set\_genome** method of **SMLink**. You should be able to make your answer different from everyone else’s.

**0.5 marks**

Write the genotype here (the list of values between 0 and 1), and paste the plot that is produced for your link.

The **SethController** is a collection of SMLinks. Glossing over the details, it also has a genome, which essentially is the concatenation of the genomes of all of its links. Its important methods are:

* **genome\_to\_links()** takes the genome and creates all of the links that it specifies
* **set\_sensor\_states()** called to let the controller know what the state is of its sensors
* **get\_motor\_output()** returns the motor output specified by the controller given the sensor states
* **procreate\_with()** this method is used to create an ‘offspring’ controller that is the descendent of two parent controllers.

**Part III: Develop an understanding of evolve.py**

Open **evolve.py** and take a look at it. There are several methods here. The most important is probably **simulate\_trial()** which is called by **evaluate\_fitness()**, which is called by **generation()**.

A **trial** is a single ‘run’, where one controller from the population is coupled to a robot that is placed in an environment where it interacts with food, water and traps. Each trial has a maximum duration (that can be cut short if either of the robot’s batteries goes flat), and for each trial the robot gets a **score** (which relates to how well it keeps its battery levels high without hitting traps).

**Fitness** for a controller is its average score over (N=10) trials. Each trial the controller and the robot’s initial position is exactly the same, but the placement of entities in its environment is different.

The evolutionary algorithm works by

1. Evaluating the fitness for every individual in the population.
2. Creating a new generation of individuals (the offspring of the old generation)
   1. The new generation always includes a copy of the best performing individual from the previous generation… (this is called “elitism”)
   2. ...plus many new ‘offspring’ individuals. Each offspring has two parents from the previous generation. The parents are selected with a likelihood that is proportional to their fitness, such that individuals with greater fitness are more likely to parent any individual offspring. The genes of the offspring are a random combination of both parents that is then ‘mutated’ i.e. changed by a small random amount. To see how this works, check out the **procreate\_with()** method of the **SethController** class.
3. Replace the old generation with the new generation and repeat!

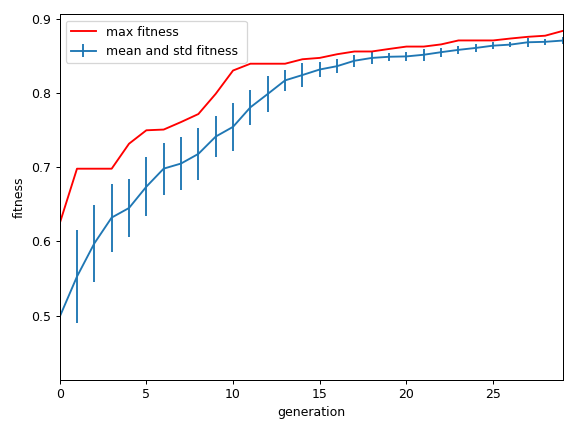
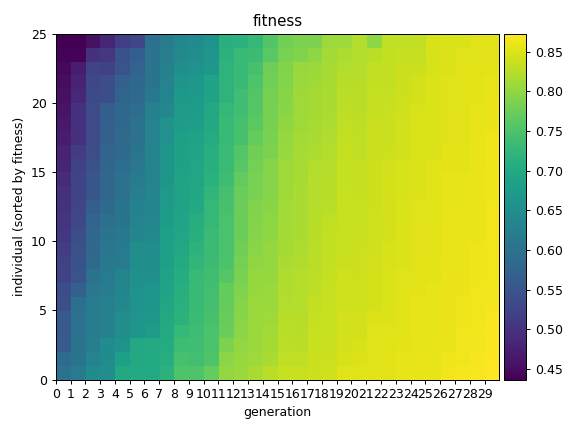
Why did I make fitness be the average of multiple trials rather than just a single trial?

**0.5 marks**

Find the ‘flag’ variable ‘**TEST\_GA**’ at the top of **evolve.py** and set it to TRUE. This disables all of the simulation and radically simplifies the evaluation of fitness to just be the simple summation of all of the genes.

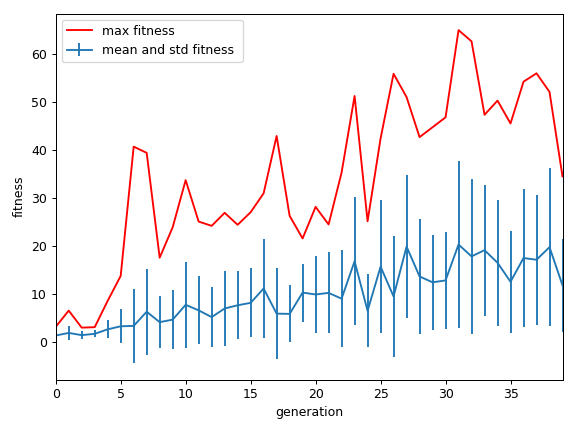
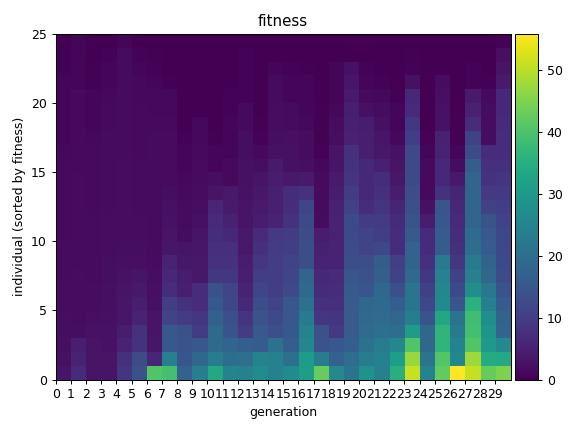
Run **evolve.py** and at the **output/fitness\_history.png**. After 30 generations or so, it should look similar to the image below (on the left). This plot shows the fitness of every individual in the population at every generation. We can see that peak and mean fitness are both increasing steadily. That fact is even more easily seen in **output/fitness\_history\_summary.png** (right image) which shows the progress of the population’s maximum fitness, mean fitness, and standard deviation of fitness in each generation.

Edit the **procreate\_with()** method in **SethController,** to change both of the ‘mu’ values to 0. This sets the rate of mutation to 0. What does this do? To the fitness history plots? Why?



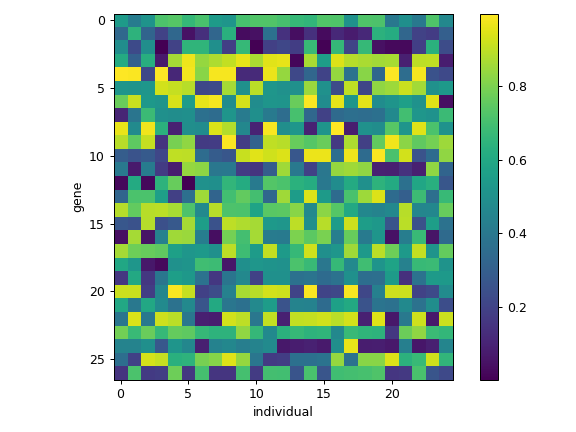
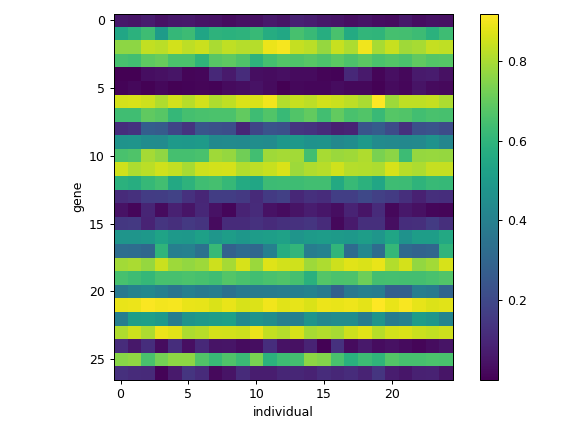
If we disable the test mode (i.e. set **TEST\_GA = False** at the top of **evolve.py**), and again run evolve.py. This makes the program recreate Seth’s study, i.e evolve robot controllers that seek food and water while avoiding traps.

The program runs much more slowly! It now has to evaluate over a hundred trials every generation (takes about 3 seconds per generation on my machine). And the output are different...

Identify and explain differences between these two plots and the two above. Why do they look so different?

**1 mark**

The images below show **output/population\_genepool.png** which plots the genotype of every individual in the population (in the most recent generation). Each column is an individual, and each row is a gene value between 0 and 1.

At the start of an evolutionary run, these plots look like the figure on the left. After several generations, they look something like they do on the right. Why? What has happened here? To understand this you may need to look at what happens in the **procreate\_with()** method of **SethController**.

**1 mark**

Make sure **TEST\_GA = False**, and run **evolve.py** for a while (50-200 generations). As it is running look at the figures generated in the output folder. You can also (as the evolutionary process is running), run `**animate\_best.py**` – which will generate an animation that shows the best performing agent in a random environment.

## PART IV: Investigate and tell me about it...

Write down approximately 500 words of observations taken during your investigation. What is happening? Has anything surprised you? What happens when you change parameters of the simulation (e.g. reduce the number of trials per fitness evaluation)? I have included a few flags in the code that allow you to make major changes to how it is working.

* In **SethController.\_\_init\_\_**, there is a line

**self.SYMMETRIC = True**

that determines if the controllers are symmetric (this is the default) or if the links on the right or the left of the robot are separately specified.

* In the **euler\_update()** method in **robot.pyx** there is a flag

**WRAP = True**

which describes what happens if the robots go too far away from the origin. When True, the robots ‘wrap around’ to the other side of the arena, when False, nothing happens! They are allowed to venture as far away from the origin as they like.

What does changing these flags do to the evolutionary process? What kinds of behaviours do these things encourage / discourage?

You can investigate these things or something else. What happens when you drain the batteries more quickly when the robot is moving faster?

Show me that you have investigated and thought deeply about this simulation and how it works!

**4 marks**

Report on your investigations here. You’re encouraged to paste images to help explain your observations.