

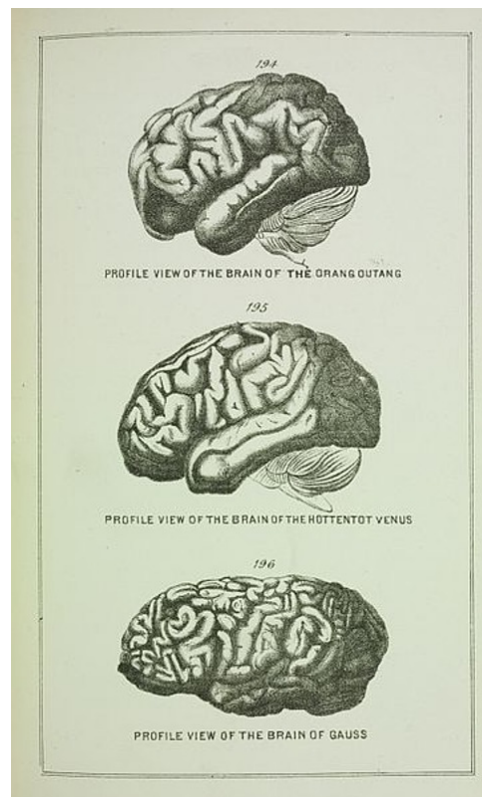
# C3

---

## Operators and Arrays

---

In this exercise you will build a simplistic computer model of Darwin's theory of evolution and use it to evolve solutions for equations. This will introduce you to arrays, how to iterate over them with loops, and how to use logic operators to control the loops. You will also practice the use of functions introduced in the previous exercise.



## **Schedule**

---

Preparation Time : 3 hour

Lab Time : 3 hours

## **Items provided**

---

Tools :

Components :

Equipment :

Software : gcc, notepad++ or equivalent text editor

## **Items to bring**

---

Identity Card

Laboratory logbook

Cover image: H.C. Chapman: Evolution of Life, 1873; Charles Darwin's Library. Public Domain; from: [commons.wikimedia.org](https://commons.wikimedia.org)

Version: October 14, 2019

©2018


Klaus-Peter Zauner

[Electronics and Computer Science](#)

[University of Southampton](#)

Before entering the laboratory you should read through this document and complete the preparatory tasks detailed in section [2](#).

**Academic Integrity** – *If you wish you may undertake the preparation jointly with other students. If you do so it is important that you acknowledge this fact in your logbook. Similarly, you will probably want to use sources from the internet to help answer some of the questions. Again, record any sources in your logbook.*

You will undertake the exercise working with your laboratory partner. During the exercise you should use your logbook to record your observations, which you can refer to in the future – perhaps to write a formal report on the exercise, or to remind you about the procedures. As such it should be legible, and observations should be clearly referenced to the appropriate part of the exercise. As a guide the  symbol has been used to indicate a mandatory entry in your logbook. However, you should always record additional observations whenever something unexpected occurs, or when you discover something of interest.

For each Task you should create a new directory so that you have a working version of every program at the end of the lab. Remember to place comments in your code.

You will be marked using the standard laboratory marking scheme; at the beginning of the exercise one of the laboratory demonstrators will mark your preparatory work and at the end of the exercise you will be marked on your progress, understanding and logbook.

## Notation

This document uses the following conventions:



An entry should be made in your logbook



Care should be exercised

command input

Command to be entered at the command line

# 1 Introduction

This lab introduces you to writing larger programs in C and to using functions to structure the program into modular units.

## 1.1 Outcomes

At the end of the exercise you should be able to:

- ▶ Use array data structures
- ▶ Use logic operators in conditions and loops.
- ▶ Use functions to organize complex code.
- ▶ Read code and figure out how it works.

## 1.2 Overview

Darwin's Theory of evolution can be viewed as an algorithm that runs a loop in which the essential ingredients are:

1. *Reproduction*: genetic information is passed from parent to offspring
2. *Variation*: reproduction is not perfect, offspring varies from the parents
3. *Selection*: the variants that perform best survive

In engineering one can sometimes learn some tricks from biology, this is called *biomimetics*. In the field of *evolutionary computation*<sup>1</sup> one simulates evolution with a computer to find good solutions to difficult engineering problems, such as antenna design,<sup>2</sup> or routing optimization in large networks.

In this exercise we will use a very simplistic version, that in practice would not be used, because it can get stuck for a long time on suboptima before finding a global optimum. The field has developed many techniques to address this issue; if you if you ever want to apply evolutionary computing a real problem, make use of these advanced techniques.

# 2 Preparation

Take a look at the skeleton code provided on the C3 laboratory notes page. The overall concept uses a single float number as the genome of an individual—an array of float variables represents the population. The task we want to solve is to find an  $x$  such that  $f(x) = y$ , where we specify  $y$ . The latter can

---

<sup>1</sup>This research field has three competing factions, that use the following terminology: ‘evolution strategies’, ‘genetic programming’, and ‘genetic algorithms’—you can also search under these terms if you like to learn more about this topic.

<sup>2</sup>[https://en.wikipedia.org/wiki/Evolved\\_antenna](https://en.wikipedia.org/wiki/Evolved_antenna)

be zero, as we can always include any offset in  $f()$ . The skeleton code has a simple sample equation  $x^2 = 4$ , i.e.  $x^2 - 4 = 0$ .

If we speak about “high fitness”, we mean something is good (or well adapted). To keep things simple in our example here, we use the distance to the target value as fitness—the smaller this distance the better. Therefore in the code you will find any fitness variables and constants prefixed with “i” (e.g. `ifit`,

`BAD_IFIT`), to indicate that the meaning is inverse, i.e. a low `ifit` value is better than a high value.

You will encounter a few new functions, not discussed in the lecture:

`pow()`: Takes the first argument to the power of the second argument: `pow(x, 2)` returns the value of  $x^2$ .

`fabs()`: Returns the absolute value of its floating point argument.

`srand()`: Seeds the (pseudo-)random generator with its integer argument as start value.

`rand()`: Returns a (pseudo-)random integer value between 0 and `RAND_MAX`.

Use the information above to see how the skeleton code implements parts of the simulated evolution. Read up (in the Textbook and/or online) on any material you require to understand the code—make notes in your logbook while you do that<sup>3</sup>.

- ▶ The `(float)` in front of `RAND_MAX` inside the `rnd()` function is called a *type cast*: it will turn the integer `RAND_MAX` into a floating point number before the division. Why is this necessary?
- ▶ The simulated evolution should stop when the best solution found yields a  $y$ -value no more than `EPSILON` away from the target value, or the maximum number of generations (`MAX_GEN`) has been reached. Write down the condition in the `while`-loop you will need for this.
- ▶ What part(s) of Darwin’s algorithm (Reproduction, Variation, Selection) is happening in the `for`-loop inside `main()`?
- ▶ Write down the condition needed in the `if`-statement inside `main()`.
- ▶ Draft the definition for the declared `initpop()` function. It should initialize all positions in the array with random values. (The evolution starts with a population of random genes.)
- ▶ Draft the definition for the declared `offspring()` function. It should copy the parent to position 0 in the population array (this makes sure we always keep the best solution from the previous generation—a parent is immortal until a better child emerges). And further, it should copy the parent to all other positions in the array, but vary the offspring such created by adding or subtracting a random value. The argument `mutst` (mutation strength) should be used as a scaling factor for this variation.

---

<sup>3</sup>By now you know that you should make a note of the source of your information, too.

### 3 Laboratory Work

#### 3.1 Part 1

Complete the skeleton code available from the C3 notes page using your prepared conditions and draft functions. Debug the code and run it.

Modify the equation to something that is not so obvious; e.g. change the exponent to make it a cubic equation ( $x^3 - 4 = 0$ ).

Make a table in your log book like the one below:

Run	RND_INIT	Generations	Solution
1	2		
2			
3			

Note the number of generations it took to find a (good enough) solution, and the found solution in the table. Then modify the seed value for the random generator (you can choose any new integer value you like) and run the code again, noting the seed, generations, and solution in the table. Repeat this until you have at least three entries in your table. What do you conclude from this?



#### 3.2 Part 2

Comment out or delete the `printhead()` function and the `printf()` in the for-loop in `main()`.<sup>4</sup>

Introduce a new variable, named `fit`, and set it to  $\frac{1}{\text{best\_ifit}}$  if `best_ifit` is larger than zero, and to  $-1.0$  if a perfect solution is found<sup>5</sup>. Fitness values here are normally never negative, so setting it to a negative number indicates a perfect solution that otherwise would cause a division-by-zero error.

Change the remaining `printf()` in `main()` to just print the generation, followed by a comma and a space, and then the value of the new variable `fit`.

Verify that your code outputs only the two values separated by a comma and space:

```
$ evol
0, 0.330953
1, 0.370935
2, 0.433937
3, 0.541325
4, 0.755303
```

Then use the shell output redirection as shown below to write the output of your program into a file rather than on the terminal.

<sup>4</sup>By now you know that it is a good idea to make a copy of your program before starting a new part.

<sup>5</sup>Even if it exists, it may not be representable as a binary number, and so never occur.

```
evol > myevoldat.csv
```

The example assumes that the executable is called `evol`. The extension `.csv` stands for comma-separated values.

Open the csv-file with the Excel spreadsheet program and produce a graph of how fitness progresses over the generations. What do you observe?



When you experiment with this, it is advisable to change the name of your csv-file for each experiment, to avoid confusing Windows/Excel.

### 3.3 Part 3

It would be interesting to investigate how population size and mutation strength influences the speed of evolution in our model. However, with what we know from Part 1, it is clear that we need to run each experiment with several different seed values to draw useful conclusions.

Modify the code from Part 2 such that it will repeat the simulated evolution several times<sup>6</sup> with different seed values for the random generator. To do that, wrap a for-loop around the while-loop inside `main()`. At the start inside this new for-loop set the seed of the random generator with the loop counter variable. Then reinitialize the population array by calling `initpop()`, and reinitialize the generation number (`gen`) and the best found so far (`best_ifit`). This is followed by the while-loop.

For debugging your code it may be convenient to reduce `MAX_GEN`.

To plot the output of the code with repeats, use a scatter-plot because you have multiple y-values for each x-value.

Make some experiments with different population sizes or mutation strength. Follow these steps:

1. Decide what you want to investigate
2. Decide what experiments you will run to investigate the issue decided in (1), make a note of this
3. Write in your log book what you expect you will find
4. Run the experiments
5. If you suspect you see a trend, decide whether you need additional repeats to make the result of the experiments convincing
6. Run the repeats
7. Document the outcome in your logbook



---

<sup>6</sup>By now you know that it will be a good idea to define a constant for this.

## 4 Optional Additional Work

We can take a different perspective on our population. Let's assume the individuals in our population (the array of floats) are fashion-conscious and hungry for attention. Limit the possible characteristics (float value) of the individuals to be in the interval 0.0–1.0. If three of them meet, then whoever stands out most in the group wins. To do this we calculate the difference between any pair in the group of three, and sum for any individual in the group the differences. The individual with the largest sum wins the round. As a reward a (small) number of followers are created by copying the winning individual with small random variations (staying within the 0.0–1.0 bounds). These followers are placed back into the array at a random location (i.e. they replace random other individuals). What do you expect to happen if we repeat this process indefinitely?



Implement the above scenario and visualize the whole population at each time step. To do this write for each individual in the population: the current generation, followed by the value of the individual to a file. I.e., for a population of hundred individuals you would have for each generation one hundred lines starting with the same generation number in the file. Use scatter-plots to visualize the temporal progress of the dynamics. (The y-axis covers the 0–1 interval and the x axis represents generation.) What can you observe?



This model has been investigated in detail here [1].

## References

- [1] Peter Dittrich, Fredrik Liljeros, Arne Soulier, and Wolfgang Banzhaf. Spontaneous group formation in the seceder model. *Phys. Rev. Lett.*, 84:3205–3208, Apr 2000. URL <https://link.aps.org/doi/10.1103/PhysRevLett.84.3205>.