LINUX VOICE CODING TUTORIAL

GENETIC ALGORITHMS: CREATE LIFE WITH PYTHON

BEN EVERARD

Everything's easy in Python. Even things that aren't easy to solve can be evolved with a little generic magic.

Computers follow a series of instructions step by step until they get to the end. This series of instructions is called a program. However, what if something can't be calculated with a series of step-by-step instructions? Or what if the series of step-by-step instructions would take so long to complete that running them is impractical?

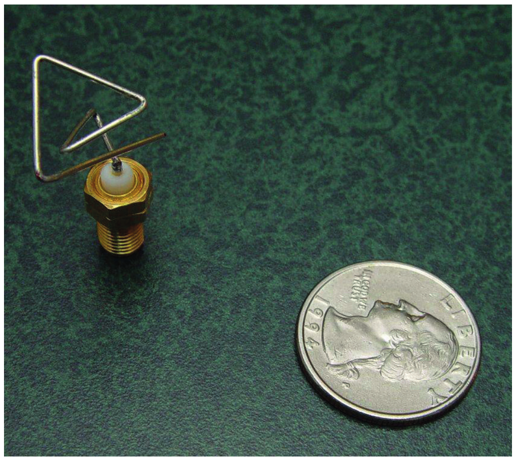
In these cases, we need a method that side-steps the main problem, but still attempts to find an answer. One way to do this is to use genetic algorithms. This mimics the natural process of evolution to attempt to solve a problem through a combination of randomisation, selection and combination.

The basic method goes like this:

- Create a random set of data in the right format to solve the problem.
- Apply some test to see which of the data solve the problem best.
- Combine the best pieces of data, throw in a little randomness and go back to step two.

In the real world, this is how we became us. Initially, there were some primeval organisms with some DNA and not much else. This was step 1. The weakest of these organisms died off leaving only the strongest. This was step 2. These remaining organisms reproduced. This is step 3. The final two steps have been repeating ever since life on earth started, and we are the result, as are all the other living things.

"Genetic algorithms mimic the natural process of evolution to solve a problem."


NASA use genetic algorithms to find the best antenna designs for spacecraft.

To model this computationally, the key thing we need is a fitness selector. This is the test that we'll apply to the data to see if it should pass on its characteristics to the next generation, or if it should be pruned from the evolutionary tree leaving stronger data to go forward.

Essentially, it's this function that defines your genetic algorithm and what data it will search for – it turns programming around, so that you write a program specifying what the solution should look like, then leave the computer to work out what it is.

Genetic square roots

Let's take a look at an example. There isn't actually an easy way to calculate square roots, however, it is very easy to go the other way around and calculate the square of a number. So, this is the sort of problem that genetic algorithms are good at. We'll program it in Python using the **pyevolve** module. This may be in your distro's repositories in the **python-pyevolve** package, or you can get it with

```
pip install stallion
```

The evaluation function for our square root finder is:

```
def eval_func(chromosome):
  score = 0.0
  for value in chromosome:
      score += 100000000-abs(((square_root_of-(value*value))))
  return score
```

This function will be passed a list of data that represent the organism that you're evaluating. In the case of our square root calculator, this will have just a single value, but in other cases, it could hold many values representing different aspects of the organism.

It returns a value that the genetic algorithm will attempt to maximise. In this case, it will try to maximise **100000000-abs(((square_root_of-(value*value))))**. **abs()** returns the absolute value of a number – this means that it just removes the negative sign on negative numbers, so **abs(10)** is 10, and **abs(-5)** is 5. The **abs()** call in this function, then, will return a larger number the further the value is from the actual square root. However, our algorithm will try to maximise the result, so we want this number to get smaller the further it is from the square root. To do this, we take the result away from 100000000.

We said that this function effectively defines the genetic algorithm, and this is true. However, we do need a bit more code to define the environment that the evolution will take place in. Since genetic algorithms rely on a certain amount of randomness to find the right values, there's no guarantee that they will ever find the right value. You can increase the chances of them working correctly by tweaking the environment for the particular problem you're trying to solve. The full code we've used is as follows:

```
from pyevolve import *

square_root_of = 1000

def eval_func(chromosome):
  score = 0.0
  for value in chromosome:
      score += 100000000-abs(((square_root_of-(value*value))))
  return score

genome = G1DList.G1DList(1)
genome.evaluator.set(eval_func)
genome.setParams(rangemin=0, rangemax=int(square_root_
of/2))
genome.crossover.set(Crossovers.G1DListCrossoverUniform)

ga = GSimpleGA.GSimpleGA(genome)
ga.setPopulationSize(square_root_of)
ga.setGenerations(50)
ga.evolve(freq_stats=10)

print ga.bestIndividual()
```

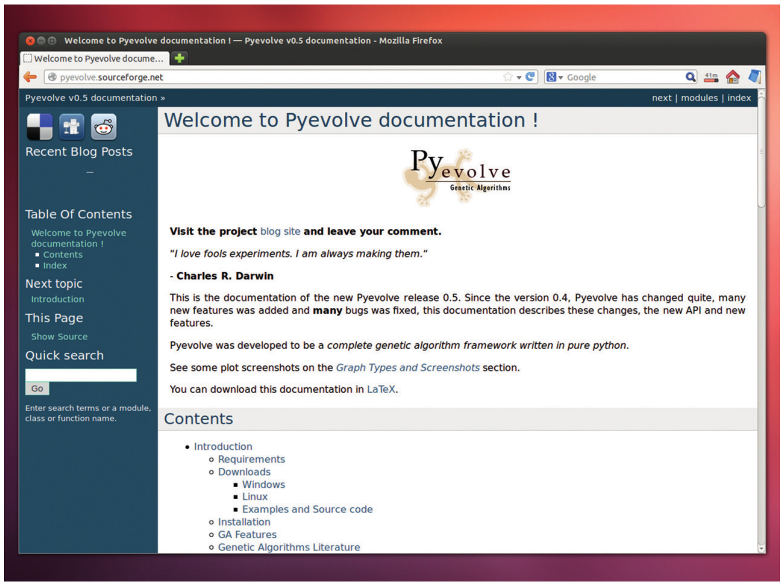
This code will attempt to find the square root of 1000, which is a little unfair since the software only works with integers. If it works correctly, it should find the closest whole number to the square root of 1000.

The variable **genome** holds an instance of **G1DList**. The parameter we gave when creating this is the number of items in the list. Once this variable is created, you can set certain attributes about it. The only thing that has to be there is the call to **evaluator.set()**. This tells the genome what function to use to test the fitness.

The other two things we've set aren't essential for it to work, but make it much more efficient. We've limited the range to between 0 and half of the number we're trying to find the square root of. The smaller we can keep the range, the less work the genetic algorithm will have to go in order to find the square root. Since we're dealing with integers, and this rounds up, it doesn't stop us getting the right answer.

The crossover is the way in which strong pieces of data are combined. There are quite a few options in **pyevolve**, but not all of them work with lists containing just one element.

The final block of code creates a genetic algorithm that takes this genome and evolves it. Again, there are some settings we can tweak to make the environment conducive for getting the right answer. The key value here is the population size. This is the number of organisms we create each cycle by combining the most successful from the previous cycle (and adding some mutations). We found that larger square roots required larger population sizes because the number


Welcome to Pyevolve documentation ! — Pyevolve v0.5 documentation - Mozilla Firefox

pyevolve.sourceforge.net

Pyevolve v0.5 documentation » next | modules | index

Recent Blog Posts

Table Of Contents
- Welcome to Pyevolve documentation !
  - Contents
  - Index

Next topic
Introduction

This Page
Show Source

Quick search
Go
Enter search terms or a module, class or function name.

Welcome to Pyevolve documentation !

Visit the project blog site **and leave your comment.**

"I love fools experiments. I am always making them."

- Charles R. Darwin

This is the documentation of the new Pyevolve release 0.5. Since the version 0.4, Pyevolve has changed quite, many new features was added and **many** bugs was fixed, this documentation describes these changes, the new API and new features.

Pyevolve was developed to be a *complete genetic algorithm framework written in pure python*.

See some plot screenshots on the *Graph Types and Screenshots* section.

You can download this documentation in LaTeX.

Contents
- Introduction
  - Requirements
  - Downloads
    - Windows
    - Linux
    - Examples and Source code
  - Installation
  - GA Features
  - Genetic Algorithms Literature

If you want to experiment further with genetic algorithms, the **pyevolve** module is well documented at **http://pyevolve.sourceforge.net**.

of values in the range is larger. Therefore, we set the population size to be the number of which we're trying to find the square root. There wasn't any clever calculation that drew us to this setting. We just tried a few different options, and this one seemed to work out the best.

You can also change the number of generations. This is, pretty obviously, the number of times you repeat the selection and recombination. Again, we came across the setting for this after a bit of trial and error. When you run the code, you'll see that it outputs the fitnesses every 10 generations, so you can easily see how quickly it's getting to the solution (or getting stuck at the wrong solution).

Now go and clone some dinosaurs!

That's all there is to it! This code is quite general-purpose, and you should be able to adapt it to your own problems. There is a certain amount of science/art/luck/witchcraft in finding the right values for the environment to produce a good result, and even with a good environment, there's no guarantee of getting the right answer. In fact, if you run this a few times, you'll probably get the wrong answer occasionally.

Genetic algorithms aren't great at every problem, but they can produce surprisingly good results to very complex problems as long as a good fitness function can be created. Essentially, it's a method of searching through a data set that's too large to exhaustively search, and where a simpler search (like binary search) won't work. Incidentally, binary searches do work well for finding square roots, and we've only used genetic algorithms here as an example.

Ben Everard is the best-selling co-author of the best-selling *Learning Python With Raspberry Pi*.