Genetic Algorithm - A Guide

By Mark Mathis

markmath@umich.edu

July 2018

In the words of Douglas Adams, “Don’t Panic”.

Hopefully, this document can explain a bit about how the overall genetic algorithm works, how it is implemented in python, and improvements which could be made on the algorithm. The algorithm was originally written by O. Albert and is a black-box optimization technique. This document should be in the same directory as a python version of the genetic algorithm. If the program ever gets lost, it can be found at https://github.com/lambdacubed/genetic-algorithm

Table of Contents

[Evolutionary Algorithm Overview 2](#_Toc519673872)

[Introduction to Evolutionary Algorithms 2](#_Toc519673873)

[Common Evolutionary Algorithms 3](#_Toc519673874)

[Our Evolutionary Algorithm 4](#_Toc519673875)

[Python Evolutionary Algorithm 6](#_Toc519673876)

[Initialization Files 6](#_Toc519673877)

[Files with Run Capability 7](#_Toc519673878)

[Genetic Algorithm 7](#_Toc519673879)

[Possible Improvements 13](#_Toc519673880)

[Zernike coefficients 13](#_Toc519673881)

[LabVIEW 13](#_Toc519673882)

[Algorithm as an object 13](#_Toc519673883)

[References 15](#_Toc519673884)

## Evolutionary Algorithm Overview

In this section, I will describe what an evolutionary algorithm is, common types of evolutionary algorithms, and the specifics of the python evolutionary algorithm implementation.

The algorithm used in this lab was originally called a genetic algorithm. While it may be possible that it is truly a genetic algorithm, it seems to be much more similar to an evolutionary strategy. These are both types of evolutionary algorithms, so I will call the implemented algorithm an evolutionary algorithm from this point on.

### Introduction to Evolutionary Algorithms

Much of this introductory information can be found on Wikipedia, so look there for a more detailed overview.

An evolutionary algorithm is a heuristic which mimics evolutionary selection to find the optimal solution to any given problem. Given that it is a heuristic, the algorithm isn’t guaranteed to converge on the best solution or any solution at all. The trade-off is that this type of algorithm doesn’t make any assumptions about what optimal solutions look like or where they are most likely found.

In the class of heuristics called evolutionary algorithms, there is a general procedure that is followed.

*Algorithm*

1. Create the first generation of individuals (define their characteristics i.e. genes)
2. Test the fitness of each individual in the environment
3. Repeat the following until the user terminates the algorithm
   1. Determine the most fit individuals and use those for reproduction (parents)
   2. Create new individuals (children) using crossover and mutation of the parents’ genes
   3. Test the fitness of this new group of people (parents and children)

There are a few words here that you might not be familiar with, so let’s explain the main aspects required to understand evolutionary algorithms in more detail. To explain these keywords, I will define **optimization object** to be the device that is changed (or optimized) to yield a desired result and **measurement object** to be the device that gives information about how well the system is performing.

#### Individuals

Individuals are made up of “genes” which fully determine the optimization object. If our optimization object is a deformable mirror, the genes should in some way represent all of actuator voltages of the deformable mirror. If the optimization object is a translation stage, the genes should in some way represent the position of the stage along each of its axes.

#### Fitness

Fitness is a measure of how well the system is performing. To measure the fitness of an individual, the individual sets the configuration of the optimization object, some amount of time elapses to allow that configuration to propagate through the system to reach the measurement object, and then the reading from the measurement object is taken. The reading from the measurement object is then processed to yield a single value, the fitness. There exists multi-objective evolutionary algorithms which use multiple values to determine the performance of an individual, but I won’t go into that here.

#### Crossover

Crossover is the process of combining the genes of parents to create a new individual. This could be randomly selecting the genes from the two parents, cloning a parent, or some other operation.

#### Mutation

Mutation is the process of randomly changing genes. There is a probability which controls the amount of mutation. This probability could control the proportion of genes that are changed, the amount each gene changes by, or both. If the probability is too high, the algorithm becomes less of an evolutionary algorithm and turns into a random search. If the probability is too low, the algorithm may get stuck optimizing a local extremum.

### Common Evolutionary Algorithms

There are only two types of evolutionary algorithms that are important to this guide: genetic algorithms and evolutionary strategies. These two algorithms are important because I believe the algorithm which is implemented here takes most aspects from evolutionary strategies, and whoever made the algorithm decided to call it a genetic algorithm. I will walk through the main evolutionary algorithm components for each type of algorithm. This is only an introduction to these types of algorithms, many papers discuss them in more depth along with their applications [1].

#### Genetic algorithms

This type of algorithm is not very similar to our implementation of an evolutionary algorithm because genetic algorithms traditionally have genes represented by binary numbers.

**Individuals** in genetic algorithms are normally represented as binary numbers (or binary settings for a specific device). While genetic algorithms are not necessarily confined to binary numbers, they are traditionally only used for them.

**Fitness** is the same as the objective the user is trying to achieve.

**Crossover** in genetic algorithms is very important. The probability for an individual to be created by using crossover instead of cloning is usually between 0.6 and 1.0. Given that there are *n* genes in each individual, the new individual created by the crossover process will have the first *x* genes be from parent 1, and the last *n – x* genes from parent 2 where *x* is a uniform random variable from 1 to *n*. This is a one-point crossover. Algorithms could decide to use an *m*-point crossover where the individual would switch parents to sample from *m* times.

**Mutation** for genetic algorithms is relatively simple because we are only dealing with binary numbers.

Let there be *n* genes in each individual.

Let represent the vector of genes of a given individual.

Let *p* be the probability that a bit in x will be inverted.

Let be a uniform random variable sampled for each bit

The mutation for each gene in the individual is the following:

#### Evolutionary strategies

This algorithm is fairly close to our implementation of an evolutionary algorithm except for a few details in the mutation as well as not using rotation angles. I’m also leaving rotation angles out of this entire introduction to evolutionary strategies to make this discussion a bit simpler.

**Individuals** in evolutionary strategies are represented as a vector of real numbers in addition to a few strategy parameters. The strategy parameters could be any other information about the individual that might be useful, but often it is the variances for the mutation of each gene. Instead of having an individual be a vector of length *n* which fully determines the optimization object, the vector would have length *2n*. The first *n* genes represent the optimization object and the last *n* genes represent the variance in mutation of each of those first *n* genes. Sometimes it is more convenient to have the same variance for each gene, so the individual would be a vector of length *n* + 1.

**Fitness** is the same as the objective the user is trying to achieve.

**Crossover** techniques in evolutionary strategies have many possibilities given that the vectors are real numbers. The crossover operation either creates a new individual from two parents or from the entire set of parents. For each gene, it either selects a random parent’s gene to inherit or it inherits a weighted average of the parents’ genes.

**Mutation** is a bit involved because having to mutate real numbers instead of binary numbers. The mutation process first mutates the standard deviations of the genes and then uses these to mutate the genes. Mathematically, the mutation process is as follows:

Let there be *n* genes in each individual,

represent the vector of genes of a given individual,

,

,

,

The mutation of the standard deviation of the *i*th gene is the following:

The mutation of the vector of genes is the following:

There are recommended values for tau and tau prime depending on *n*, but these values wouldn’t necessarily apply to what I’ve presented here because of the lack of rotation angles.

### Our Evolutionary Algorithm

Now that you are familiar with the general aspects of evolutionary algorithms, I will present the specifics of the evolutionary algorithm that we have in our lab. It follows the general evolutionary algorithm procedure presented in the “Introduction to Evolutionary Algorithms” section, so I will only present details on the individuals, fitness, crossover, and mutation.

#### Individuals in our evolutionary algorithm normally represent mirrors. So the genes for each individual either represent the voltages sent to the mirror, or the genes represent Zernike coefficients that eventually map to voltages sent to the mirror.

#### Fitness in our evolutionary algorithm is determined using a data acquisition device and figure of merit. The data acquisition device will give the program information in form of a voltage, a picture, a spectrum, etc. Then a figure of merit function (written by the user) simplifies this information into a single value indicating how well the system is performing.

#### Crossover takes genes from random parents. For the ith gene, the algorithm selects a random parent and inherits the value of that parent’s ith gene. This happens until all of the genes have been given values, and then the algorithm determines if the person would break the optimization object. If it would break the optimization object, it does this crossover process again.

#### Mutation is relatively close to the method defined in an evolutionary strategy. The difference is that we do not mutate the variance of the genes and we only use one variance in the genes.

Let ,

,

be a Gaussian random variable evaluated for each gene,

be a uniform random variable evaluated for each gene,

be a uniform random variable evaluated for each gene,

An individual is mutated if , and the individual would then be mutated by .

The could be defined as a new random variable by using the distribution function technique if you want to be very precise about the probability of mutation.

## Python Evolutionary Algorithm

Now that you understand the implemented evolutionary algorithm at a high level, I will present information about the algorithm at a medium level. This will include a discussion of important files and a flow chart of the algorithm using names of files and functions.

There are files important to anyone using or programing the genetic algorithm. These files can be separated into two types: files which initialize variables and files which can be run (similar to GeneticAlgorithm.py).

If you want to know the details of how the program works – or a low level understand of the algorithm, you should read the program itself. I put many comments in the code to explain what functions and lines of code do.

Much of the information presented here (except for the flow charts) is in the README.txt file in a very abridged version.

### Initialization Files

There are two things which will need to be initialized correctly for a proper run of the genetic algorithm: the data acquisition device and the genetic algorithm itself. To determine the initialization settings, the program reads variables from .ini files. These files are structured so that variables are placed on every other line with an explanatory statement at the top of the file. The variable values are the third set of characters separated by spaces. As the file uses space separated values, it is very dangerous to use tabs when entering in values. So **don’t use tabs**. Each variable will have its variable name as well as a comment after it explaining what that variable is or what its possible values are. The .ini files for the genetic algorithm and data acquisition are located in different places and give different information.

#### Genetic algorithm initialization

The default values used to initialize the evolutionary algorithm are contained in genetic\_algorithm.ini. Most of the variables this controls should be self-explanatory given the person understood the general evolutionary algorithm described above. A few important things to note is that the algorithm always searches for the highest figure of merit (if you want it to minimize it, make the figure of merit negative), the radial\_order variable doesn’t matter if you are not running the Zernike polynomial mode, you can only initialize the algorithm with a constant voltage or a gene file – not both, and the fom\_num will be useful if people are using the same device with different figures of merit.

#### Data acquisition initialization

The data acquisition default variables change for every type of data acquisition device. Because of this, each data acquisition device has its own folder. For a data acquisition device named “device1”, there is a “device1” folder and a “device1 properties.ini” file within that folder. For example, when using the Andor camera, the program accesses "Andor/Andor properties.ini" to get the initialization settings for that camera.

I’m sure the current genetic algorithm does not have every data acquisition device that will be used with the algorithm in the future, so I will go through the method to add a new data acquisition device.

1. Name your device. Make sure this name does not have any spaces because of the .ini file being space separated.
2. Create a folder for your new device. Within that folder, create a properties.ini file formatted the same as the other .ini files.
3. Write a class in the data\_acquisition\_devices.py file for your device with an \_\_init\_\_, acquire, figure\_of\_merit, and shut\_down function.
4. Include your device as an available option in the genetic\_algorithm.ini file, in DAQ\_DEVICES, and in initialize\_daq\_device() in data\_acquisition\_devices.py
5. Create a figure of merit function within figure\_of\_merit\_functions.py for your specific device to call within its figure\_of\_merit function.

Example: If you want to call your device "device1", create a folder named "device1". Within that folder, create a file named "device1 properties.ini". Adjust data\_acquisition\_devices.py to have an option for "elif (self.device == DAQ\_DEVICES[{insert index of "device1" in DAQ\_DEVICES}]):" and have it return the device1 class you created with an \_\_init\_\_, acquire, figure\_of\_merit, and shut\_down function. Then go into figure\_of\_merit\_fuctions.py and create a function named “device1\_FOM” which calculates the figure of merit you desire.

### Files with Run Capability

These files are good for testing and recreating experimental conditions. It is important to know that python looks beneath “if \_\_name\_\_ == "\_\_main\_\_":” when running a .py file

#### data\_acquisition\_functions.py

This file can be used to test a data acquisition device and its figure of merit function. You can specify the number of consecutive times to test the device as well as the figure of merit number to test a specific figure of merit calculation.

#### Set\_voltages.py

This program allows you to send voltages to a mirror. It specifies which mirror you are sending voltages to and which amplifier you are using. With this information, it can send a gene file (actuator or Zernike) to the mirror, send a constant voltage to the mirror, or enter a testing individual actuator mode. It also asks whether you will be in Zernike polynomial mode or not; but if you answer yes, you can only send files to the mirror.

### Genetic Algorithm

The entire operation of the genetic algorithm can be quite complex. I will go through different modes of operation of the genetic algorithm as well as show a flow chart to help clarify how the algorithm works.

#### Modes of operation

Currently, there are two modes of operation: optimizing actuators or Zernike coefficients. This mode is determined in the genetic\_algorithm.ini file with “zernike\_polynomial\_mode”. If this parameter is set to “False”, an actuator basis is used; if this parameter is set to “True”, a Zernike polynomial basis is used.

An actuator basis is the simplest programmatic genetic algorithm because this is how the mirror is represented in the program. By optimizing voltages, there are no additional computational steps to convert individuals to a mirror shape. This will probably make it the fastest (computationally) version of the genetic algorithm. The drawback of this version is that it does not give any physical insight or use optical phenomena to affect its search.

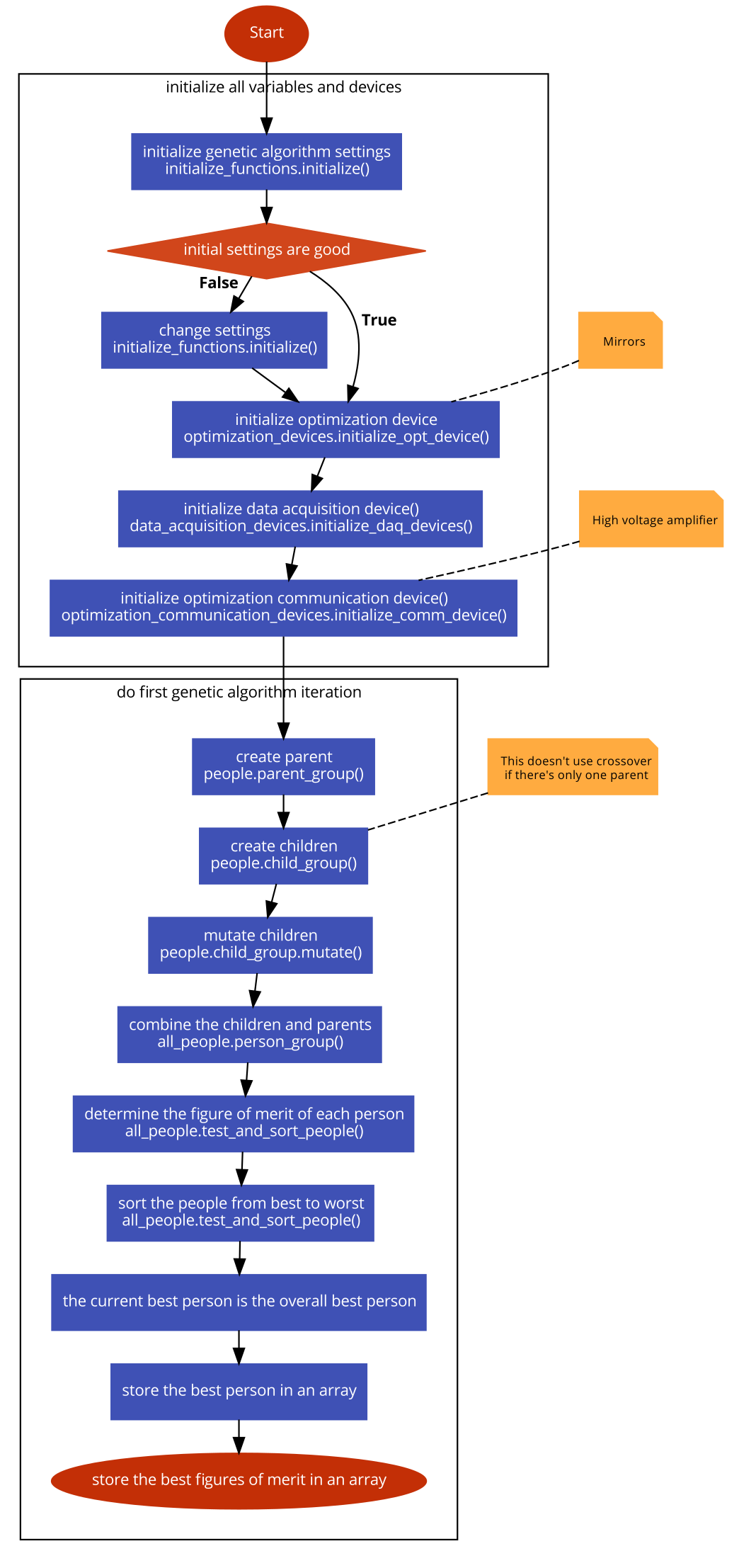
A Zernike polynomial basis is a physically motivated basis because low order Zernike polynomials represent common optical aberrations. While this algorithm may be slower computationally, the algorithm itself might be faster because of its ability to correct low order aberrations faster. It should be noted that a discrete sampling of the Zernike polynomials must be used. While the mirror gets rid of the discrete nature of the polynomial, the mirror might no longer look like a true Zernike polynomial. It would probably be valuable to measure the fit of the mirror to Zernike polynomials [2].

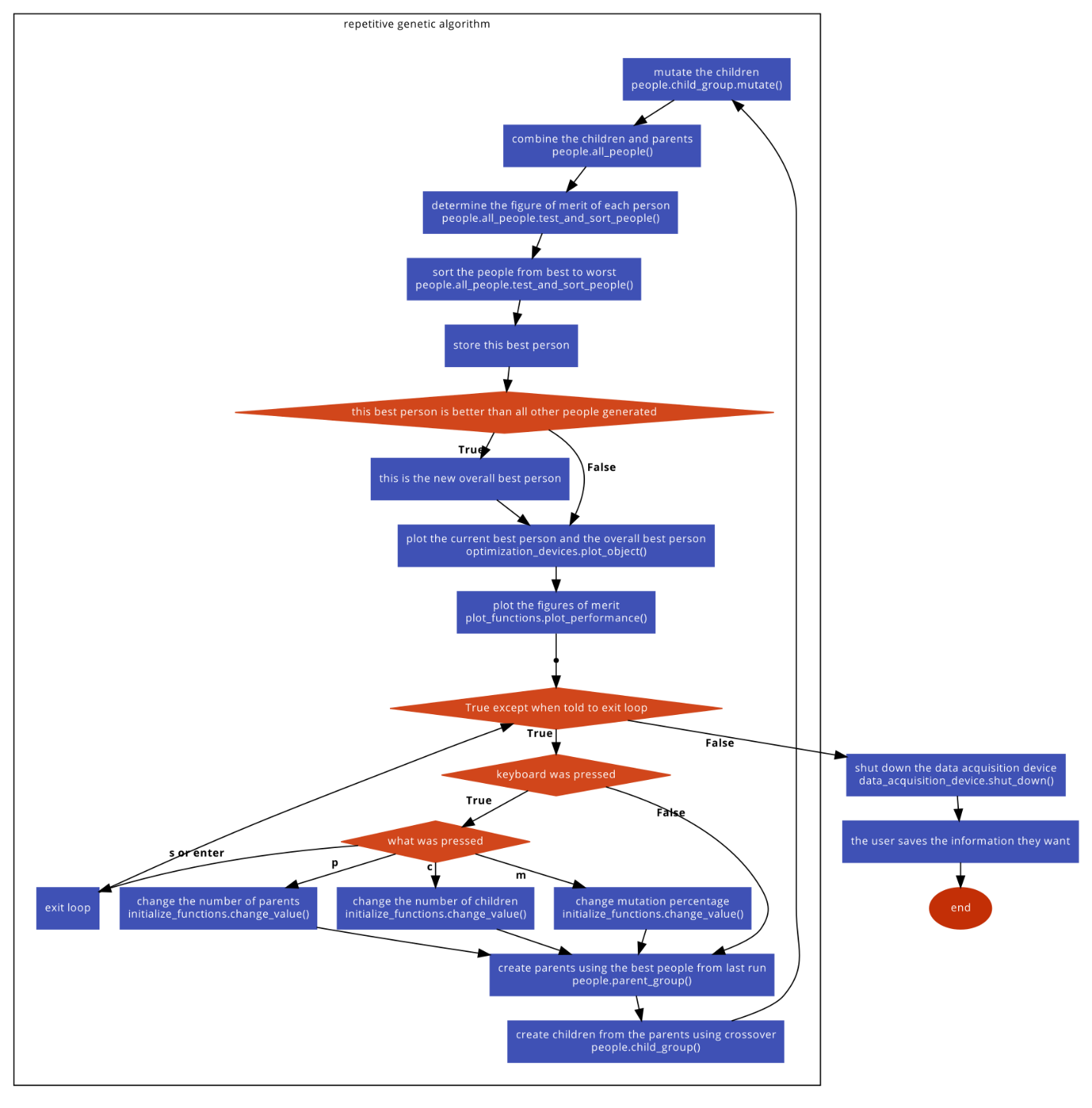
#### Flow chart

The flow chart describes the code that runs in GeneticAlgorithm.py. The first line of a box explains in plain English what that step of the algorithm is, and the second line of a box lists the name and location of that function within the program. For a function named add\_numbers within math.py, the box would look like this:

add numbers together  
math.add\_numbers()

Sometimes, there are two steps in the flow chart which I programmed into the same function. For this, you will see the same second line consecutively. Also, there are a few things in the flow chart that don’t have a second line. This means that either it uses a common python function which wouldn’t make sense to locate, or they are lines of code within the GeneticAlgorithm.py file itself.





Here is the code which generated the flow charts used above. If anyone updates the algorithm (or wants to generate a flow chart), go to code2flow.com and update the code below.

#### Flow chart 1

Start;

block initialize all variables and devices {

initialize genetic algorithm settings

initialize\_functions.initialize();

if (!initial settings are good) {

change settings

initialize\_functions.initialize();

}

// Mirrors

initialize optimization device

optimization\_devices.initialize\_opt\_device();

initialize data acquisition device()

data\_acquisition\_devices.initialize\_daq\_devices();

// High voltage amplifier

initialize optimization communication device()

optimization\_communication\_devices.initialize\_comm\_device();

}

block do first genetic algorithm iteration {

create parent

people.parent\_group();

// This doesn't use crossover if there's only one parent

create children

people.child\_group();

mutate children

people.child\_group.mutate();

combine the children and parents

all\_people.person\_group();

determine the figure of merit of each person

all\_people.test\_and\_sort\_people();

sort the people from best to worst

all\_people.test\_and\_sort\_people();

the current best person is the overall best person;

store the best person in an array;

store the best figures of merit in an array;

}

#### Flow chart 2

block repetitive genetic algorithm {

start:

while(True except when told to exit loop) {

if(keyboard was pressed) {

switch(what was pressed) {

case p:

change the number of parents

initialize\_functions.change\_value();

break;

case c:

change the number of children

initialize\_functions.change\_value();

break;

case s or enter:

exit loop;

loop start;

break;

case m:

change mutation percentage

initialize\_functions.change\_value();

break;

}

}

create parents using the best people from last run

people.parent\_group();

create children from the parents using crossover

people.child\_group();

mutate the children

people.child\_group.mutate();

combine the children and parents

people.all\_people();

determine the figure of merit of each person

people.all\_people.test\_and\_sort\_people();

sort the people from best to worst

people.all\_people.test\_and\_sort\_people();

store this best person;

if (this best person is better than all other people generated) {

this is the new overall best person;

}

plot the current best person and the overall best person

optimization\_devices.plot\_object();

plot the figures of merit

plot\_functions.plot\_performance();

}

}

shut down the data acquisition device

data\_acquisition\_device.shut\_down();

the user saves the information they want;

end;

## Possible Improvements

There are few things that could be improved about the algorithm itself including speeding up the Zernike coefficient calculation, rewriting the algorithm, and getting rid of LabVIEW.

Also, to improve the algorithm, maybe just using a different type of optimization heuristic might be the best answer.

### Zernike coefficients

The use of Zernike polynomials as a basis is quite new, so I didn’t spend a lot of time worrying about how fast the program runs. I really only spent time fixing things which didn’t work instead of the code that worked, but was slow. There are probably a few things which are slow, but the most obvious process is the way Zernike polynomials are converted to voltages.

Every time an individual is converted from their Zernike coefficients to voltages, the map from the 2D array

[[-1,-1,28,27,26,-1,-1],

[-1,29,14,13,12,25,-1],

[30,15, 4, 3, 2,11,24],

[31,16, 5, 0, 1,10,23],

[32,17, 6, 7, 8, 9,22],

[-1,33,18,19,20,21,-1],

[-1,-1,34,35,36,-1,-1]]

to the actuator voltage array

[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36]

is determined every time. It would be much faster if this was determined at the beginning of the algorithm in a look up table. This is a fairly simple fix that will probably have a large effect on the speed of the algorithm.

### LabVIEW

Currently, LabVIEW must be open to run the program. It would be great to be rid of LabVIEW so that the algorithm can stand on its own. I’ve tried to get rid of LabVIEW using PyVISA, but it didn’t work. PyVISA did not support PXI devices at the time. It might support them now, but I doubt it.

To get rid of LabVIEW, you must be able to communicate with PCI devices and USB devices through python. I’ve recently found the python library pcilib which will probably be your best bet. Also, a library I found for communicating over usb devices is pyusb, but there might be a better library.

### Algorithm as an object

Something to keep in mind when viewing this code is that it was transferred from LabVIEW, a graphical programming language, to python, an object-oriented programming language. Because of this, I originally made the program with only one object, people. After I had the program working, I revised the algorithm quite a few times so that the algorithm is more and more object-oriented. This has made the various files and objects itself more portable, so that it would be easy (relative to its LabVIEW version) to make the algorithm itself an object. The algorithm as an object would imply that there is a class which takes in an initial person and then has methods which cover crossover and mutation. This is probably going to mean that you combine people.py and GeneticAlgorithm.py.

This would make it much easier to change optimization algorithm types. You could have function with one of its inputs being which type of algorithm to use. Someone could create many different optimization algorithm objects and interchange them easily.

## References

[1] T. Back, H.P. Schwefel, "An overview of evolutionary algorithms for parameter optimization", Evolutionary computation, vol. 1, no. 1, pp. 1-23, 1993.

[2] N. Ling, X. Rao, L. Wang and S. Jiao, "Characteristic of a Novel Small PZT Deformable Mirror", in Adaptive Optics for Industry and Medicine - Proceedings of the 2nd International Workshop, G. Love, World Scientific, 2000, pp. 129-135.