Software Design of a Digital Filter Using Evolutionary Methods

Dr. Dick Blandford

University of Evansville

Introduction

Evolutionary computation was conceived and articulated in the 1960's as a method of solving otherwise intractable problems. Computer programs based on evolutionary techniques typically consume lots of computer resources and until the 1990's the technique was implemented only by the few who had access to those resources. Over the last ten years, computers have become fast enough and enough memory has become cheaply available that evolutionary computation is now a workable technique for a typical desktop computer running a program for several hours or over a weekend.

Evolutionary computation is a powerful technique and has been applied to a variety of problems including electric circuit design, time-optimal control circuits, quantum computer algorithms, robotics and many others[1]. The algorithms used to implement evolutionary computation are not complex and are accessible to the typical undergraduate engineering major.

This paper describes the fundamentals of evolutionary computation as it is applied to the DSP problem of designing a digital filter. It describes a computer program[7] which presents the user with a graphical user interface (GUI) for selecting filter parameters and other restraints on the algorithm. The program runs with a background priority and can be set to run for a few minutes up to several days. The evolutionary techniques of fitness, mutation, crossover, and growth are used.

Terminology

Evolutionary computation is based on Darwinian evolution and much of the terminology originates with terms defined in biological systems. The field is still very new and many of the definitions used here do not rigidly apply but are instead "consensus" definitions gleaned from other sources.

Evolutionary Algorithm – (EA) a generic term used for all programming methods which rely on evolution to achieve a stated goal.

Evolutionary Computation – (EC) any computation that makes use of evolutionary algorithms.

Evolutionary Programming – (EP) an evolutionary algorithm which uses mutation but not crossover to achieve a goal.

Evolutionary Strategy – (ES) an evolutionary algorithm which uses mutation and crossover to achieve a goal.

Genetic Algorithm – (GA) a generic term often used interchangeably with evolutionary algorithm. Some use the term "genetic algorithm" to mean those evolutionary

algorithms which rely most heavily on crossover, and in which mutation is not used or plays a minor role.

Genetic Programming – (GP) evolutionary techniques are applied to a problem to create a program. In genetic programming the output is a program, typically in Lisp, that has evolved to solve a particular problem.

Fitness – a measure or assessment of the quality of an individual item in a population as compared to an ideal. In an evolutionary algorithm, only the fittest individuals survive to the next generation.

Mutation – a random change in some portion of an individual's make up.

Growth – a small change in an individual directed toward a better fit to a goal.

Crossover – similar to biological reproduction. Characteristics of two or more parent individuals are combined by some algorithm to form a new individual.

How It Works

An evolutionary algorithm works in the following manner:

- An original population is selected. This is typically done in a random fashion so that each member of this population represents a solution to the problem. This original population is said to be the first generation.
- The population is assessed to determine each member's fitness. In a typical program the population is sorted with the best at the top.
- After sorting, parents are selected from the population. The parents are those members which are most fit. For example the top 30% may be chosen as parents.
- Crossover is performed using the parents to generate children. For example, two randomly selected parents may be chosen with parts of each going together to form a child. This process continues until all of the desired children are produced. If, say 30% of the population are parents then the remaining 70% may be replaced with new children.
- Mutation is performed. Mutation involves small random changes to certain elements of a member of the population without regard to whether or not those changes are beneficial. Typically some small percentage of the population is involved.
- Growth operations may be performed. Growth allows some small group of the population to change in a positive fashion. Growth is similar to mutation except that for growth, members are assessed immediately after a change and only positive changes are allowed.
- The population with the survivors (typically the parents) and the new children become the next generation.
- The process is repeated many times until some time limit expires or until some member of the population achieves a desired assessed value.

A Simple Example: Generating Phrases

As a purely academic example, consider the problem of using an evolutionary algorithm to generate a phrase such as "Twas brillig, and the slithy toves". We begin by selecting an original population of say 1,000 randomly generated phrases. For simplicity we will specify that each randomly generated phrase is the same length as the ideal phrase or 33 characters (counting the spaces as characters but not the comma). To assess the fitness of each phrase we sum up the differences between each character of a given phrase and the corresponding character in the ideal phrase using ASCII values. A partial list of the population and the error numbers might look like that shown in Figure 1.

If we choose the top 30% to be parents then we have 300 parents and 700 children. We choose two random parents to generate each child phrase. For example, referring to Figure 1, if we take strings 1 and 5 to be parents the two parents are

```
1  OYBV DWRTFJGLRRMFQEM AIDEMZMNUGCY
5  VCJEHISELMDA GYASODKHIBOFIUMXRJHP
```

No.	Err	String
1	295	OYBV DWRTFJGLRRMFQEM AIDEMZMNUGCY
2	337	BUMNKWELEEMGNENJATGGXWVGWHT NRPRU
3	338	XABP HPFZIBSGBDQYVAXJRFKYIF QOFUU
4	343	ULXSAPQWGLCMHNFU VKIDITVXHXJSNQVW
5	350	VCJEHISELMDA GYASODKHIBOFIUMXRJHP
..	..	...
996	649	EE QTLLEDBHDLKPH CDNHFM M NYGHEH
997	650	UZOZNTFRYM FDS REETOYYK PQNOO OE
998	662	VKHQUSDUGUYIKGHBH V YNPR YTFD E L
999	674	BUBLPFQX EZWRUOUEJDHASXYPURVC SB
1000	678	V WMGM BDAHLKKPXRWHVN VSCWKUSU U

Figure 1

The original population of 1,000 randomly generated strings. The error number is generated by subtracting ASCII values of each string from those of the ideal string and summing up the absolute value of the differences. The character set consists only of the capital letters plus the space character.

Next we randomly choose a substring from each parent. Suppose we take the substring `YBV DWRTFJGLRR` from parent 1 and the substring `BOFIUM` from parent 5. We then choose an existing child, say string 999 in Figure 1, and substitute the two substrings from the parents in a corresponding positions as shown in Figure 2. This forms a new child string. This process is continued until all 700 children are generated.

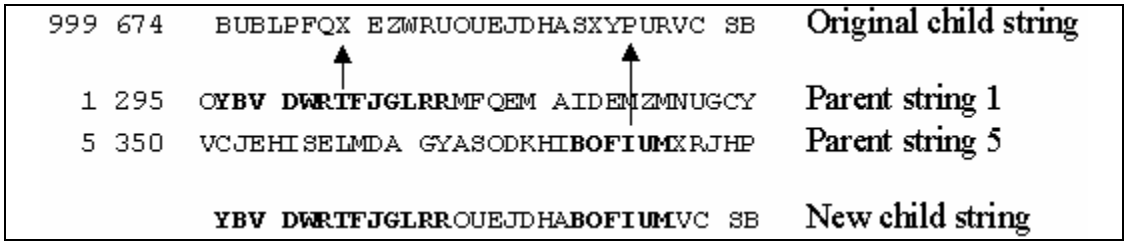

Figure 2

Substrings from parent strings 1 and 5 are chosen to fit into child string 999 to form a new child string.

After all of the children are generated by the crossover operation we select a small population for mutation. In mutation we change a small subset of letters in a child in a random fashion. The mutation may result in the child having a better or worse fit and there is no attempt made to do assessment of the mutation results at this time. Typically mutation is done to less than 5% of the population.

The process of growth is similar to mutation. Mutation is done to children but growth is applied to the surviving parents (the parents that have reproduced and will go on to the next generation). In growth we make a random change but we immediately assess its impact by determining whether the error became larger or smaller. The result of the growth is kept only if it leads to a smaller error value for the parent string.

The operations of fitness sorting, crossover, mutation, and growth are applied to the population in a given generation to form the next generation. The process is repeated for thousands (or perhaps millions) of generations. The operations stop when a given amount of time has passed or when the error is sufficiently small.

Applying Evolutionary Algorithms to Digital Filter Design

The design of a digital filter using evolutionary algorithms is analogous to the problem in the example above which creates a phrase. The output of the algorithm will be a transfer function made up of poles and zeros. The poles and zeros are analogous to the letters in the phrase creation problem. The objective is to create a filter which minimizes the error between itself and some idealized filter. An evolutionary algorithm has been implemented to achieve this objective. The processes of fitness sorting, crossover, mutation, and growth are described below:

- *Fitness* – for each filter that is created, an error is calculated which is typically the discrete root mean square (rms) error between the filter and the ideal filter. The error function can be weighted by the user as a function of frequency. Thus, the user can specify that error in the transition band, for example, is twice as important as error in the stop band. A data structure has been created to store the filter description and the error.
- *Crossover* – to do crossover, two parents are selected from the parent population. A pole or zero pair is selected at random from each filter. These parent pole or zero pairs replace corresponding terms in the child filter.
- *Mutation* – mutation is performed on a small subset of the children by moving a pole or zero term in a random fashion. Care is taken so that randomly chosen terms do not cause instability. Most often, this movement increases the error for the filter but occasionally it leads to good results.
- *Growth* – the growth operation is performed on a subset of the parent terms. This operation is considerably more complex than it is for the phrase problem. Each pole or zero can move in two dimensions. Effectively four new filters are created. Two new filters are created by incrementing or decrementing the magnitude of a pole or zero term by a small amount; Two other filters are created by moving the original pole or zero term by a small amount in the theta dimension. One of these four new filters is chosen at random and tested against the original filter. If the new filter is an improvement it is retained and the program proceeds. If the test fails the next filter is tried and so forth until all four new filters are tested against the original or one of them is found to have less error than the original. If the error is reduced the pole or zero term is allowed to stay in its new position; otherwise it is restored and no growth takes place. For this particular program design, only one pole or zero term is allowed to grow per parent per generation.

Program Description

The program is written in C# using the Windows .NET framework. (The program as a source project file and as an executable is available on the author's web site.[7]) It features a Graphical User Interface (GUI) and allows the user to determine the following items:

- The number of generations to run.
- The number of transfer functions (filters) per generation. This is the *population* size.
- The filter maximum order. The order of each transfer function can vary between 3 and the maximum number set by the user.
- The number of sample points. The error calculation, the filter magnitude data, and the frequency plot are all discrete and use this number of points (typically 256).
- The type of ideal filter. Two types are provided. In the first type an ideal filter is specified as a line plot and no ripple is specified. The error is the difference between this line and the calculated filter at a particular frequency. The second type allows the user to specify ripple bands and to adjust the weighting factor for each of those bands for the error function.
- The sample frequency. This is specified on the ideal filter selection window.
- The filter's normalization. The user can set the normalization so that the maximum magnitude plot value is unity or the user can specify that the filter be normalized to an arbitrary value at an arbitrary frequency.

The opening window for Evolution is shown in Figure 3. After entering the parameters the user clicks on *Begin* to start a run. For every 10 generations the filter status is displayed in the status window and for every 100 generations a frequency plot, pole/zero plot, and a pole/zero list is generated. The filter runs with background priority so that the computer on which it runs is not completely tied up.

A run completes when the error is reduced to zero or when the number of generations specified by the user is achieved. After a run is complete the user can graph the filter status or view all of the filters for the current generation and plot any one of them. The graph status shows how the error decreases with time as the generations progress.

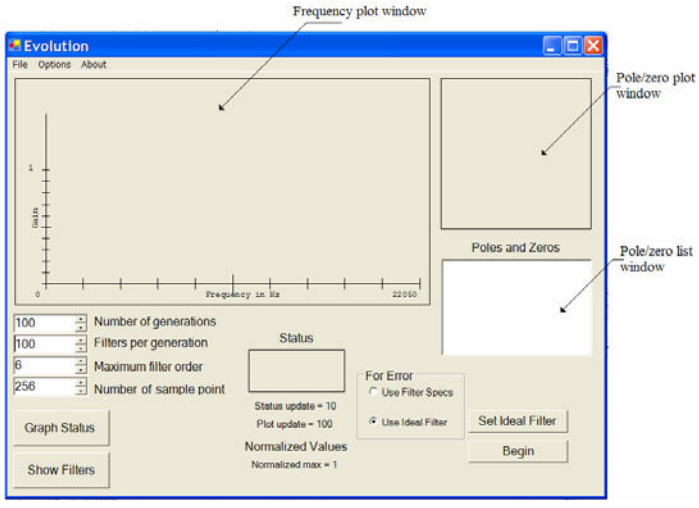

Figure 3
The opening screen of the Evolution program.

A Sample Run

To begin a sample run, we click on the "Use Filter Specs" button and select the "Set Filter Specs" button to get a window similar to that shown in Figure 4. Here we enter the ideal filter specifications from which the error will be determined. Exiting this window and going back to the main window (Figure 3) we set the "Number of generations" to 5,000, the "Filters per generation" to 100, the "Maximum filter order" to 6, and the "Number of sample points" to 256. Click on "Begin" to start the run.

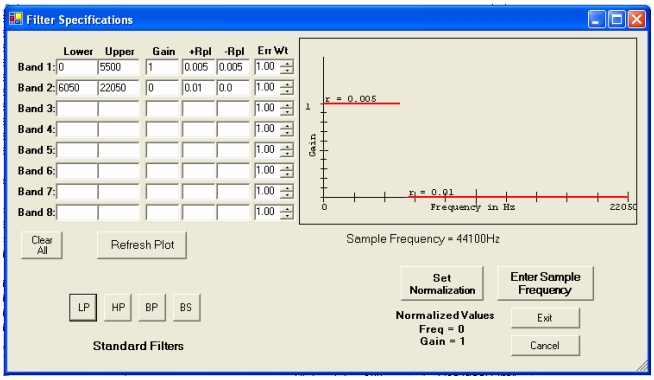

Figure 4

The setup screen for entering filter specifications for the ideal filter. The sample frequency has been set to 44,100 Hz. The filter has a pass band from 0 to 5,500 Hz and a stop band from 6,050 Hz to 22,100 Hz. The pass band ripple is ±0.005 (40db) and the stop band ripple is 0.01(40db). The filter normalization has been set such that the gain at 0 Hz is unity.

As the program runs, the frequency plot and the pole zero plot are refreshed every 100 generations with the current best filter. The status which includes the current error is displayed in the status window every 10 generations. After a few minutes,* the run completes its specified 5,000 generations and the final display is presented as shown in Figure 5.

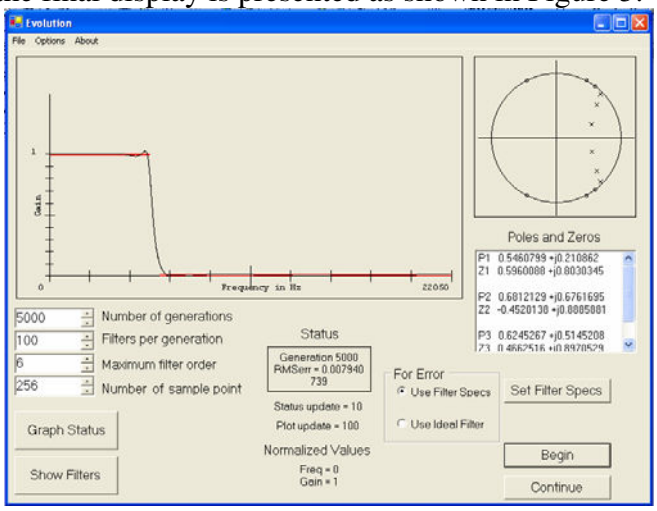

Figure 5

The results of a short run with the specifications set up as shown in Figure 4.
Note that the system selects pseudorandom numbers based on the start time so every run will produce a slightly different result. The final rms error for this run was about 0.0079.

*The run took about 3 minutes and 50 seconds on a 2.6 GHz P4 with 512 MBytes of RAM.

Click on "Graph Status" to see a plot of the error vs. the generation number as shown in Figure 6. This figure is typical. The error tends to drop off rapidly at first but as the filter improves, the error tends to level off and further improvement is slow.

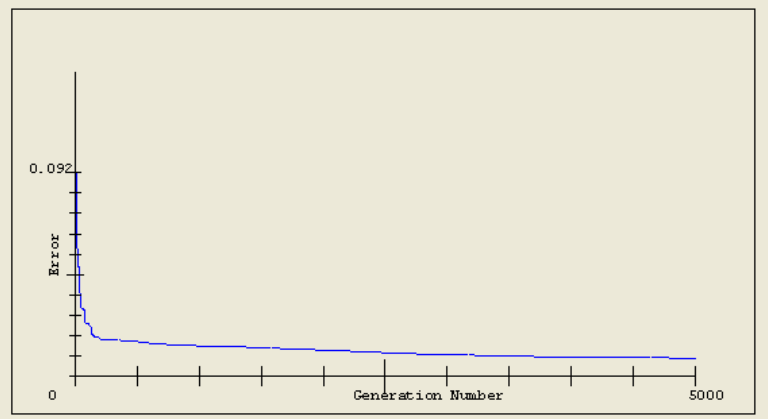

Figure 6
The error vs. Generation number plot for the sample run filter.

More Extensive Testing and Results

A standard low pass filter specification was created by creating specs that could be met by a sixth order elliptic filter. The cutoff and sample frequency was arbitrary. The filter specification is shown in Figure 7.

Sample frequency	44100Hz
pass band	0Hz to 5500Hz
pass band ripple	±0.005
stop band	6500Hz to 22050Hz
stop band ripple	0.01

Figure 7
Specifications for a standard low pass filter which can be met by a sixth order elliptic filter. This filter spec is shown graphically in Figure 4.

Twenty two lab machines were available for a run of 200,000 generations with 1,000 filters per generation. A typical 2.6 GHz Pentium 4 computer with 512 Mbytes of RAM required about 170 hours of run time to complete 200,000 generations. At the end of 200,000 generations none of the machines had generated a filter which fully met specifications. The error was considered to be zero when specifications were met; otherwise, the rms error was determined by using the discrete differences between the actual value of the filter and the specified value. The minimum rms error for all machines was 0.0047 and the maximum was 0.012. Figures 8, 9, and 10 show the frequency and pole/zero plots for the best filter and for an elliptic filter which meets all specifications.

Use in the Classroom

This program is used in an introductory class on digital signal processing as a supplement to a section in the text on direct design of IIR filters. Students are given access to the source code

and are able to use multiple lab machines for long periods overnight and on weekends. Students are asked to do a filter design to specifications and are able to modify such parameters as the number of parents, the number of mutations, the filter-weighting factors, and the method by which crossover is done.

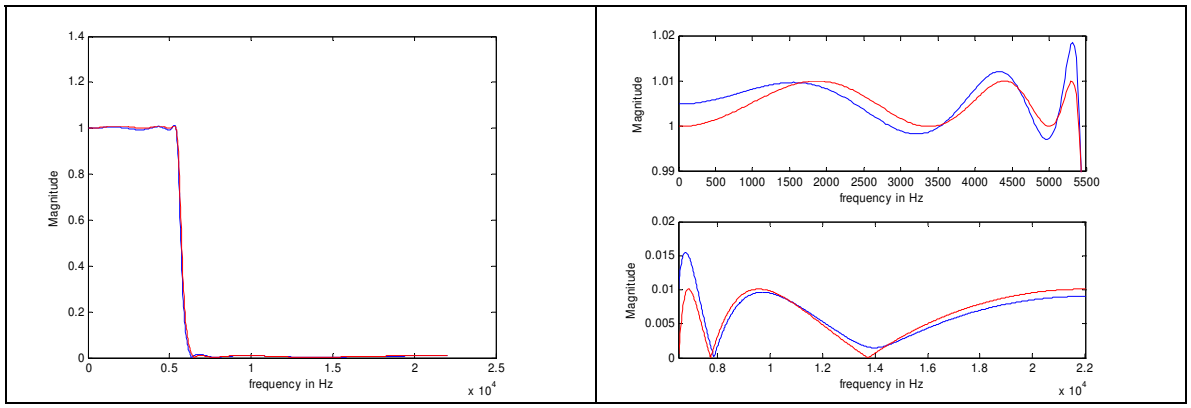

Figure 8

Linear plot of the best filter after 200,000 generations. The red plot is that of an elliptic filter which meets all specifications. The blue plot is that of the filter generated by a genetic algorithm. Exploded views on the right show the pass and stop bands.

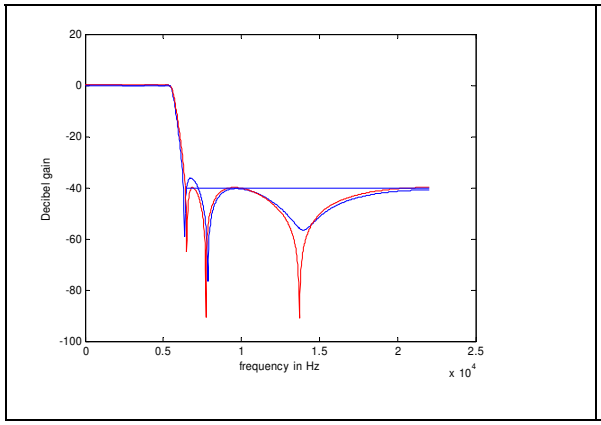

Figure 9

Decibel plot of the best filter after 200,000 generations. The red plot is an elliptic filter which meets all specifications. The blue plot is that of the filter generated by a genetic algorithm.

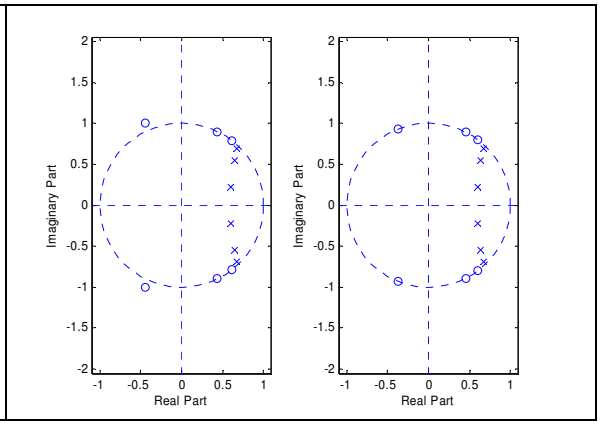

Figure 10

Pole/zero plot for the best filter (left) and an elliptic filter which meets all specifications (right).

Conclusions and Future Directions

Genetic algorithms rarely provide ideal solutions or solutions that are truly optimal. But they do provide solutions that are *statistically optimal* which means the solution found is the best in a very large sample. In this paper a genetic algorithm was applied to find the design of an elliptic filter, which is a problem that has already been solved in an optimal manner. This was done to gain a sense of how good a solution the genetic algorithm will produce in a reasonable time on a desktop computer. This program has produced results that are usable for some applications and

if the program were applied to a design problem for which no optimal solution yet exists, the output would be useful.

From a classroom perspective the program provides a mechanism to solve otherwise intractable problems. The genetic algorithm solution is not only applicable to dsp problems, but it is generally applicable to a wide range of problems in the engineering discipline.[1,2,4,5]

For the future there are three significant problems to be addressed that make use of this program and genetic algorithm.

1. What is the best arrangement of the number of parents in a population and the number of mutations that are allowed per generation? Empirical evidence from this program suggests that the number of parents should be about 10% and the number of mutants permitted per generation should be about 1%. But more study needs to be done in this area. These numbers provide a good solution for the early generations but growth seems to stagnate or stop altogether in later generations (> 200K).
2. Can the error-weighting for the specified filter be chosen in a manner such as to make the program converge on a solution more quickly? Some of the tests done with this program suggest that an error-weighting that changes as the generations progress might be useful. Due to the long run times, the author has had little chance to explore this area.
3. Interactive evolution[5] has been considered by others and may be applicable to this problem.

References

1. Koza, John, Keane, Martin, and Streeter, Matthew, "Evolving Inventions", Scientific American, February, 2003 pp. 52-59.
2. http://www.genetic-programming.org/
3. Haupt, Randy and Haupt, Sue Ellen, Practical Genetic Algorithms, Wiley-Interscience, 1998.
4. Koza, John, Genetic Programming, MIT Press, 1992.
5. Koza, John, Genetic Programming II, MIT Press, 1994.
6. http://www.genetic-programming.com/
7. http://csserver.evansville.edu/~blandfor

Biographical information

DICK K. BLANDFORD

Dr. Dick K. Blandford is the Chair of the Electrical Engineering and Computer Science Department at the University of Evansville.