

# Designing two tools for general purpose optimization

Chad R. Brewbaker  
 Depts. of Computer Science and Mathematics  
 Iowa State University  
 Ames, IA 50010  
 Email: crb002@iastate.edu

**Abstract**—The aim of this paper is to improve upon the usability of general-purpose optimization tools. We present two tools. The first is a minimalist general-purpose optimizer. The other an implementation of a method for describing complex search spaces. The design considerations of both are given along with a short survey of existing software

## I. INTRODUCTION:

Today, general-purpose optimization software is applied towards many areas such as aircraft design, VLSI, Motion Planning/Robotics, Financial/Business Planning, and scores of other endeavors. Solvers that can perform amazing feats of optimization have been written, but that is not our primary goal. Our interest lies in reducing the amount of complexity, time, and expert knowledge needed to use these powerful applications.

This interest in usability comes largely from our experience with applying general-purpose optimization techniques to our problems. The process consists of spending hours finding or creating libraries of general-purpose optimization routines and coupling them to a simulation with source code in C or Java. After hours frustration we wanted to come up with some optimization tools that put the human at the center of the optimization process, not the source code. We had the following requirements:

- \*The optimizer must work without having to touch any source code.
- \*The optimizer must employ some general purpose optimization routines.
- \*The optimizer must have a minimalist interface so it can be used on portable devices with small screen space.
- \*There must be a way to express complex simulations that is not tied to a specific programming language or set of libraries.

## II. MINI SURVEY

We present a short survey of existing software. The main category of software exists in the form of source code libraries. Among the hundreds we have seen here are three: Openai, <http://openai.sourceforge.net/>, is a java development library for machine learning. It also has a few graphical tools for designing neural nets. Open Beagle, <http://www.gel.ulaval.ca/beagle/>, is a C++ library for evolutionary computation.

DAKOTA(Design Analysis Kit for Optimization and Terascale Applications), <http://endo.sandia.gov/DAKOTA/>, is another

C++ library for optimization written at Sandia national labs aimed at high performance computing. The amount of work required to set up a simulation with this library seems daunting.

Another class of general-purpose optimizers is those aimed at being a network resource. They tout the ability to use them from a remote location, and the ability to harness the power of many computers on a network. They seem to be in their infancy, but here are two examples: Genetic Daemon, <http://geneticd.sourceforge.net/>, as a background process. The downside is that there is little documentation, and it seems to require the user examine the source code in order to use it.

Genetic Server, <http://geneticserver.sourceforge.net/>, is another network aware optimization tool. Working with the source code is required to use it.

The third area of general-purpose optimization software that we found was in a league of its own. GOSSET, <http://www.research.att.com/njas/gossetindex.html>, is targeted specifically at the construction of experimental designs. It comes with a specialized language that lets users describe the type of model they want to construct. This is then compiled into a solver application and ran. Two things about this solver stood out. First of all, no source code had to be touched in order to get it working. However, one may argue that the specialized language had to be studied. Secondly, it employs a solver that has had success handling over 1000 variables at a time. The method used is called Hooke and Jeeves pattern search.

Here is a list of the bad things we found from our short survey: Every one, with the exception of GOSSET, required the user to touch C,C++,Java,Python, or Pearl code. The documentation for the most part was poor. Also, the need for such extensive documentation shows the level of complexity that users must deal with. There was no common framework. Every library had its own interface, and few of them could be explained within a page of text.

Here is a list of good things we found from our short survey: GOSSET runs on any POSIX compliant OS ( Linux, Mac OSX, Solaris,...). We would like to keep this functionality. Small documentation. We want our applications to be usable without a 50 page manual.

Our answers to these shortcomings and good ideas is a general-purpose optimizer we named "Gnu Swiss-army Chainsaw", or GSC. In addition to flaws found in existing software

we aimed to address other design considerations.

### III. DESIGN CONSIDERATIONS

The first problem found with interactive solvers is that they can produce too many updates to the user at once. For example, at the start of many problems the computer finds 1000 better solutions in a row. If all of the solutions are displayed to the screen, then the flurry of messages might make the user feel that the application is out of control. However, if the solver has not found a solution for some time, then this can lead to a feeling of euphoria that the solver is not stuck. Our solution is to only update the best result every 5 seconds. This gives the user enough time to glance at a solution, yet it does not bore them since the updates come at 5-second intervals.

The second problem is that many times the solver does not find a better solution for minutes to hours on end. The knee jerk reaction is to add some sort of progress bar or flashy spinning icon to let the user know that the application is still alive. Instead we take this as a feature. The computer has the optimization task under control; so let the user get bored and find something constructive to do. They should be excited by good results, not cute stuff to waste their time. However, users do get curious as to when a solution was found so a timestamp is added to every solution.

Another problem is the need for a standard interface with the underlying simulation or objective function. Our solution is to only optimize real valued functions with variables between 0 and 1. The first consideration was that these are natural inputs to most engineering and financial models when properly scaled. The problem lies in representing high level data types. There are some simple solutions for this. For example, a permutation can be represented as a vector of reals, and sorted to obtain the order of the elements. To meet the needs of high-level inputs we developed a form of grammatical evolution software called GELEX. For now think of it as mad libs for problem solving. This is not a perfect solution, but one many will be happy with.

Another problem is letting the solver give output to visualization software. If the optimizer were designing something like an aircraft wing, engineers might like to see a drawing of it and not just the variables for the dimensions. This should be easy to implement because our solver writes to stdout, so the output could be easily piped to another application.

This brings us to the issue of graphical user interface, GUI vs. command line. In our experience GUIs allow the user to do two things. The first is to decrease the amount of memory a user has to employ by giving menus and dialogs. The second is to provide a visual interface for input such a drawing. One of the goals of this project is to eliminate the need for manuals and menus. The CLI was chosen for its ability to be pipe output in a POSIX environment, and its small use of space, which is essential for wearable and portable displays. How long should the solver run before stopping? The solver is only going to run for a finite amount of time, but many times a user does not know that they are going to get interesting results only after 6 hours of computation. We chose for the solver to run

until the user kills it. Inputting an automated stopping time may be convenient for some, but this requires another input variable. An issue of readability comes from distinguishing the value of the objective function from the rest of the data. To separate this we add parentheses around the objective function value and the time stamp.

Also, there is the issue of precision. How many significant digits should be output? Most applications cannot handle more than an IEEE 754 double precision floating point. Although the thought of using arbitrary precision floating point numbers was batted around, we decided to stick with the IEEE 754 double. 52 bit fractions and 11 bit exponents are enough for most users. Arbitrary precision, in our experience, is great for numerically verifying results against Plouffes Inverter, <http://pi.lacim.uqam.ca/eng/>, but is not of much practicality to real world optimization problems.

Performance is also a key part of usability. If the optimizer does not perform well nobody will want to use it, no matter how good the user interface is. One performance hit we take is decoupling the objective function from the solver. In order to call the objective function a child process must be forked, pipes must be set up for communication, and the objective function application must overwrite the child process. On modern computers this takes only a few milliseconds, but for very simple objective functions this is a lot of overhead. This method scales up well because the ratio call over head to simulation runtime is negligible as objective functions take longer to compute. We feel that this is an acceptable tradeoff, but we would like it if this over head could be reduced.

Another issue of performance is the underlying optimization algorithms used. We have chosen three robust methods: genetic algorithm, simulated annealing, and Hooke-Jeves. The are ran in a round robin fasion with a quanta of timesteps for each. This quanta grows as the simulation progresses. The exact value for this quanta has not been expirementaly determined, so we set it initially at one second, and the quanta doubles each round.

Future incarnations of the optimizer might include a tabu list or lookup table of previously computed results.

Finally, the issue of many variable simulations comes up. When users are optimizing hundreds of variables at once they are more interested in the value of the objective function at a given timestep than the list of variables. Thus, the value of the obective function must be displayed after the variables. Otherwise, the result of the objective function would scroll off screen before the user has a chance to look at it. Here is an example of GSC optimizing the 5 variable objectiveFunc.exe:

```
popeye:~>gsc objectiveFunc.exe 5
.33334 .36326 .23436 .23423 .33424
(234.47 1:30pm)

.36333 .36386 .23436 .23423 .33424
(236.93 1:31pm)
```

Note that the columns are tab spaced for elegance of presentation. The result is placed below the command prompt to get the users attention. It is our experience as command line interface users that we are used to the screen scrolling

down and tends to train our eyes at the prompt and below it. It would be interesting to find some empirical evidence on how CLI users scan a terminal window, but we were unable to find any data.

#### IV. GELEX

The second goal of this project was to make a robust implementation of one of the newer tools for optimization, grammatical evolution. Although it might more appropriately be called mad libs for modelers. Many remember the joys of playing the game mad libs as a child. One person would read states out of a book such as: VERB, PERSON, ADJECTIVE, NOUN, and PLACE. The second person would come up with expansions to the states like: NOUN: hula-hoop PERSON: Margret Thatcher — Blas Pascal Then, these states would be expanded for a mystery sentence resulting in something like:

PERSON went to PLACE for NOUN lessons, but instead met up with PERSON at the PLACE.

Blas Pascal went to Omaha for hula-hoop lessons, but instead met up with Margaret Thatcher at the gas station.

Here is an example in formal notation. To the left are state names followed by a colon, and to the right are expansions separated by a bar and finished off with a semicolon. Expansions in a grammar always start from the first state listed, in this case SENTENCE.

```
SENTENCE: NAME is my PET;
NAME: Fluffy | Waldo | Terence | Alf;
PET: cat | goat | llama;
```

```
Fluffy is my goat. Terence is my llama.
Alf is my goat. Waldo is my cat.
```

All of this may seem like silliness, but humans seem to quickly pick up the concept of grammatical expansions. Compiler writers and English teachers have done much to curb interest in the subject. Now that our stigma of grammars has hopefully worn off, lets see how we can use them in modeling complex problem spaces. Here is a grammar to solve a motion planning problem with a robot on a 2D grid:

```
BOT: move( DIR, DIST) BOT | move(DIR,DIST);
DIR: up | down | left | right;
DIST: 1 | 2 | 3 | 4;
```

A grammar for the automatic generation of math simple functions:

```
EXP:EXP OP EXP
| (EXP OP EXP)
| FUNC(EXP)
| VAR;
OP: + | - | / | *;
FUNC: Sin | Cos | Tan;
VAR: x | y | 1.0;
```

Here is a grammar for growing a tree like antenna in 3D.

```
ANT: SPLIT | stop | GROW;
SPLIT: ANT GROW | ANT STOP;
GROW: DIR GROW | DIR SPLIT | DIR STOP;
```

```
DIR: x | y | z | -x | -y | -z;
```

What does this have to do with GCS? We can use GCS to automatically find expansions in our grammar that represent possible solutions. The approach is to map the interval (0,1) to an expansion for some state in a grammar. The nave way is to split (0,1) into equal segments based on the number of expansions in a given state. Bringing back the pets example we get:

```
SENTENCE: NAME is my PET(0,1);
NAME: Fluffy (0,.25)
|Waldo (.25,.5) | Terence (.5,.75) | Alf(.75,1.0);
PET: cat(0,.33) | goat(.33,.66) | llama(.66,1);
```

[.38, .34 , .80] evaluates to: Waldo is my llama.

[.36 .93, .53] evaluates to: Alf is my goat.

Our version of the grammar expander is called GELEX. It can compile the grammar into a finite state representation in C, take in a real vector as input, and expand the grammar in the nave fasion. The effects of the expansion can then be tied in with a simulation code. Unlike GCS, GELEX requires intimate knowledge of the simulation it is interfaced with. GELEX is meant to simplify the input to a simulation.

#### V. USABILITY

We must now ask ourselves the question: How useful is grammatical evolution, and how useful is it in the design of a simulation? Most applications of this technique we found in the literature are toy problems. Ultimately, we think the usability of this tool is much like the usability of any compiler. The user must first learn the language. The format of our grammar language is extremely simple, so this task should be the easy part. Secondly, the user must learn the art of describing his ideas in this new language. With GELEX the user starts with a clean slate. There are no predefined grammars. If the user is faced with the daunting task of writing a finite state expander for some grammar, GELEX might be a helpful tool. Otherwise, it will probably not be of much use. How can GELEX be made more usable? The first option is to compile finite state machines in as many languages as possible. Our current incarnation only outputs finite state expanders in C, but there is no reason it could not output Java, Python, or any other language. The second idea we came up with was tweaking the representation of our context free grammar. Past implementations have used Backus-Naur form grammars that look like:

```
<exp>:= <exp><op><exp>
|<var>
<op>:= + | -
<var>:= x | y
```

We had a few problems with this representation. First of all were the extra ( $\iota, \iota$ ) symbols. Secondly, the = symbol can be removed. Finally, the end of line character ; was added to denote the end of an expression. As python has shown, this makes for clean representation. However, the user might have to deal with white space considerations, so it was inserted. The grammar now looks like:

```
exp:  exp op exp | var;
op:  + | - ;
var:  x | y;
```

Another problem was the use of forbidden characters. This was accomplished by using the escape character trick of printf statements. The question is does this added flexibility influence usability? This is a problem of plain text only interfaces. There is no way to color code certain expressions, so more convoluted expressions must be constructed to differentiate between meanings.

Another problem with BNF form was that it did not allow for weighted grammars. For some simulations it is favorable to give unequal weighting to different expansions within a state. Thus, we end up getting grammars that look like:

```
exp:  exp op exp(.75) | var (.25);
```

Thus, `exp op exp` would be triggered by 75% of inputs and `var` would be triggered by 25%. Where should this feature creep stop? When does the grammar expression language get to complex? From a usability standpoint we think the addition of escape characters and weights is ok. This is because they are not required. Grammars without weights and escape characters will work perfectly fine.

## VI. FUTURE WORK

We intend to release the beta versions of both GSC and GELEX on <http://www.sourceforge.net> this summer. It is a free service that allows users of the software to post comments on the design, report bugs, and offer other advise. Hopefully, from this process we will notice other design flaws that we have not addressed. With GELEX the comments should be extensive. Tool is only useful if the target language the user wants their grammatical expander in has been implemented. A meta tool to help with porting it to new languages has been thought of, but this is outside the scope of this project.

Bibliography:

## REFERENCES

- [1] Toby Ord and Alan Blair, *Exploitation and peacekeeping: introducing more sophisticated interactions to the iterated prisoner's dilemma*, Proceedings of the 2002 Congress on Evolutionary Computation, 1606–1611
- [2] Daniel Ashlock, *Complex Adaptive Systems Text*, in preparation.
- [3] Connor Ryan and Michel O'Neil, *Grammatical evolution papers*
- [4] Robert Axelrod and William D. Hamilton, *The Evolution of Cooperation*, Science, New Series, Volume 211, Issue 4489 (Mar.27,1981)
- [5] David Allen, *Human-computer interaction*, Prentice Hall, New York, 1993
- [6] Robert Axelrod and Douglas Dion, *The Further Evolution of Cooperation*, Science, New Series, Volume 242, Issue 4884, (Dec. 9, 1988)
- [7] Chad Brewbaker and Daniel Wengerhoff, *Optimization and Analysis with Centroidal Voronoi Tessellations*, NSF REU 2001 technical paper, Iowa State University, [http://www.math.iastate.edu/reu/2001/voronoi-paper/voronoi\\_paper.html](http://www.math.iastate.edu/reu/2001/voronoi-paper/voronoi_paper.html)
- [8] Jenny Preece, *Human-Computer Interaction*, Addison-Wesley, New York, 1994
- [9] Max Gunzburger, Qiang Du, V. Faber *Centroidal Voronoi tessellations: applications and algorithms*; SIAM Review 41, 1999, 637-676  
*UNIX* Robbins, Steven and Robbins, Kay A. *Unix Systems Programming: Communication, Concurrency, and Threads*, Prentice Hall, New York, 2003
- [10] Aho, Sethi, and Ulman, *Compilers: Principals, Techniques, and Tools*, Addison-Wesley 1986, ISBN 0201100886