

GPRS-kit 0.2 User Guide

Henry Cook
hcook@eecs.berkeley.edu

1 Introduction

This toolkit provides a set of tools for automatically creating a response surface for a target design space. Response surfaces are equations that express a performance metric used to evaluate a set of possible designs in terms of the designs' parameters. If you can parameterize your design and quantify a performance metric of interest to you then this toolkit can vastly reduce the number of highly accurate simulations or time-consuming experiments you must conduct in order to find globally optimal designs or important design space trends.

By creating a predictive response surface from the measured performance of a small subset of designs, it is possible to evaluate all possible designs without taking the time to experimentally evaluate every single one. Numerical or analytic methods can be applied to the response surface in order to identify local or global optima, trends, and variable relationships. See [1, 3] for more information.

Following in the footsteps of Alvarez [1], this toolkit uses genetic programming to create response surfaces from a suggested sample subset. Genetic programming works like a standard genetic algorithm, but operates on a population of non-linear polynomial equations (i.e. potential response surfaces) rather than a population of designs.

While the sample used to create a GPRS can be randomly selected, it is often beneficial to use a Design of Experiments to guaranteed a representative sample. The Audze-Eglais Uniform Latin Hypercube Design of Experiments is particularly robust. This toolkit provides a reimplementaion of the Audze-Eglais formulation algorithm described by Bates et al. [2].

This software is open sourced under the GPL license. I am not a professional software developer, and this software is far from perfect, but I want to make it available to anyone who might be interested in trying it out. Depending on interest expressed, new versions and improvements may appear.

2 Overview of the tools

This toolkit includes two programs written in C++, and several Matlab® scripts.

The primary tool is `gprscree`. This tool takes a set of sample points (design parameter bindings and experimentally determined performance scores) and uses them to create genetically programmed response surfaces. The core task of this tool is an evolutionary search through the space of equations to find one that accurately predicts performance. `gprscree` is a C++ implementation of the tool described in [1].

The `aedocree` tool generates a sample subset of possible designs using the Audze-Eglais Uniform Latin Hypercube Design of Experiments. The tool takes a list of parameters and their

possible values as input, and the output is the list of sample point parameter bindings which can in turn be fed to `gprscree`. The core task of this tool is an evolutionary search through the space of sample subsets to find one that maximizes the distance between the included sample points. `aedocreate` is a C++ implementation of the `binGen` tool described in [2].

Finally, I include several Matlab® scripts that were useful to me in evaluating the quality of the GPRSs created by `gprscree` and in using the surfaces to find the optimal designs in the space.

3 System requirements and compiling

This toolkit has been compiled and used on Intel-based Linux and OSX systems using GCC version 4.0 and higher. Other systems and C++ compilers may work. You will probably have to edit the simple Makefile to match your system configuration.

The only major external requirement is that `gprscree` uses the `levmar` library to perform unconstrained Levenberg-Marquardt optimization. This library is GPL'd and available at <http://www.ics.forth.gr/~lourakis/levmar/>, with a description of its functionality provided in [5]. The library in turn requires that your system have BLAS/ATLAS installed. Follow the directions available at the above URL to install `levmar` and then make sure the path to the library is included in the `gprscree` Makefile. This code has been tested with `levmar` version 2.1 and 2.2. If you are using a newer version, you may have to replace `gprscree/include/lm.h` with the file from the most recent version.

The source code for both tools is provided not only for open source distribution but also because it is almost certain you will have to modify certain parameters of the genetic algorithms in order to get the best results for your specific problem. Generally these parameters are defined in a specific `*param.h` file. See the configuration sections below to see what you might want to adjust before compiling or recompiling.

4 Audze-Eglais Design of Experiments Creator (`aedocreate`)

4.1 Configuration

To change these parameters you will have to edit them in `GAParams.h` and recompile `aedocreate`.

Detecting convergence. `FIT_IMPROVEMENT` and `IMPROVE_CUTOFF` control when the evolutionary search is determined to have converged. If the fitness of the best individual does not improve by more than `FIT_IMPROVEMENT` for more than `IMPROVE_CUTOFF` generations, the search is halted. You may have to adjust `FIT_IMPROVEMENT` to be meaningful depending on the sample size you are searching over and the dimensionality of your space.

Size and runtime. `POP_SIZE` and `NUM_GENERATIONS` impact the amount of exploration the algorithm is allowed to do, and naturally they thereby effect the runtime of the algorithm. `POP_SIZE` is the number of individual samples bred and evaluated within each generation (larger populations allow for more variability), while `NUM_GENERATIONS` provides a cap on the number of generations for which the search is allowed to continue. Populations smaller than 500 are not recommended.

Evolution. `KILL_PERCENT`, `ELITE_PERCENT`, and `MUTATION_PROB` govern the evolutionary aspect of the search algorithm. `KILL_PERCENT` specifies the portion of a generation’s individuals who are killed prior to being allowed to breed (the `KILL_PERCENT`% worst are removed). `ELITE_PERCENT` specifies the portion of a generation’s individuals who are copied over unchanged into the next generation’s population (the `ELITE_PERCENT`% best are copied). Increasing these percentages tends to speed the rate of convergence and the expense of further exploration. `MUTATION_PROB` governs the rate at which mutations occur in each generation; more mutations result in more diversity, slower convergence, and increased exploration. In this specific problem context, high mutation rates (greater than 1%) seem to be very beneficial.

4.2 Generating Input

Input for `aedocreate` is generated by the user. The user specifies the desired sample size; generally you should choose as large a sample as possible, given the time constraints of your study and the length of the highly detailed experiments or simulations you will perform. While the design space coverage provided by the Audze-Eglaiss formulation is robust in the face of small samples, the actual GPRS technique may require relatively large sample sizes (tens or hundreds of points) to converge.

The user then specifies a name and type for each design variable. There are four types:

Cardinal. A variable that can take on an integer value from a finite range. Specify the minimum, maximum and step for the range.

Real. A variable that can take on a real value from a finite range. Specify the minimum, maximum and step for the range.

Boolean. A binary variable that is either on or off.

Nominal. A set of mutually exclusive, non-quantifiable choices. Specify the number of possible choices.

4.3 Runtime Considerations

The metric by which candidate samples are evaluated is a distance-based metric that mimics a potential energy calculation for a set of masses exerting gravitational forces on each other. Currently, this evaluation is $O(n^2)$ in the number of points included in the sample. This can have a significant impact on the time it takes to generate a final sample. Another factor is the speed at which the algorithm converges on a solution. In general it is preferable to encourage longer explorations, as they tend to produce more globally optimal solutions.

4.4 Interpreting Output

The output of `aedocreate` is a list of sample points, one per line. For each sample point, the design variables are bound to specific values, and these values are listed on the sample point’s line. For each point, the user should configure their simulation or experiment accordingly, and then run it through and collect any pertinent performance information.

It is important to note that single nominal design variables are converted into multiple boolean variables according to a ‘one-hot’ encoding scheme. In other words, each possible choice for the

nominal variable is assigned to a boolean variable, and only one of the choice boolean variables can be true at a time. This conversion is done to allow the genetic programming algorithm to determine the significant of certain choices without having to assign them a numeric value. It is left to the user to determine which binary variable's activation corresponds with which choice

As discussed below, the input to the `gprscree` tool is this same list of sample point variable bindings, concatenated with the collected performance data.

5 Genetically Programmed Response Surface Creator

The algorithm used to create genetically programmed response surfaces is discussed in great detail in [3] and [1].

5.1 Configuration

To change these parameters you will have to edit them in `GPparams.h` and recompile `gprscree`. There are quite a large number of configuration variables that govern the operation and scope of the evolutionary search through the space of possible response surfaces.

Search space size. `NUM_DESIGN_VARS` is the number of unique design variables that define the design space. Remember to count single nominal variables as multiple distinct binary variables as per the output of `aedocreate`. `NUM_FULL_SIM_DESIGN_POINTS` is the size of the sample subset for which the user collected real data.

Validation. In order to prevent overfitting, it is possible to hold back some measurements taken as part of the sample set and use them to measure the accuracy of the current population of response surfaces. If the surface ever grows worse at predicting the values of these points, it is likely becoming overfit to the rest of the sample data. `NUM_USED_FOR_VALIDATION` specifies the number of entries in the sample set that will be withheld for validation purposes. `GP_VALIDATION_CUTOFFCOUNT` specifies the number of generations where validation measurements are allowed to grow worse before the surface is judged to be overfit. Generally, using this capability is advised when proper values for `GP_EXCESSIVE_LENGTH_CONSTANT` are unknown (see below).

Expression operators. `NUM_USED_BINARY_OPS`, `NUM_BINARY_OPS`, `NUM_USED_UNARY_OPS`, and `NUM_UNARY_OPS` define the number and type of operators which can be used in the response surface equations. Adding new operators requires modifying the source file `ExpTree.h` and any functions which evaluate response surface expression trees. You can choose to use only a subset of the defined operators using the `USED` variables; the first X variables in terms of the order they are defined in `ExpTree.h` will be used.

Tuning parameters. The genetic programming algorithm deterministically assigns tuning parameters to individual candidate response surface expressions, and tunes them to match the sample data as closely as possible before evaluating the expression. The Levenberg-Marquardt algorithm is used for this tuning process. However, its success is to a degree dependent on the initial values assigned to the parameters:

The simple approach is to seed the tuning parameters with random initial values before they are tuned. An alternative suggested by Alvarez [1] is to perform a quick genetic search over

the space of possible tuning parameter values, in an attempt to seed the L-M algorithm with reasonably good values. Both approaches are implemented in `gprcreate`, and the `GP_USE_GA_FOR_TUNING` controls which one is used (1 to use the values from the genetic search, 0 to use the random seeds). In my experience, including the GA search actually increases runtime while offering a statistically insignificant improvement in overall results. The user should feel free to experiment.

Maximum run time. `NUM_GP_GENERATIONS` and `NUM_GA_GENERATIONS` set the maximum search time for the outer genetic programming search through potential response surfaces and the inner genetic algorithm search through possible tuning parameter bindings.

Expression complexity. `MAX_NUM_TUNING` caps the number of tuning parameters that can be assigned at once (across the entire population), and this number should be increased for larger populations. `MAX_TREE_DEPTH` and `GP_EXCESSIVE_LENGTH_CONSTANT` control the complexity of individual expression trees: `MAX_TREE_DEPTH` caps the depth of the initial population of expression trees while `GP_EXCESSIVE_LENGTH_CONSTANT` penalizes excessively long expressions. `GP_EXCESSIVE_LENGTH_CONSTANT` is a very important setting, as it prevents the expressions from overfitting to the sample data. Users will likely have to experiment with their specific problem domain to determine an appropriate setting for this parameter. Smaller values allow longer expressions that more closely match the data. If you are using part of the sample as a validation set, a smaller value is preferable.

Evolution and convergence. The remaining `GA_*` and `GP_*` parameters govern the evolutionary algorithms that search for accurate response surfaces and within each surface search for the optimal tuning parameter values. The functionality governed by these algorithmic parameters mirrors that described in Section 4.1. Use them to govern the speed at which the searches converge on solutions.

5.2 Generating Input

Upon running the program, the user specifies input and output files.

Input to the `gprcreate` tool is in the same format as the output of the `aedocreate` tool, though the input file could also be created by hand if the user is using another method to sample the design space. The format is: one sample point per line, with each line containing the variable values of that sample point, and then the value of the performance metric measured for that point. Nominal variables with N choices should be transformed into N binary variables with a ‘one-hot’ encoding.

5.3 Runtime Considerations

There is an inherent tradeoff between faster convergence and increased accuracy, such that searches that are allowed to run longer are more likely to produce better results. This tradeoff is somewhat problem dependent, as it will generally take longer to create surfaces that accurately predict performance metrics with more complicated, non-linear behavior. Larger populations naturally result in longer run times because more individuals must be evaluated in each generation. The time it takes to evaluate an individual increases with the complexity of the expressions and especially with the sample size of the training data. However, the runtime is not directly dependent on the number of dimensions/design variables included in the problem.

5.4 Interpreting Output

At each generation, if the best individual response surface of the generation is more fit than any previous response surfaces then the equation representing that response surface is recorded in the output file. This means that runs may record useful results even if they have to be aborted or if the final surface is deemed to have overfit the sample data. Fitness and the time since the last significant improvement are also reported at each generation.

Response surfaces are output as in-order expression trees. Variables are identified by their index, i.e. the order they appear in the input file. So “x0” is the first variable listed on each line of the input file, “x1” is the second, and so on.

Some variables may not appear in the response surface expressions at all. This implies that the response surfaces were not any more accurate when they included this design variable, and so in favoring shorter expressions the evolutionary process eventually stripped them from the population. If a variable has been removed, than it is not useful in predicting the performance metric in question (i.e. it has a negligible impact on that metric). This explicit information is extremely useful and one of the primary advantages of using GPRS over similar artificial neural network-based methodologies.

6 Matlab® Scripts

Several Matlab® scripts are included in the package that give examples of how to evaluate the response surfaces created by `gprscree`. These scripts should serve as examples or starting points, rather than finished products: the user will have to adjust each script to match the number of parameters present in their design space and place the equation to be evaluated in the `CALC` function. A user without access to Matlab® could undoubtedly port them to the open-source alternative **R** if necessary.

`GPRSeval.m` calculates the mean and standard deviation of the percentage error between real data and the values predicted by the response surface. `GPRSevalforCDF.m` puts the difference between the predicted and real data in a matrix so that Matlab® can build a CDF out of it. `plotPredVsReal.m` plots values of both real and predicted values. These scripts all show the accuracy of the GPRS in matching the sample data, or predicting non-sampled data collected by the user.

Matlab® can also be used to relatively swiftly analytically evaluate every possible design point, and in turn select the best ones for further study. I also found Mathematica® to be a useful source of numerical methods for identifying the global optima of the expression. For both programs, the output expression from `gprstool` can be copied and used verbatim. However, lengthy expressions may exceed Matlab®’s frustrating cap on nested parenthesis, requiring some manual editing.

7 Future Improvements

Two areas stick out in my mind as areas for potential improvement.

N-fold cross validation. A more robust method for determining when the response surface expression is beginning to overfit the data is to withhold a percentage of the sample data from the true training set and use it as a proxy for the expression’s accuracy on all non-sample data. Even better is to simultaneously create multiple response surfaces out of the different

possible subsets of the sample data and to use an average of their predictions as the true prediction. If $X\%$ of the sample set is used for validation, then $1/X$ folds should be made, and a response surface constructed for each. This methodology was used in [4] to prevent the neural networks from overfitting the sample data.

Implementing this functionality in `gprcreate` will require an added outer loop to subdivide the sample set into multiple subsets and create multiple response surfaces on top of them. Ideally, these surfaces should be created in parallel. Even so, using N -fold cross validation will require N times as much work, so evaluations should be done to demonstrate that the improvement in global accuracy is worthwhile.

Parallelization. The genetic algorithms used in both tools are inherently parallel during many phases of their execution. Each member of the population should ideally be created, tuned, evaluated, ordered and bred in parallel. In a sequential implementation, creation, ordering and breeding occupy small percentages of runtime effort, which is dominated by tuning and evaluation. So when I attempted to add parallelism to the `gprcreate` code by means of `pthread` I focused on tuning and evaluation.

Unfortunately, the `levmar` library used to provide the final tuning parameter values does not appear to be threadsafe. Thus, all individual tuning threads lock on the call to this library. Hopefully, a future version of this library or another open source alternative will be more conducive to parallelization efforts.

`aedocreate` is currently implemented in an entirely sequential fashion. For the problem domains and sample sizes I was working with it was not worth the effort to reduce runtime by adding `pthread` support. Such support should be relatively easy to add if the code in `gprcreate` is used as an example. `aedocreate` does not use the unsafe library, so there should be no limitations on the effectiveness of a parallel implementation.

References

- [1] L. Alvarez. *Design Optimization based on Genetic Programming*. PhD thesis, University of Bradford, 2000.
- [2] S. J. Bates, J. Sienz, and D. S. Langley. Formulation of the audze–eglais uniform latin hypercube design of experiments. *Adv. Eng. Softw.*, 34(8):493–506, 2003.
- [3] H. Cook and K. Skadron. Predictive design space exploration using genetically programmed response surfaces. In *45th ACM/IEEE Conference on Design Automation (DAC)*, June 2008.
- [4] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 195–206, New York, NY, USA, 2006. ACM.
- [5] M. Lourakis. `levmar`: Levenberg-marquardt nonlinear least squares algorithms in C/C++. <http://www.ics.forth.gr/~lourakis/levmar/>, 2004.