User Manual

Scott Pratt, Eren Erdogan, Ekaksh Kataria

Contents

1 Overview

This manual describes how to install and run the Smooth Emulator software. The software performs three basic functions. First, the Simplex sampler simply chooses a set of points in model parameter space, at which full model runs will be performed to then tune the emulator. The user must provide a description of the model parameters and the prior in a text files in a standard format. There are several options, the first of which is to choose the points that represent a simplex, e.g. an equilateral triangle in two dimensions or a tetrahedron in three dimensions. In a simplex, all points are equi-distant from one another. The number of training points is then determined as well. In addition to the standard simplex, there are additional options which are motivated by the simplex form. For the simplex form, if there are $\boldsymbol{N_p}$ model parameters, there are $\boldsymbol{N_p + 1}$ training points in the simplex, which matches the number of points needed to determine a linear fit. Another choice, which is based on the simplex chooses enough points to determine a quadratic fit, $\boldsymbol{(N_p + 1)(N_p + 2)/2}$. The software will write the information about the training points in a standard format, which is described in the manual.

If the user decides to use training points from a different procedure, the user can still write the information about the points in a different format, and the emulator tuning will still work, as the emulator itself is not predicated on a specific choice of training points. The user is then responsible for running the full model at the training points and expressing observables for each training points in a standard format. The manual describes the output format.

The second functionality of the software is to build and tune the emulator. Once the observables are calculated at the training points, the emulator will be built and tuned. User input mainly consists of altering a text file of parameters, which provide the user several choices in how the emulator functions. After being trained, the Taylor coefficients representing the emulator are written to a file. One can always add additional training points, and retrain the emulator.

The third functionality of the software is to perform a MCMC exploration of parameter space using the emulator. This user must express the experimental observables and their uncertainties in a standard format. The MCMC software will read the emulator coefficients from file and perform the MCMC exploration. This procedure is also guided by a simple text file of parameters. The MCMC software uses python and Matplotlib to generate plots that describe the posterior.

2 Getting Started

2.1 Installation

Smooth Emulator software should run on UNIX, Mac OS or Linux, but is not supported for Windows OS.

Step 0 : Prerequisites

Smooth Emulator is largely a C++ project. In addition to a C++ compiler, the user needs the following software installed.

- git

- CMake
- Eigen3 (Linear Algebra Package)
- GSL (Gnu Scientific Library)
- Python/Matplotlib (only for generating plots in the MCMC procedure)

CMake is an open-source, cross-platform build system that helps automate the process of building, testing, and packaging software projects. It provides a simple and efficient way to generate platform-specific build files from a platform-independent configuration. Hopefully, CMake will perform the needed gymnastics to find the Eigen3 and GSL installations. To install CMake, either visit the CMake website (https://cmake.org/), or use the package manager.

Eigen is a C++ template library for vector and matrix math, i.e. linear algebra. The user can visit the Eigen website (https://eigen.tuxfamily.org/dox/), or use their system's package manager.

The GNU Scientific Library (GSL) is a numerical library for C and C++ programmers. The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite. One can either download the software from the GSL website (https://www.gnu.org/software/gsl/), or use a package manager.

Step 1 : Making Home Directory and Setting Home Environment Variable

The software requires two repositories. They should be cloned into the same directory. For compilation purposes this directory needs to be accessed via an environmental directory. For example the directory might be named `githome_msu` (another name would be fine, just substitute it everywhere below). First, create the directory, then enter that directory.

```
% mkdir (some path)/githome_msu
% cd (some path)/githome_msu
```

Then, create an environmental variable which describes the location of the directory above.

```
% export GITHOME_MSU=(some path)/githome_msu
```

It is recommended to copy this command into the user's .bashrc (or equivalent) file to avoid re-enerting it each time one needs to recompile.

Step 2 : Downloading and Compiling

```
% git clone https://github.com/scottedwardpratt/smooth.git
% git clone https://github.com/scottedwardpratt/commonutils.git
```

This creates 2 directories: `.../msu_git/commonutils` and `.../msu_git/smooth`. Next, create the Make files with cmake, then compile them.

```
% cd ../githome_msu/commonutils/software
% cmake .
% cd ../githome_msu/smooth/software
% cmake .
% cd ../githome_msu/smooth/program
% cmake .
% make simplex
% make smoothy_writecoefficients
% make smoothy_readcoefficients
```

As described further below, the foundational software and libraries are in the `githome_msu/commonutils/software` and `githome_msu/smooth/software` directories. The `githome_msu/smooth/program` directory has the short programs which call functions from the other directories. This arrangement should more easily allow the reader to copy the codes in `programs` directory to make their own versions.

Step 3 : Making a Project Directory

```
% cp -r ../template myproject
```

After compiling the communities and smooth directory in the githome_msu, create the direcotry for the user project by copping the template directory. This will set up the directory structure for use.

Step 4 : CMake

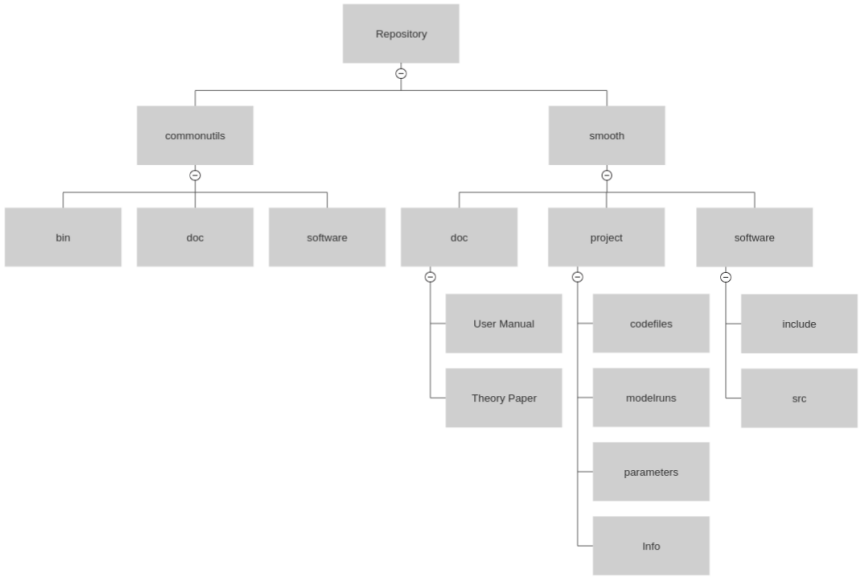
In the project directory do the following command to compile and make the executables in the project directory

```
% cmake .
% make
```

This will create the executables with are mentioned below the structure code with their functionality and

2.2 Structure

The directory structure of the repository is as follows:



2.2.1 Bin Directory

This directory contains the codes with the .cc files that can be referenced to understand the executables.

Template Project Directory 1. simplex.cc This cc helps in choosing the training points and making the model parameters and creates the model run directory using the prior info for the simplex

2. fakemodel.cc The fakemodel.cc is a template used to represent a potential model. It will be replaced by the actual model created by the user.

3. smoothy_writecoefficients.cc This cc files read in the parameter files and reads in the training point info from the simplex and tune the function Y values and generate the coefficient for each observable and make a directory of the possible coefficient values

4. smoothy_readcoefficients.cc This cc file reads in the coefficient files and reads in the training info from the simplex and tests the samples from the emulator at the training points.

2.2.2 Software Directory

The software file contains all the classes and the main functionality of the emulator. The Software file contains the two main directories, the Include directory and the source directory called src.

2.2.3 Include Directory

This directory contains all the header files. The header files contain a set of predefined library functions and newly defined classes for the different functions used in the emulator code.

2.2.4 Source Directory (Src)

Within this directory, you will find the core functions of the emulator in the form of .cc files. Each step of the process has its own class with distinct constructors.

2.2.5 Project Directory

The project directory is where the emulator works. There are files in this directory:

1. Info Directory

The Directory includes the parameter and observable file. The prior info file contains the list of parameters and their maximum and minimum values in the following example structure.

Parameter Name	Distribution Type	Min Value	Max Value
par1	gaussian	0	100
par2	linear	0	37
par3	uniform	0	56
⋮	⋮	⋮	⋮

Table 1: `prior_info.txt`

Observable Name	Unit	Value
length	meters	0.3
mass	kg	100
time	seconds	20
⋮	⋮	⋮

Table 2: `observable_info.txt`

2. Parameter Directory

The Emulator Parameter is the file user changes in order to use and optimize the code according to the model the user has and the output user is expecting. The parameters described below can be found in the parameter directory and in the file emulator_parameter.txt. The functionality is mentioned in the Appendix A. These files are structured in the following manner.

Parameter Name	Value
par1	1
par2	somePath
par3	true
⋮	⋮

Table 3: `mod_parameters.txt`

3. Modelruns Directory

The Modelrun Directory includes the parameters and observables provided by the user in each run directory. It has two files: one for the model parameters with the parameter name and value, and another for the observable with its name, value, and uncertainty. These files are structured in the following manner.

Parameter Name	Value
par1	96.472
par2	17.244
par3	29.456
⋮	⋮

Table 4: `mod_parameters.txt`

Parameter Name	Value	Uncertainty
length	109.384	7.234
mass	58.34	2.59
time	15.23	0.97
⋮	⋮	⋮

Table 5: `obs.txt`

4. Cmake files

This contains the Cmake files and their compilation have been mentioned in the section above 2.1

2.3 Usage

The user is required to have some knowledge of C++ in order to write a routine program that calls methods from the software directory for their needs. Some example codes can be viewed in the template directory.

3 Simplex

3.1 Summary

From the motivation, we can interpret that the purpose of the simplex is to produce training points. The emulator will use these training points to tune itself, and output results. Thus, it is important that one takes their time, and chooses how they want to generate their training points carefully.

The user first needs to make sure that they follow the instructions under Section 2.3. After this, one should decide on how they want to generate their training points. The code provides the user with different options, but one can choose to generate the training points in a different way as well.

3.2 Parameters

To access the simplex parameter file one should go to their project directory and follow the steps listed below:

```
% cd parameters
% ls
```

After these commands, one should see a `simplex_parameters.txt` file. The user should open this file with their preferred editor. The user should see three different parameters with different input variables. The parameters are listed below:

3.2.1 TrainType

This parameter determines the training type that the simplex will use. The default type is 1. All the types have different uses, pros, and cons. The differences between types are explained in the next section.

3.2.2 RTrain

This parameter determines the distance from the origin that the training points will be selected throughout the parameter space.

3.2.3 ModelRunDirName

This parameters is used to determine the path file that the run files will be created in. The default name is `modelruns`, but the user can change this to anything they want.

3.3 Training Types

3.3.1 Type 1

Depending on the number of parameters, the program creates a simplex in $\boldsymbol{N}$ dimensions. This simplex's vertices will be used to generate new training points. These points will be scaled by different values so the training points aren't in the same radius. This results in the minimum number of required points for linear fits. Thus, the user might want to select this training type if their model is a linear one.

3.3.2 Type 2

Depending on the number of parameters, the program creates a simplex in $\boldsymbol{N}$ dimensions. This simplex's vertices will be used to generate new training points there and along the edges. These points will be scaled to be in different radii from the center. This results in the minimum number of required points for quadratic fits. Thus, the user might want to select this training type if their model is a quadratic one.

3.3.3 Type 3

This training type creates two different simplexes to put them on top of each other in a reflected way. The 2-dimensional visualization of this would look like the "Star of David". The points created by the crossed edges are used as training points for the emulation.

3.3.4 Type 4

3.4 Training Point Generation

Simplex is a mathematical term that generalizes the notion of a triangle or a tetrahedron to arbitrary dimensions. The simplex method uses the vertices of simplexes to get training points at different radii along the edges. This method also is computationally cheap due to its iterative process.

To use the simplex method, use the following command in your project directory:

```
% simplextest parameters/simplex_parameters.txt
```

This code is also responsible for generating the directory that the run files are stored. The stored values will be the values that are used in the emulator.

4 Tuning the Emulator

4.1 Summary

From the motivation, we can interpret that the purpose of the emulator is to reproduce the model reasonably away from the training points and the goal is to focus on a particular class of functions that are smooth.

The emulator has a specific functional form with numerous coefficients denoted as $\boldsymbol{A}_{\vec{n}}$. The theory of steps illustrated on how to constrain the emulator is mentioned in the theory section below. After Running the simplex.cc file we output the model_par.txt files for each run using the prior information using the number of parameters sent from the model_parameter.txt file from the info directory and use the simplex to create the training points.

4.2 Code

The Code uses a fake model which acts as a template used to represent a potential model. It will be replaced by the actual model created by the user. The file reads in the model prior info and the observable info files from the info directory and generates the observable text files in the run directory.

To use the fake model run the following command in the terminal window in the project directory.

```
% fakemodel
```

4.3 Writing Coefficients

In the terminal window run the following command after the simplex and importing the model.

```
% smoothy parameters/emulator_parameters.txt
```

This code is responsible for invoking the primary functionality that adjusts the training points for the model. It produces samples of coefficients, which are then arranged in a directory within the project directory that provides details about each observable and various coefficient values.

This function aims to perform a Markov Chain Monte Carlo (MCMC) parameter tuning process to optimize some coefficients of a smooth emulator. The code attempts to find the best set of coefficients that maximize the log probability of the emulator given some training data.

Final Output and Results: After completing the specified number of iterations, the algorithm calculates the success percentage (proportion of successful updates), the final value of SigmaA, and the log-likelihood normalized by the number of degrees of freedom (Ndof) for the best parameter set (BestLogP/Ndof).

Interpreting the Results: The success percentage gives an indication of the acceptance rate of new parameter sets during the MCMC process. A higher success rate generally indicates efficient parameter tuning. The value of SigmaA represents the estimated uncertainty or spread in the parameter space. The BestLogP/Ndof provides a measure of the goodness of fit achieved by the best parameter set.

The function also updates statistics for the sampled variance (SigmaA) to calculate the average variance (SigmaAbar) over all the generated samples. In summary, It iteratively tunes the coefficients using MCMC and stores the optimized coefficient samples in a matrix for further analysis. It is used for the process of generating and analyzing samples for the smooth emulator.

The Code also Logs the comparison between the predicted value and the actual training data value for the current observable. Also, log the corresponding uncertainty. The function evaluates the smooth emulator's accuracy by generating predictions for each observable at the training points and comparing these predictions with the actual training data.

The code also writes metadata about the number of parameters (NPars), maximum rank (MaxRank), and total number of coefficients (NCoefficients) to a file named "meta.txt" in the created directory. For each sample generated using the function, it writes the corresponding coefficients (parameters) to a separate file named "sampleX.txt" in the created directory, where "X" represents the sample index. The number of coefficients is equal to NCoefficients, and they are written in a column-wise format.

5 MCMC

6 Troubleshooting

7 Theory

Constraining the emulator

1. $\boldsymbol{M}$ full model runs are conducted at different positions $\vec{\boldsymbol{\theta}}_{\boldsymbol{m=1,M}}$. These runs provide corresponding values $\boldsymbol{F}_{\boldsymbol{m=1,M}}$.

2. The functional form of the emulator has a large number of coefficients $\boldsymbol{A}_{\vec{\boldsymbol{n}}}$, which exceeds the number of training points M. However, the dependence of the coefficient A is purely linear.

3. The coefficients are enumerated as $\boldsymbol{A}_{\boldsymbol{c}}$ with $\boldsymbol{c} = \boldsymbol{1} \cdots \boldsymbol{C}$, where $\boldsymbol{C} > \boldsymbol{M}$.

4. Random values are assigned to the coefficients $\boldsymbol{A}_{\boldsymbol{M}}$ through $\boldsymbol{A}_{\boldsymbol{C}}$, excluding the first $\boldsymbol{M}$ coefficients.

5. A set of linear equations is solved to determine the values of the first $\boldsymbol{M}$ coefficients. This process involves finding the coefficients that best fit the training points.

6. A weight is applied to the values of $\boldsymbol{A}$, taking into account their consistency with the prior likelihood of $\boldsymbol{A}$ and the given constraints. This step helps to incorporate prior knowledge and constraints into the emulator.

7. A representative set of A is generated, typically consisting of a dozen samples. Each sample function passes through all the training points but may deviate further from them.

8. By averaging over $\boldsymbol{N}_{\mathbf{sample}}$ sets of coefficients, a prediction for the emulator can be made at a specific point $\boldsymbol{\theta}$. The $\boldsymbol{N}_{\mathbf{sample}}$ points provide an estimate, and their variance represents the uncertainty associated with the emulator's prediction.

A Emulator Parameters

The Emulator Parameter is the file user changes in order to use and optimize the code according to the model the user has and the output user is expecting. The parameters described below can be found in the parameter directory and in the file emulator_parameter.txt.

It is organized in the following way

The parameter map has a prefix for each of the parameters SmoothEmulator and in the appendix, they are mentioned directly with the values in front.

A.1 NTrainingPts

The number of training points determines how many data points are available for training the smooth emulator. It affects the number of training samples used in various calculations and tuning steps, and ultimately influences the accuracy and generalization ability of the emulator model.

A larger number will lead to a more accurate emulator, but it could also increase computational costs. The choice of the number depends on the specific application and the trade-off between accuracy and efficiency.

A.2 SigmaYMin

This value is used to set a lower limit for SigmaY, it might be used to prevent the emulator from overfitting or producing unrealistic predictions by penalizing or restricting very small uncertainties in the observed data.

A.3 NMC

This parameter specifies the number of iterations (steps) to be performed in the MCMC algorithm which is used for tuning the emulator parameters to find the optimal values that maximize the likelihood of the observed data. A larger value of NMC allows for a more thorough exploration of the parameter space, potentially leading to more accurate tuning results.

However, it also increases the computational cost of the tuning process since each iteration requires calculating the emulator output and evaluating the likelihood.

The choice of NMC should be made carefully to strike a balance between the accuracy of the tuning results and the computational efficiency of the algorithm. The optimal value of NMC may depend on the complexity of the problem, the size of the training data, and the available computational resources.

A.4 NASample

This parameter represents the number of samples generated from the smooth emulator to estimate uncertainty in the emulator predictions. It controls the number of times the emulator coefficients (A) are sampled to obtain multiple emulator outputs.

The value of the NASample should be chosen based on the desired level of confidence in the emulator predictions and the computational resources available. Larger values of the NASample generally provide more reliable uncertainty estimates but also require more computational time to generate the samples.

A.5 MCStepSizeis

This parameter controls the step size of the Metropolis-Hastings Markov Chain Monte Carlo (MCMC) algorithm during the tuning process.

A larger value of MCStepSize means larger steps, which can lead to faster exploration of the parameter space but might also result in less precise tuning. Smaller MCStepSize leads to smaller steps, which may improve accuracy but may require more iterations to fully explore the parameter space.

A.6 MCSigmaAStepSize

This parameter controls the step size for updating the standard deviation SigmaA during the Metropolis-Hastings Markov Chain Monte Carlo (MCMC) algorithm's tuning process.

This enables the algorithm to simultaneously find the optimal values of A and SigmaA that best fit the training data and provide accurate emulator predictions.

A.7 TuneChooseMCMC

This boolean parameter that controls the choice of tuning method used in the smooth emulator. The choice between MCMC and "perfect" tuning methods can significantly impact the accuracy and computational efficiency of the emulator.

The MCMC method is more robust and can handle complex parameter spaces, but it requires a larger number of iterations (NMC) and can be computationally expensive. On the other hand, the "perfect" tuning method may be faster, but it may not fully explore the parameter space and may not find the global optimum.

A.8 UseSigmaYRreal

This is also a boolean parameter that determines whether the true (real) standard deviation (SigmaY) of the observed data is used during the parameter tuning process of the smooth emulator.

By considering the true standard deviation SigmaY in the likelihood calculation, the tuning process accounts for the uncertainty in the observed data. This can lead to more accurate parameter tuning and better emulator predictions, especially when the true standard deviation is known and informative about the variability in the observed data.

A.9 ConstrainA0

This boolean parameter controls whether the emulator coefficients $\boldsymbol{A_i}$ are constrained during the tuning process. Specifically, it determines whether the first coefficient $\boldsymbol{A_0}$ is fixed or allowed to vary during the tuning iterations.

For some problems, it may be known or desired that the first coefficient $\boldsymbol{A_0}$ should take a specific value based on physical constraints or prior knowledge. In such cases, setting the parameter to true ensures that the tuning process respects this constraint.

On the other hand, if there is no specific reason to constrain $\boldsymbol{A_0}$, setting the parameter to false allows the tuning algorithm to explore the full parameter space, including the possibility of $\boldsymbol{A_0}$ taking different values, which might lead to better-fitted emulator coefficients.

A.10 CutoffA

This parameter sets the extra width that keeps sigma A from drifting off to infinity.

A.11 LAMBDA

This parameter represents a regularization term used in the smooth emulator for tuning the model coefficients (A). The term "LAMBDA" is typically used to denote the smoothness parameter and it serves as a tuning knob to control the complexity of the smooth emulator model.

The Lamda helps control the trade-off between fitting the training data accurately and keeping the model complexity in check. A higher value of LAMBDA leads to a more regularized (smoother) model with simpler coefficients, while a lower value allows the model to adapt more to the training data, potentially leading to more complex and flexible coefficients. It is often determined through experimentation and fine-tuning to achieve the best balance between model accuracy and complexity.

A.12 ModelRunDirName

It is a string variable that stores the name of the directory where the smooth emulator stores the information related to a specific model run.

A.13 CoefficientsDirName

It is a string variable that stores the name of the directory where the smooth emulator stores the coefficients (parameters) obtained during the tuning process.

A.14 MAXRANK

It is a variable that represents the maximum rank allowed for the smooth emulator and it determines the number of basis functions used to represent the smooth functions during the modeling process.

A higher MAXRANK allows the emulator to capture more complex and flexible response surfaces, but it may require more data points and computational resources. Conversely, lower MAXRANK results in a simpler model with reduced flexibility but may be more computationally efficient and require fewer data points.

A.15 UseRFactor

It is a boolean parameter that determines whether the smooth emulator uses an "R-factor" regularization term during the tuning process. It is a measure of how well the smooth emulator reproduces the training data points.

By incorporating the R-factor into the tuning process, the emulator can optimize the coefficients (A) in a way that balances the fit to the training data with the overall smoothness of the model.