User Manual

Scott Pratt, Eren Erdogan, Ekaksh Kataria

Contents

1 Motivation

A prerequisite to the entire emulator is the assumption that the user has a basic understanding of C and C++.

2 Getting Started

2.1 Installation

For installing, one should do a git pull to the smooth emulator using the link and using commands git pull with the fooling documentation in the terminal by creating a directory in which one wants to work. An important note to mention is that the emulator works in UNIX, Mac OS or Linux and is not supported by Windows OS.

Step 0 : Prerequisites

Smooth Emulator uses some extra libraries that need to be installed on your computer and added to the PATH environment for it to work. These libraries are:

- CMake
- Eigen3
- GSL

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 (e.g., Makefiles, Visual Studio projects) from a platform-independent configuration.

Visit the CMake website (https://cmake.org/) to download it.

Eigen is a lightweight C++ template library for vector and matrix math, a.k.a. linear algebra. The program uses Eigen's matrix solvers to find solutions to certain problems.

Visit the Eigen website (https://eigen.tuxfamily.org/dox/) to download it.

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.

Visit the GSL website (https://www.gnu.org/software/gsl/) to download it.

Step 1 : Making Directory and Environment Variable

The first step is 2 fold, first start by creating a directory in the users computer when the user is in the .bash and calling it git_msu using the following command

```
mkdir githome_msu
```

Then start by creating an environmental variable which is done by adding the command in the Unix shell and for Mac-OS do the following commands and add it in the zsh's user configuration for interactive shells.

```
e .zshrc
```

After opening the .zshrc file add the following command

```
export GITHOME_MSU="/Users/'username'/git"
```

after saving the .zshrc file and restating the terminal one can make sure this works one can run the following command

```
echo ${GITHOME_MSU}
```

and it will output

```
"/Users/'username'/git"
```

where the username is going to be varied for different users.

Step 2 : Downloading

After creating the git directory take the terminal window and run the following command

```
git pull https://github.com/scottedwardpratt/
```

This downloads 2 directories: communities and smooth

Their functionality is mentioned in the structure section. After pulling one should get familiar with the environment created and the structure. This program runs on any C++ and c editors such as Atom, Visual Studio etc.

After downloading create a directory in the smooth directory such as projectX and using the git copy command, copy the items in the scottrun directory into the new directory using the following command when the terminal is in the project directory.

Step 3 : Making a Project Directory

```
cp -r ../template/model* .
```

This will set up the directory structure for use.

Step 4 : CMake

```
cmake /path/to/project
```

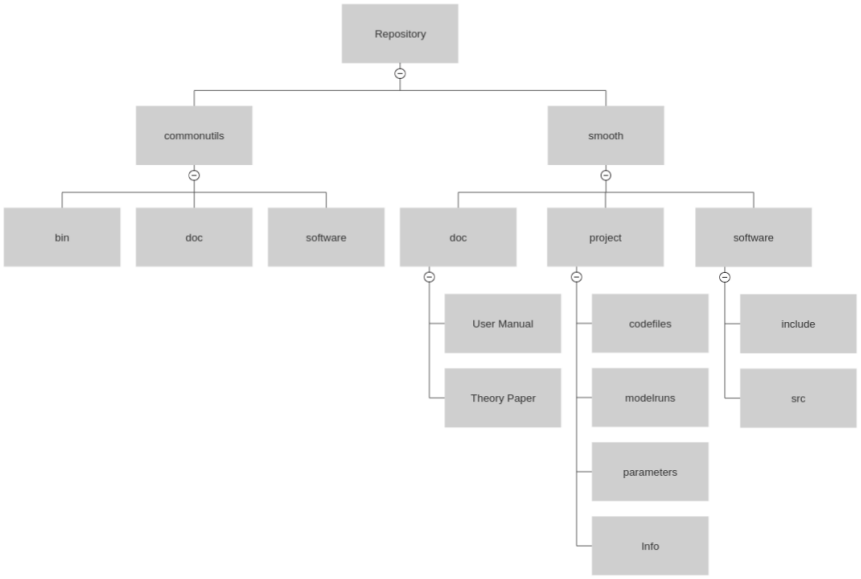
The user should also be informed about CMake.

Once the installation is complete, open a terminal or command prompt and navigate to the directory where your project's CMakeLists.txt file is located.

Run the cmake command followed by the path to the project directory. For example:

2.2 Structure

The directory structure of the repository is as follows:



2.2.1 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.

2.2.2 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.

2.2.3 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. These .cc files reference the corresponding .h file using the

"#include" feature and then define each class with unique initial values, methods, and constructors.

2.2.4 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 with SigmaA values for referencing the expected values of added parameters and observables by the user. These values represent the standard deviation of the Gaussian distribution that is used to sample the emulator coefficients A during the tuning process.

The value of SigmaA controls the step size of the random walk in the parameter space during the MCMC sampling. A larger SigmaA means larger jumps in the parameter space, leading to faster exploration of the space but possibly sacrificing the accuracy of the tuning process. Conversely, a smaller SigmaA leads to smaller steps, which may improve accuracy but may also slow down convergence.

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.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.

The order in how the code works is mentioned in the next section

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 Training Types

The user can change the training type via the parameter map file. To learn how, please refer to the appendix.

3.2.1 Type 1

This type makes sure the training points are generated

3.2.2 Type 2

3.2.3 Type 3

3.2.4 Type 4

3.3 Training Point Generation

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

For training the emulator we create the coefficient samples using the smoothy _writecoefficients.cc. The Functionality, how to use and the output of these files are explained below.

The fakemodel.cc is 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.

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 generates coefficient samples, which are then organized in a coefficient directory within the project directory. The directory contains information about each observable and samples of various coefficient values. The following constructors are used to accomplish this task:

Step 1 : Reading the Training Information

```
master.ReadTrainingInfo();
```

This is called from the trainnginfo.cc file and takes in the run directory name and reads in the values from each run directory and stores the values of the observables and the parameters values and stores in the current code.

Step 2 : Tuning in all functions(Y).

```
master.TuneAllY();
```

The purpose of this function is 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.

1. Adaptive Step Size:

The algorithm employs an adaptive step size (stepsize) for each parameter during the trial update process. The step size depends on SigmaA, MCStepSize, and a constant LAMBDA.

2. Acceptance Criteria:

If the trial log-likelihood (logPTrial) is higher than the current log-likelihood (logP), the trial parameter set is accepted as the new current parameter set, and the algorithm moves to the next iteration.

If the trial log-likelihood is lower, the algorithm applies the Metropolis-Hastings acceptance criterion: the trial parameter set is accepted with a probability of exp(dlp) (where dlp = logPTrial - logP). The algorithm also keeps track of the best log-likelihood encountered so far (BestLogP).

3. Parameter Updates and Success Count:

Successful updates are tracked using the success counter, which increments every time a trial parameter set is accepted (either unconditionally or based on the acceptance criterion).

4. 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).

These results are displayed in the console using CLog::Info().

5. 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.

6. Optimization and Iterations: For better parameter tuning results, you may need to experiment with different values of MCStepSize, MCSigmaAStepSize, and LAMBDA and Increase the number of iterations (NMC).

Overall, this code performs an MCMC parameter tuning process for the coefficients of a smooth emulator, attempting to find the optimal set of coefficients that best match the training data. The success rate, final variance (SigmaA), and log probabilities are reported for evaluation purposes.

Step 3 : Generating the coefficient (A) samples

```
master.GenerateCoefficientSamples()
```

This function is responsible for generating multiple samples of coefficients (A) for a smooth emulator using the MCMC tuning process. The function also updates statistics for the sampled variance (SigmaA) to calculate the average variance (SigmaAbar) over all the generated samples.

In summary, GenerateASamples 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.

Step 4: Testing the Coefficients At Training Points

```
master.TestAtTrainingPts();
```

This function is used to assess the accuracy of a smooth emulator by comparing its predictions with the actual training data at each training point.

It 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.

Step 5: Writing Coefficients for functions.

```
master.WriteCoefficientsAllY();
```

This function is responsible for writing the coefficients of a smooth emulator to separate text files on the disk.

1. Writing Metadata:

The function 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.

2. Writing Coefficients for Each Sample:

For each sample generated using the GenerateASamples() function, the function writes the corresponding coefficients (parameters) to a separate file named "sampleX.txt" in the created directory, where "X" represents the sample index.

Each file contains the coefficients for the smoothing emulator. The number of coefficients is equal to smooth-¿NCoefficients, and they are written in a column-wise format. This creates a directory containing the coefficient files and a metadata file for the observable. The directory name is specified by smoothmaster $\rightarrow$ CoefficientsDirName

In summary, the 'WriteCoefficients' function saves the coefficients of a smooth emulator to separate text files. It organizes the files in directories based on observable names, writes metadata, and stores the coefficients for each sample in a formatted manner.

5 Performing the Full Model Runs

6 MCMC

7 Troubleshooting

8 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 $\boldsymbol{c} = \boldsymbol{1 \cdots C}$, 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 Parameter Map

The parameter map 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 parameter map has a prefix for each of the parameters SmoothEmulator and in the appendix they are mentioned direcotry

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.

A.16 TrainType

Determines the training type

A.17 RTrain