Smooth Emulator & *Simplex Sampler* User Manual

Scott Pratt, Eren Erdogan, Ekaksh Kataria

Department of Physics and Facility for Rare Isotope Beams

Michigan State University, East Lansing Michigan, 48824

September 20, 2023

Contents

1 Overview

This manual describes how to install and run *Smooth Emulator* software. The software performs three basic functions. First, the *Simplex Sampler* chooses a set of points in model parameter space, at which full model runs will 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 equidistant from one another, and the number of training points is $\boldsymbol{N_p + 1}$, where $\boldsymbol{N_p}$ is the number of parameters. In addition to the standard simplex, there are additional options which are motivated by the simplex form. For the standard form the $\boldsymbol{N_p + 1}$ training points in the simplex match 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 record the information about the points in a same 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, and the uncertainties, for each training point in a standard format. The manual describes the output format.

The second functionality of the software is to build and tune the emulator, referred here as *Smooth Emulator*. The emulator reads the information above, along with another user-provided parameter file to choose which observables are to be emulated, which parameters will be varied, and which emulator options will be applied. 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 Installation and Getting Started

2.1 Prerequisites

Smooth Emulator software should run on UNIX, Mac OS or Linux, but is not supported for Windows OS. *Smooth Emulator* is largely written in C++. 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 compiling and linking for software projects. 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 system's package manager for the specific system. For example, on Mac OS, if one uses *homebrew* as a package manager, the command is

```
% brew install cmake
```

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. For example on Mac OS with *homeebrew*,

```
% brew install eigen
```

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. Again, for *homebrew* on Mac OS,

```
% brew install gsl
```

2.2 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 variable. For example the directory might be named `/Users/CarlosSmith/git_msu`. The User needs to set an environmental variable, `GITHOME_MSU`, to the full path of the directory, e.g.

```
% export GITHOME_MSU="/Users/CarlosSmith/git_msu"
```

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

2.3 Downloading

From within the `GITHOME_MSU/` directory,

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

This creates 2 directories: `.../GITHOME_MSU/commonutils` and `.../GITHOME_MSU/smooth`.

The User needs to create a project directory from which the User would perform most projects. This is easiest accomplished by copying a template from the *Smooth* distribution,

```
% cp -r GITHOME_MSU/templates/myproject MY_PROJECT
```

Hence forth, `MY_PROJECT` will refer to the directory, including the path, from which the User will perform most of the analysis. The User may wish to have several such directories. These directories should be outside the `smooth/` or `commonutils` paths.

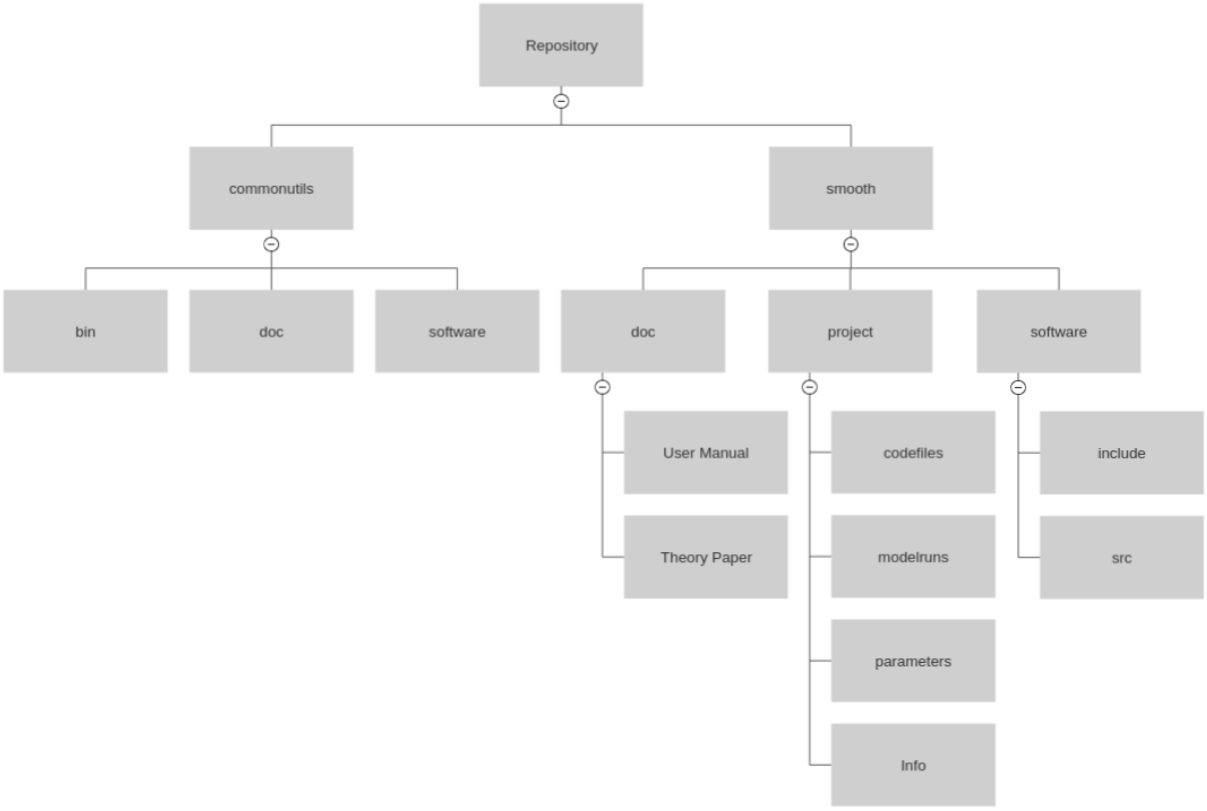
Although the main source code, include files and libraries are all located in the software directory, the main programs and executables are not. The motivation for this decision is to allow the User to easily modify their own versions of the main programs. These tend to be very short programs. For that reason their is a separate directory to store the main programs and their executables. The User can easily set this up by copying a template directory,

```
% cp -r GITHOME_MSU/templates/mylocal MY_LOCAL
```

Here `MY_LOCAL` will hence forth refer to the full path of this directory. This directory should be outside the `smooth/` or `commonutils` paths.

2.4 Directory and File Structure

The directory structure of the repository is as follows:



Once compiled, the libraries in the `commonutils/` directory are used for a variety of tasks. These libaries are not particularly designed for *Smooth Emulator* or *Simplex Sampler*. The `smooth/` directory contains codes that are used to create libraries specific to the sampler and emulator. The Executables are stored in `smooth/local/bin`. The short main program source files are located in `smooth/local/main_programs/`. It is not envisioned that the User would edit files in the `smooth/software/` directory, but that the User may well wish to create custom versions of the short main programs in `smooth/local/main_programs/`.

2.5 Compiling Libraries

First, change into software directories, then create the makefiles with cmake, then compile them.

```
% cd GITHOME_MSU/commonutils/software
GITHOME_MSU/commonutils/software% cmake .
GITHOME_MSU/commonutils/software% make
GITHOME_MSU/commonutils/software% cd ../GITHOME_MSU/smooth/software
GITHOME_MSU/smooth/software% cmake .
GITHOME_MSU/smooth/software% make
```

`At this point all the libraries are built, but this does not include the main programs. The main programs, are short, and are located in a separate location, as they are meant to serve as examples which the User might copy and edit at will.`

Finally, compile the main programs. Below, this illustrates how to build the programs used for generating training points with Simplex and for tuning the emulator with Smoothy:

```
% cd MY_LOCAL/build
MY_LOCAL/build% cmake .
MY_LOCAL/build% make simplex
MY_LOCAL/build% make smoothy_tune
.
.
```

Other source codes for main programs can be found in `MY_LOCAL/main_programs/`. If you build your own main programs (probably using these as examples), you can edit the `CMakeList.txt` file in `GITHOME_MSU/smooth/local/build`, using the existing entries as an example. The executables should appear in `MY_LOCAL/bin/`.

2.6 The Project Directory

Within `MY_PROJECT/` there are three sub-directories (assuming it was created from the template). The first is `MY_PROJECT/Info/`. Information about the model parameters, and their priors is stored in `MY_PROJECT/prior_info.txt`, and information about the observables is store in `MY_PROJECT/observable_in` The `MY_PROJECT/parameters` directory stores user-defined parameter files used by *Simplex Sampler*, `MY_PROJECT/parameters/simplex_parameters.txt`, and by *Smooth Emulator* `MY_PROJECT/parameters/emul` The `MY_PROJECT/modelruns` directory will store information for each full-model run. The directories `MY_PROJECT/modelruns/run0/`, `MY_PROJECT/modelruns/run1/`, $\cdots$, have files describing the model parameters for each run, along with the output required by the emulator for each specific full-model run. For example, the `MY_PROJECT/modelruns/run1/` directory has the files `mod_parameters.txt` and `obs.txt`. The first file stores the model parameter values for that particular training run. The User then runs their full model based on those parameters and stores the corresponding observables in `obs.txt`. The User may generate the `mod_parameters.txt` files using *Simplex Sampler*, or the user might generate them according to some other prescription. Once the User has then generated the `obs.txt` files, *Smooth Emulator* can then build and tune the emulator.

3 Generating Training Points with *Simplex Sampler*

3.1 Summary

Simplex Sampler produces a list of points in model-parameter space to be used for training an emulator. The algorithms are based on the $\boldsymbol{n}-$dimensional simplex. For example, in two dimensions the points are arranged in an equilateral triangle, and for three dimensions of parameters points are arranged in a tetrahedron. The program reads a simple text file that provides the names of model parameters and the range of their prior distribution. Options for *Simplex sampler* are taken from a separate text file. This enables the User to make choices, such as which algorithm to apply when generating the training points. *Simplex Sampler* determines the number of training points based on the algorithm. The User is free to run the full model at additional points.

3.2 Simplex Parameters (not model parameters!)

These are parameters representing choices made by the User. Note these are NOT the model parameters, which are then generated by *Simplex Sampler*. If one visits the User's project directory, these parameters are typically stored in the file `MY_PROJECT/parameters/simplex_parameters.txt`, where the path is relative to the project directory. Parameters files can have any name or location. These files are text files in the format. An example of a parameter file is:

```
#Simplex_LogFileName          simplexlog.txt      # if blank, output to screen
Simplex_TrainType             1                   # Must be 1 or 2 or 3
Simplex_RTrain                0.95                # Radius of simplex
Simplex_ModelRunDirName       modelruns           # Directory with training
                                                          point information
```

For the parameter file, the first string is the parameter name and is followed by the value. Both are single strings (without spaces). The # symbol is used for comments. Each parameter has a default value, which will be used if the parameter is not mentioned in the parameter file. *Simplex Sampler* has four User-defined parameters.

1. **Simplex_TrainType**
   Possible values are "1", "2" or "3". The default, "1", will position points according to a simplex, i.e. in two dimensions this is an equilateral triangle and in three dimensions, it is a tetrahedron. In $\boldsymbol{n}$ dimensions there are $\boldsymbol{n+1}$ points separated at equal distances from one another and centered at the origin.
2. **Simplex_RTrain**
   This sets the distance from the origin that the training points will be selected throughout the parameter space. The default is 0.95.
3. **Simplex_ModelRunDirName**
   This sets 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. The default is "modeldirs". The path is relative to the project directory, i.e. the directory from which you run the *simplex* command.

4. **Simplex_LogFileName**
   If this is left blank, *Simplex Sampler* will write output to the string. Otherwise it will write output to a file. Given that *Simplex Sampler* runs in a few seconds, the program is usually run interactively and output is sent to the screen.

3.3 Specifying Model Parameters and Priors

Before proceeding, Simplex requires information about the parameters, specifically, their ranges. The User enters this information into the file `Info/modelpar_info.txt`. An example of such a file might be

```
NuclearCompressibility  gaussian    210   40
ScreeningMass           uniform     0.3   1.2
Viscosity               uniform     0.08  0.3
```

The first column is the model-parameter name, and the last three parameters describe the range of the parameters, which is usually the prior, assuming the prior is uniform or Gausian. The second entry for each parameter defines whether the range/prior is uniform or `gaussian`. If the prior is `uniform`, the next two numbers specify the lower and upper ranges of the parameter. If the range/prior is `gaussian`, the third entry describes the center of the Gaussian, $\boldsymbol{\theta_0}$, and the fourth entry describes the Gaussian width, $\boldsymbol{\sigma_0}$, where the prior distribution is $\boldsymbol{\propto \exp\{-(\theta-\theta_0)^2/2\sigma_0^2\}}$. Simplex will read the information to determine the number of parameters. It will then assign the points assuming each model parameter has a range from -1 to 1, then translate and scale the points according to the ranges/priors defined in the file.

3.4 Training Types

3.4.1 Type 1

Depending on the number of parameters, $\boldsymbol{N}$, 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.

3.4.2 Type 2

Depending on the number of parameters, the program first 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.

3.4.3 Type 3

This training type creates two different simplexes to put them on top of each other, with the second simplex being a reflection of the first. The 2-dimensional visualization of this would look like the "Star of David". Finally, one extra training point is added at the origin.

3.5 Running Simplex to Generate Training Points

To run *Simplex Sampler*, first make sure the program is compiled. To compile the programs, change into the MSU_GITHOME/smooth/local/build/ diretory and enter the following command and enter

```
GITHOME_MSU/local/build% cmake .
GITHOME_MSU/local/build% make simplex
```

Next, change into your project directory and run the program.

```
MY_PROJECT% GITHOME_MSU/smooth/local/bin/simplex PARAMETER_FILE_NAME
```

Here `GITHOME_MSU` is the path to the *Smooth Emulator* installation, and `PARAMETER_FILE_NAME` is the name of the simplex parameter file, e.g. `parameters/simplex_parameters.txt`. Paths are either absolute, or are relative to the project directory.

Simplex will write the information about the training points in the directory defined by the `Simplex_ModelRunDirName` parameter. Within the directory, a sub-directory will be created for each training point, named `run0/, run1/, run2/···`. Within each subdirectory, Simplex creates a file `Simplex_ModelRunDirName/runI/mod_parameters.txt` for the $\mathrm{I}^{\mathbf{th}}$ training point. For example, the `run0/mod_parameters.txt` file might be

```
NuclearCompressibility     229.08
ScreeningMass              0.453
Viscosity                  0.192
```

At this point, it is up to the User to run their full model at each training point and create a file `runI/obs.txt`, which stores values of the observables at those training points as calculated by the full model. Once that is completed, one can run the full model at the training points.

4 Performing Full Model Runs

Once the training points are generated, the User can run the full model for each of the training points. At this point there is a directory, usually called `MY_PROJECT/modelruns/`, in which there are sub directories, `run0/, run1/, run2/....` Within each sub-directory, `runI`, there should exist a text file `MY_PROJECT/modelruns/runI/mod_parameters.txt`. These files could have been generated by *Simplex Sampler*, but could have been generated by any other means, including by hand. The files should be of the form,

```
par1_name  par1_value
par2_name  par2_value
par3_name  par3_value
.
.
```

The parameter names must match those defined in `MY_PROJECT/Info/modelpar_info.txt`, the format of which is described in Sec. 3.

The User must then perform full model runs using the model-parameter values as defined in each sub-directory. The full model runs then need to produce results and write a list of observable values in each run directory. Each file must be named `MY_PROJECT/modelruns/runI/obs.txt`. The format of those text files should be

```
observable1_name  observable1_value  observable1_random_uncertainty
observable2_name  observable2_value  observable2_random_uncertainty
observable3_name  observable3_value  observable3_random_uncertainty
.
.
```

The names must match those listed in `MY_PROJECT/Info/observable_info.txt`, which will be used by *Smooth Emulator*, as described in Sec. 5. The values are the observable values as calculated by the full model for the model-parameter values listed in the corresponding `mod_parameters.txt` file in the same directory. The random uncertainties refer only to those uncertainties due to noise in the full model. Random noise is that, which if the full model would be rerun at the same model-parameter values, would represent the variation in the observable values. In most cases this would be set to zero. But, if the full model has some aspect of sampling to it, for example generating observables from event generators with a finite number of events, that variation should be listed here. This variation is required for the emulator. If there is such a variation, the emulator should not be constrained to exactly reproduce the training point observables at the training points.

Once the observable files are produced for each of the full model runs, the User can then proceed to build and tune an emulator using *Smooth Emulator*.

5 Tuning the Emulator

5.1 Summary

Smooth emulator finds a sample set of Taylor expansion coefficients that reproduce a set of observables at a set of training points. For a given observables, a particular sample set of coefficients gives the following emulated function:

$$\boldsymbol{E(\vec{\theta}) = \sum_{\vec{n},s.t.\sum_i n_i \leq \mathbf{MaxRank}} \frac{d(\vec{n})}{(n_1+n_2+\cdots)!} A_{\vec{n}} \left(\frac{\theta_1}{\Lambda}\right)^{n_1} \left(\frac{\theta_2}{\Lambda}\right)^{n_2} \cdots .} \tag{5.1}$$

Here, $\boldsymbol{\theta_1\theta_2\cdots}$ are the model parameters, scaled so that their priors range from -1 to +1, or if they are Gaussian, have unit variance. The degeneracy factor, $\boldsymbol{d(\vec{n})}$ is the number of different ways to sum the powers $\boldsymbol{n_i}$ to a given rank,

$$\boldsymbol{d(\vec{n}) = \frac{n_1!n_2!\cdots}{(n_1+n_2+\cdots)!}.} \tag{5.2}$$

The emulator finds a predetermined number of sets of coefficients, where each set of coefficients provides a function that reproduces the real model at the training points. The User sets the number of sets of coefficients, typically of order $\boldsymbol{N_{sample} \approx 10}$, in a parameter file. Away from the training points, the uncertainty of the emulator is represented by the spread of the values amongst the $\boldsymbol{N_{sample}}$ predictions.

Smooth Emulator solves for the coefficients from the training data, then stores those coefficients in files for later use. *Smooth Emulator* can emulate either the full-model observables directly, or their principal components. Training the emulator follows the same steps for either approach.

Before training the emulator, one must first run the full model at a given set of training points. In addition to a parameter file, which sets numerous options, the User must provide the following:

1. A file listing the names of observables, their uncertainties, and an estimate of the variance of each observable throughout the model-parameter space. This file will typically be `Info/obs.txt`, where the path is relative to the project directory.
2. If the number of full-model runs performed is $\boldsymbol{N_{train}}$, Smooth emulator requires files for each run. Each file, typically `modelruns/runI/mod_parameters.txt` where $\boldsymbol{I}$ varies from $\boldsymbol{1 - N_{train}}$, describes the point in parameter space for the $\boldsymbol{I^{th}}$ full-model run.
3. In the same directory, *Smooth Emulator* requires the observables calculated at the training points mentioned above. This information is provided in `modelruns/runI/obs.txt`.

The parameter file, typically stored in `parameters/emulator_parameters.txt`, enables the User to select numerous option. For example, the User might use training data from a different directory, not `modelruns/`, or might choose to use principal components rather than the observables directly.

In the following subsections, we first review the format for each of the required input files, then describe how to run *Smooth Emulator*, how its output is stored, and how to switch PCA observables for the real observables.

5.2 *Smooth Emulator* Parameters (not model parameters!)

Smooth Emulator requires a parameter file. This can be located anywhere, as it will be specified on the command line when running *Smooth Emulator*, but is typically `parameters/emulator_parameters.txt`. The parameter file is simply a list, of parameter names followed by values.

```
#SmoothEmulator_LogFileName          smoothlog.txt
SmoothEmulator_LAMBDA                3.0
SmoothEmulator_MAXRANK               4
SmoothEmulator_NMC                   100000  # Steps between samples
SmoothEmulator_NASample              8  # No. of coefficient samples
SmoothEmulator_TuneChooseMCMC        true # set true if NPars>~5
SmoothEmulator_UseSigmaYRreal        false
SmoothEmulator_ConstrainA0           true
SmoothEmulator_CutoffA               false
SmoothEmulator_ModelRunDirName       modelruns
SmoothEmulator_TrainingPts           0-9
SmoothEmulator_UsePCA                false
RANDY_SEED                           1234
```

If any of these parameters are missing from the parameters file, *Smooth Emulator* will assign a default value.

1. **SmoothEmulator_LAMBDA**
   This is the smoothness parameter $\mathbf{\Lambda}$. It sets the relative importance of terms of various rank. If $\mathbf{\Lambda}$ is unity or less, it suggests that the Taylor expansion converges slowly. The default is 3.
2. **SmoothEmulator_LogFileName**
   If this is commented out, as it is in the example above, *Smooth Emulator*'s main output will be directed to the screen. Otherwise, the output will be recorded in the specified file.
3. **SmoothEmulator_MAXRANK**
   As *Smooth Emulator* assumes a Taylor expansion, this the maximum power of $\boldsymbol{\theta^n}$ that is considered. Higher values require more coefficients, which in turn, slows down the tuning process. The default is 4.
4. **SmoothEmulator_TuneChooseMCMC**
   If set to `false`, *Smooth Emulator* will set all but $\boldsymbol{N_{\mathbf{train}}}$ coefficients randomly, according to their Gaussian prior. Then, it will solve for the remaining coefficients in order to fit the training data. The weight is calculated for the remaining coefficients, at which point the coefficients are kept or rejected proportional to the weight. The coefficients chosen in this manner are perfectly independent of one another, but at the cost of requiring many samplings before finding a weight to keep. This choice is efficient when the number of training points is small. If `SmoothEmulator_SmoothEmulator_TuneChooseMCMC` is set to `true`, *Smooth Emulator* will choose the coefficients as a small random step from the previous coefficients, then keep or reject the coefficients according to a Metropolis algorithm. The downside is that many steps are required to create a sampling set of coefficients that are independent of one another. This method is preferable for larger numbers of training points. The default is `true`.

5. **SmoothEmulator_NMC**
   When the previous parameter is set to `true`, this sets the number of steps between retained samples of coefficients. For larger numbers of parameters, this should be set at many thousands. Higher values lead to more independent sets of coefficients, but the calculation then requires more time.

6. **SmoothEmulator_NASample**
   Smooth Emulator finds $\boldsymbol{N_{\mathrm{sample}}}$ sets of coefficients. Each set reproduces the training points, but differs away from the training points. Setting $\boldsymbol{N_{\mathrm{sample}} \sim 10}$ should reasonably represent the uncertainty of the emulator. The default is set at 8.

7. **SmoothEmulator_UseSigmaYRreal**
   If the real model has noise, the emulator should not be constrained to exactly reproduce the observables at the training points. In fact, if two training points are located close to one another in parameter space, *Smooth Emulator* might be force to find a particularly uneven function so that the points are exactly reproduced. If the User wishes to exactly reproduce the training points, this should be set to `false`, as is the default.

8. **SmoothEmulator_ConstrainA0**
   The coefficients in the Taylor expansion are assumed to have some weight,

$$\boldsymbol{W(A_i) = \frac{1}{\sqrt{2\pi\sigma_A^2}} e^{-A_i^2/2\sigma_A^2}.}$$

   The term $\boldsymbol{\sigma_A}$ is allowed to vary during the tuning to maximize the likelihood of the expansion. If the User wishes to exempt the lowest term, i.e. the constant term in the Taylor expansion from the weight, the User may set `SmoothEmulator_ConstrainA0` to `false`. The default is `false`.

9. **SmoothEmulator_CutoffA**
   This applies an additional multiplicative weight to the weight for $\boldsymbol{A}$ above.

$$\boldsymbol{W(A_i)_{\mathrm{additional}} = \frac{1}{1 + \frac{1}{4}\frac{A_i^2}{\sigma_A^2}}.}$$

   Here $\boldsymbol{\sigma_{A0}}$ is the initial guess for the spread. This can safeguard against the width $\boldsymbol{\sigma_A}$ drifting off to arbitrarily large values. Unless necessary, it is recommended to leave this at the default, `false`.

10. **SmoothEmulator_ModelRunDirName**
    This gives the directory in which the training data from the full model runs is stored. The default is `modelruns`.

11. **SmoothEmulator_TrainingPts**
    This lists which full-model training runs SmoothEmulator will use to train the emulator. This provides the User with the flexibility to use some subset for training, as may be the case when testing the accuracy. The default is "1". An example the User might enter could be
    `SmoothEmulator_TrainingPts 0-4,13,15`
    This would choose the training information from the directories `run0,run1,run2,run3,run4,run13` and `run14`, which would be found in the directory denoted by the `SmoothEmulator_ModelRunDirName` parameter.

12. **RANDY_SEED**
    This sets the seed for the random number generator. If the line is commented out, it will be set to `std::time(NULL)`.
13. **SmoothEmultor_UsePCA**
    By default, this is set to false. If one wishes to emulate the PCA observables, i.e. those that are linear combinations of the real observables, this should be set to true. One must then be sure to have run the pca decomposition programs first. For more, see Sec. 6.

5.3 Editing Info Files

1. **Info/observable_info.txt**
   Smooth Emulator requires knowledge of the observables. An example of such a file is

   ```
   meanpt_pion  40
   meanpt_proton 60
   meanv2_pion 0.05
   ```

   The second line provides an initial estimate for the parameter $\boldsymbol{\sigma_A}$. The User needn't worry if this is off by a few factors of two from the final value, but if it is off by orders of magnitude, it might take *Smooth Emulator* a long time to find the appropriate range.

2. **Info/modelpar_info.txt**
   This file provides the names and ranges of the model parameters, i.e. the prior. *Smooth Emulator* translates the scales the parameters so that they have uniform ranges, in the case of uniform distributions, or uniform widths, and zero mean for the Gaussian distributions. This same file was used for running *Simplex Sampler* and is described in Sec. 3.3.

5.4 Running the *Smooth Emulator* Program

The source code for the *Smooth Emulator* main program can be found in the `GITHOME_MSU/smooth/local/main` directory. This directory contains source code for several main programs. They are separated from the bulk of the software, which is in the `GITHOME_MSU/smooth/software/` directory. The main programs are designed so that the User can easily copy and edit them to create versions that might be more appropriate to the User's specific needs. When compiled, from the `GITHOME_MSU/smooth/local/build/` directory, the executables appear in the `GITHOME_MSU/smooth/local/bin/` directory. The source code `smoothy_tune_main.cc` is

```
using namespace std;
int main(int argc,char *argv[]){
    if(argc!=2){
        printf("Usage smoothy emulator parameter filename");
        exit(1);
    }
    CparameterMap *parmap=new CparameterMap();
```

```
    parmap->ReadParsFromFile(string(argv[1]));
    CSmoothMaster master(parmap);
    master.ReadTrainingInfo();
    master.GenerateCoefficientSamples();
    //master.TestAtTrainingPts();
    master.WriteCoefficientsAllY();
    return 0;
}
```

Similarly, there is a code `GITHOME_MSU/smooth/local/main_programs/smoothy_readcoefficients_main.cc`, which provides an example of how one might read the coefficients for later use.

From within the `GITHOME_MSU/smooth/local/build/` directory, one can compile the tuning program with the commands:

```
~/GITHOME_MSU/smooth/local/build % cmake .
~/GITHOME_MSU/smooth/local/build % make smoothy_tune
```

The executable `smoothy_tune` should now appear in the `GITHOME_MSU/smooth/local/bin/` directory. Assuming the `bin/` directory has been added to the User's path, the User may switch to the User's project directory, and enter the command

```
~/MY_PROJECT % smoothy_tune PARAMETERS/MY_PARAMETERS.TXT
```

In the above, `GITHOME_MSU` is the path of the *Smooth Emulator* installation. `MY_PROJECT`, and `PARAMETERS/MY_PARAMETERS.TXT` would be replaced by the User.

The program will write the Taylor coefficients for the $\boldsymbol{N}_{\mathbf{sample}}$ samples to files in the `coefficients` directory. The coefficients for each observable are given in separate subdirectories, named by the observables. Along with the coefficients, in the same directory *Smooth Emulator* writes a file `meta.txt` which provides information, such as the maximum rank and net number of model parameters, to make it possible to read the coefficients later on.

Smooth Emulator will output lines describing its progress, either to the screen or to a file, as specified by the `SmoothEmulator_LogFile` parameter described above. This output includes a report on the percentage of steps in the MCMC program that were successful. The line `BestLogP/Ndof` describes the weight used to evaluate the likelihood of a coefficients sample. This value should roughly converge once the Metropolis procedure has settled on the most likely region.

6 Emulating Principal Components

This section is incomplete, and the PCA software is not yet fully tested.

6.1 Summary

Rather than emulating all observables, it can be more efficient to emulate a handful of principal components. After generating the training-point data, one can run the `pca` program included with the distribution. This will create files that shadow those used to emulate the observables. This will create a file `Info/pca_info.txt` alongside `Info/observable_info.txt`. The difference is that the observables will be named `z1,z2`$\cdots$. In each run directory, alongside the `obs.txt` files, there will be a `obs_pca.txt` file. Finally, there will be a file `PCA_Info/tranformation_info.txt` file that contains all the information and matrices required to perform the basis transformation. If the parameter `Use_PCA` is set to `true`, the emulator will use the PCA files above instead of the observable files. The emulator will then store the Taylor coefficients in the directory `coefficients_pca/` rather than in `coefficients/`.

To get an idea of the capabilities and functionality of the PCA elements of the *Smooth Emulator* Distribution, one can view the sample main program,
`GITHOME_MSU/smooth/local/main_programs/pca_main.cc`.

6.2 PCA Parameters (not model parameters!)

The PCA programs uses parameter that are prefixed with **SmoothEmulator**. One would typically use the same parameter file as used for running *Smooth Emulator*. The relevant parameters are:

1. **SmoothEmultor_UsePCA**
   If one wishes to emulate the PCA observables, i.e. those that are linear combinations of the real observables, this should be set to true. One must then be sure to have run the PCA decomposition programs first.
2. **SmoothEmulator_ModelRunDirName** and **SmoothEmulator_TrainingPts** should be set the same as used by *Smooth Emulator*.

6.3 Running the PCA programs

The first sample program is `pca_calctransformation`, which reads the training information from the full model runs and calculates the principal component information. Quantities such as the PCA eigenvalues and eigenvectors are stored. This provides the User knowledge of which linear combination of observables carry significant resolving power. By storing the eigenvectors, the information may be retrieved later. This allows the User to easily transform from the observable, $\boldsymbol{y_a}$, to the PCA components $\boldsymbol{z_a}$.

One should first to `GITHOME_MSU/local/build` and compile/install the program
`GITHOME_MSU/local/bin/pca_calctransformation`.

```
.../local/build % cmake .
.../local/build % make pca_calctransformation
```

Next, from the project directory (assuming the training point information has already been collected) one can enter the command (assuming the path includes `GITHOME_MSU/local/bin`)

```
../my_project/ % pca_calctransformation PARAMETER_FILENAME
```

Here, `PARAMETER_FILENAME` is likely `parameters/emulator_parameters.txt`. At this point, all the information about observables from the training has equivalent representation for the PCA components.

In order to emulate the PCA components, one must set the parameter `SmoothEmulator_UsePCA` to true. Then, running the program `smoothy_tune` as described above will build and tune an emulator for the PCA components. It will store the Taylor coefficients in the directory `coefficients_pca/`.

The second sample program reads the transformation information written by `pca_calctransformation`. This program gives an example of transforming a vector of principal components, $\boldsymbol{z_a}$, to a vector of observables $\boldsymbol{y_a}$. To compile the programs, change into the build directory as above and enter:

```
.../local/build % cmake .
.../local/build % make pca_readtransformation
```

7 MCMC Generation of Posterior

An MCMC module will be added soon. Hopefully, it is not too difficult for the User to incorporate the emulator into a third-party MCMC program. An MCMC code specifically written to incorporate *Smooth Emulator* should be ready by Spring of 2024.

8 Template-Based Tutorial

8.1 Overview

This section privies the steps the user takes in order to run the full template code provided with the explanations in the sections above. The User should refer to this section to get a brief overview of how to run the template and edit their respective `my_project/` directory.

8.2 Step 1: Installation

After completing the necessary prerequisites listed in section 2[Installation] and following the steps outlined in section 2[Prerequisites] to install the required cmake, eigen, and gsl libraries, and setting the Home Environment Variable by creating the Home Directory as described in section 2[Making Home Directory and Setting Home Environment Variable], the user must proceed to clone the smooth and commonutils directories and compile the libraries, as explained in sections 2[Downloading] and [Compiling Libraries]. Then, the user can establish a personalized project directory by duplicating the project_template directory onto their computer.

8.3 Step 2: Running the Simplex

After all the installations the User should be present in the `GITHOME MSU/smooth/project_template` Directory and start by running the Simplex to generate the training points by entering the following command from your project directory:

```
~/PROJECT_TEMPLATE % GITHOME_MSU/smooth/local/bin/simplex PARAMETER_FILE_NAME
```

Here `GITHOME_MSU` is the path to the *Smooth Emulator* installation, and `PARAMETER_FILE_NAME` is the name of the simplex parameter file, e.g. `parameters/simplex_parameters.txt`. Paths are either absolute or relative to the project directory.

The information about the training points will be stored by Simplex in the directory specified by the `Simplex_ModelRunDirName` parameter. For each training point, a subdirectory named `run0/`, `run1/`, `run2/···` will be created within this directory. Inside each subdirectory, Simplex will create the file `mod_parameters.txt` to store the specific parameters for that training point. For more information refer to the section 3[*Simplex Sampler*].

Within each subdirectory, Simplex creates a file `runI/mod_parameters.txt` for the $\mathrm{I}^{\mathbf{th}}$ training point. For example, the `run0/mod_parameters.txt` file might be

```
NuclearCompressibility     229.08
ScreeningMass              0.453
Viscosity                  0.192
```

8.4 Step 3: Fake full model template

Once the training points have been generated, the user will input a Real full model based on the given structure, tailored to address their specific problem. The full Real model will input the model_prior_info.txt and observable_info.txt

Within `GITHOME MSU/smooth/local/main programs/fakemodel/` there are three files. The first is the information about the model parameters, and their priors is stored in `GITHOME MSU/smooth/local/fakemod` and information about the observables is store in `GITHOME MSU/smooth/templete/Info/observable_info.tx` The code `GITHOME MSU/smooth/local/fakemodel/templatemod.py` in simulates a fake mathematical model to generate data based on specific functions and coefficients. The user inputs the precise model based on the given structure, tailored to address their specific problem. The code outputs the Model Runs Files, creates the `GITHOME MSU/smooth/project_template/modelruns` directory that stores information for each full-model run. To run the program, use the command:

```
~/PROJECT_TEMPLATE % python3 GITHOME MSU/smooth/local/fakemodel/templatemod.py
```

The program reads the parameters, generates obs.txt files as output, and creates output files within the `PROJECT_TEMPLATE/modelruns` directories, The directories `PROJECT_TEMPLATE/modelruns/run0/`,`PROJECT` $\cdots$, having files describing the model parameters for each run, along with the output required by the emulator for each specific full-model run. These files contain observables generated by the fake model, including values and associated sigma values. The code's structure includes the 'FakeModel' class, which creates random coefficients, calculates specific functions, and prints information related to the fake model. The user should replace the observable files and change the following parameters according to the given Real model as needed.

8.5 Step 4: Running the *Smooth Emulator*

To compile the tuning program, User should navigate to `GITHOME_MSU/smooth/local/build/` directory and execute the following commands.

```
~/GITHOME_MSU/smooth/local/build % cmake .
~/GITHOME_MSU/smooth/local/build % make smoothy_tune
```

The executable `smoothy_tune` should now appear in the `GITHOME_MSU/smooth/local/bin/` directory. After the addition of the `bin/` directory to the User's path, the User may switch to the User's project directory, and enter the command

```
~/MY_PROJECT % GITHOME_MSU/smooth/local/bin/smoothy_tune PARAMETERS/MY_PARAMETERS.TXT
```

In the above, `GITHOME_MSU` is the path of the *Smooth Emulator* installation. `PROJECT_TEMPLATE`, and `PARAMETERS/emulator_parameters.txt` would be replaced by the User.

The program generates Taylor coefficients for $\boldsymbol{N}_{\mathbf{sample}}$ samples and saves them in the `coefficients` directory. Each observable has its own subdirectory with its name. Besides the coefficients, *Smooth Emulator* also produces a `meta.txt` file in the same directory. It includes essential information like

the observable's maximum rank and net number of model parameters. This file can be used to access the coefficients later.

Smooth Emulator offers progress updates throughout its execution, which may appear on the screen or be saved to a file, depending on the settings established in the SmoothEmulator_LogFile parameter. The progress report encompasses data on the success rate of the MCMC program. Specifically, the line BestLogP/Ndof represents the weight utilized to assess the likelihood of a coefficients sample. It is worth noting that this value should converge roughly when the Metropolis procedure settles on the most probable region. For Further understanding refer to section 5Tuning the Emulator

9 Underlying Theory of *Smooth Emulator*

The choice of model emulators, $\boldsymbol{E}(\vec{\boldsymbol{\theta}})$, depends on the prior understanding of the model being emulated, $\boldsymbol{M}(\vec{\boldsymbol{\theta}})$. If one knows that a function is linear, then a linear fit is clearly the best choice. Whereas to reproduce lumpy features, where the lumps have a characteristic length scale, Gaussian process emulators are highly effective. The quality of an emulator can be assessed through the following criteria:

- $\boldsymbol{E}(\vec{\boldsymbol{\theta}}_t) = \boldsymbol{M}(\vec{\boldsymbol{\theta}}_t)$ at the training points, $\vec{\boldsymbol{\theta}}_t$.
- The emulator should reasonably reproduce the model away from the training points. This should hold true for either interpolation or extrapolation.
- The emulator should reasonably represent its uncertainty
- A minimal number of training points should be needed
- The method should easily adjust to larger numbers of parameters, $\boldsymbol{\theta_i},\ \boldsymbol{i = 1 \cdots N}$
- The emulator should not be affected by unitary transformations of the parameter space
- The emulator should be able to account for noisy models
- Training and running the emulator should not be numerically intensive

Here the goal is to focus on a particular class of functions: functions that are *smooth*. Smoothness is a prior knowledge of the function. It is an expectation that the linear terms of the function are likely to provide more variance than the quadratic contributions, which are in turn likely to be more important than the cubic corrections, and so on.

9.1 Mathematical Form of *Smooth Emulator*

To that end the following form for $\boldsymbol{E}(\vec{\boldsymbol{\theta}})$ is chosen,

$$\boldsymbol{E}(\vec{\boldsymbol{\theta}}) = \sum_{\vec{\boldsymbol{n}},\textbf{s.t. } \boldsymbol{K}(\vec{\boldsymbol{n}}) \leq \boldsymbol{K}_{\max}} \boldsymbol{d}_{\vec{\boldsymbol{n}}} \boldsymbol{f}_{\boldsymbol{K}(\vec{\boldsymbol{n}})}(|\vec{\boldsymbol{\theta}}|) \boldsymbol{A}_{\vec{\boldsymbol{n}}} \left(\frac{\boldsymbol{\theta_1}}{\boldsymbol{\Lambda}}\right)^{\boldsymbol{n_1}} \left(\frac{\boldsymbol{\theta_2}}{\boldsymbol{\Lambda}}\right)^{\boldsymbol{n_2}} \cdots \left(\frac{\boldsymbol{\theta_N}}{\boldsymbol{\Lambda}}\right)^{\boldsymbol{n_N}}. \tag{9.1}$$

Each term has a rank $\boldsymbol{K}(\vec{\boldsymbol{n}}) = \boldsymbol{n_1} + \boldsymbol{n_2} + \cdots \boldsymbol{n_N}$. If $\boldsymbol{f}$ is constant, the rank of that term corresponds to the power of $|\vec{\boldsymbol{\theta}}|/\boldsymbol{\Lambda}$. All terms are included up to a given rank, $\boldsymbol{K}_{\max}$. The coefficients $\boldsymbol{A}$ are stochastically distributed. The coefficients $\boldsymbol{d}_{\vec{\boldsymbol{n}}}$ will ensure that the variance is independent of the direction of $\vec{\boldsymbol{\theta}}$, with the constraint that $\boldsymbol{d}_{\boldsymbol{K},\boldsymbol{0},\boldsymbol{0}\ldots} = \boldsymbol{1}$. The function $\boldsymbol{f}_{\boldsymbol{K}}(|\vec{\boldsymbol{\theta}}|)$ provides the freedom to alter how the behavior depends on the distance from the origin, $|\vec{\boldsymbol{\theta}}|$, and on the rank, $\boldsymbol{K}$. Given that the variance of $\boldsymbol{A}_{\vec{\boldsymbol{n}}}$ can be changed, $\boldsymbol{f}_{\boldsymbol{K}=\boldsymbol{0}}(|\vec{\boldsymbol{\theta}}| = \boldsymbol{0})$ is also set to unity for all $\boldsymbol{K}$ without loss of generality. For each combination $\vec{\boldsymbol{n}}$, the prior probability for any the $\boldsymbol{A}$ coefficients is given by

$$\begin{aligned} \boldsymbol{p}(\boldsymbol{A}_{\vec{\boldsymbol{n}}}) &= \frac{\boldsymbol{1}}{\sqrt{\boldsymbol{2\pi\sigma^2_{K(\vec{n})}}}} \boldsymbol{e}^{-\boldsymbol{A^2_{\vec{n}}/2\sigma^2_{K(\vec{n})}}}, \\ \langle \boldsymbol{A^2_{\vec{n}}} \rangle &= \boldsymbol{\sigma^2_{K(\vec{n})}}. \end{aligned} \tag{9.2}$$

The variance, $\boldsymbol{\sigma_K^2}$, is allowed to vary as a function of $\boldsymbol{K}$.

The parameter $\boldsymbol{\Lambda}$ will be referred to as the *smoothness parameter*. Here, we assume that all parameters have a similar range, of order unity, e.g. $\boldsymbol{-1 < \theta_i < 1}$. Thus, the relative importance of each term Eq. (9.1) falls with increasing rank, $\boldsymbol{K}$, as $\boldsymbol{(1/\Lambda)^K}$. For now, the smoothness parameter is fixed by prior knowledge, i.e. one chooses higher values of $\boldsymbol{\Lambda}$ if one believes the function to be close to linear.

First, we consider the variance of the emulator at a given point, $\boldsymbol{\vec{\theta}}$. Requiring that the variance is independent of the direction of $\boldsymbol{\vec{\theta}}$ will fix $\boldsymbol{d_{\vec{n}}}$. For example, transforming $\boldsymbol{\theta_1}$ and $\boldsymbol{\theta_2}$ to parameters $\boldsymbol{(\theta_1 \pm \theta_2)/\sqrt{2}}$ should not affect the accuracy or uncertainty of the emulator.

At $\boldsymbol{|\vec{\theta}| = 0}$ the only term in Eq. (9.1) that contributes to the variance is the one $\boldsymbol{K = 0}$ term. Averaging over the $\boldsymbol{A}$ coefficients, which can be either positive or negative with equal probability,

$$\boldsymbol{\langle E(\vec{\theta})\rangle = 0,} \tag{9.3}$$

where the averaging refers to an average over the $\boldsymbol{A}$ coefficients. At the origin, $\boldsymbol{|\vec{\theta}| = 0}$, the variance of $\boldsymbol{E}$ is

$$\boldsymbol{\langle E(\theta_1 = \theta_2 = \cdots \theta_N = 0)^2\rangle = d^2_{n_i=0}\sigma^2_{K=0}f^2_{K=0}(\vec{\theta} = 0).} \tag{9.4}$$

Choosing $\boldsymbol{f_{K=0}(0) = 1}$ and $\boldsymbol{d_{n_i=0} = 1}$, the variance of $\boldsymbol{E}$ is indeed $\boldsymbol{\sigma_0^2}$. The variance at some point $\boldsymbol{\vec{\theta} \neq 0}$ is

$$\boldsymbol{\langle E^2(\vec{\theta})\rangle = \sum_{\vec{n}} f^2_K(|\vec{\theta}|)\sigma^2_{K(\vec{n})}d^2_{\vec{n}}\left(\frac{\theta_1^{2n_1}}{\Lambda^2}\right)\left(\frac{\theta_2^{2n_2}}{\Lambda^2}\right)\cdots\left(\frac{\theta_N^{2n_N}}{\Lambda^2}\right).} \tag{9.5}$$

If $\boldsymbol{\langle E^2\rangle}$ is to be independent of the direction of $\boldsymbol{\vec{\theta}}$, the sum above must be a function of $\boldsymbol{|\vec{\theta}|^2}$ only. This requires the net contribution from each rank, $\boldsymbol{K}$ to be proportional to $\boldsymbol{|\vec{\theta}|^{2K}}$ multiplied by some function of $\boldsymbol{K}$. Using the fact that

$$\boldsymbol{(\vec{\theta}_a \cdot \vec{\theta}_b)^K = \sum_{\vec{n},s.t.\sum_i n_i = K} \frac{K!}{n_1!\cdots n_N!}(\theta_{a1}\theta_{b1})^{n_1}\cdots(\theta_{aN}\theta_{bN})^{n_N},} \tag{9.6}$$

one can see that if the sum is to depend only on the norm of $\boldsymbol{\vec{\theta}}$,

$$\boldsymbol{d^2_{\vec{n}} = \frac{K(\vec{n})!}{n_1!n_2!\cdots n_N!}.} \tag{9.7}$$

The factor of $\boldsymbol{K!}$ in the numerator was chosen to satisfy the condition that $\boldsymbol{d_{K,0,0,0} = 1}$.

One can now calculate the correlation between the emulator at two different points, averaged over all possible values of $\boldsymbol{A}$,

$$\boldsymbol{\langle E(\vec{\theta}_a)E(\vec{\theta}_b)\rangle = \sum_{K=0}^{K_{\max}} \sigma^2_K f^2_K(|\vec{\theta}|)\left(\frac{\vec{\theta}_a \cdot \vec{\theta}_b}{\Lambda^2}\right)^K.} \tag{9.8}$$

Requiring $\boldsymbol{f(|\theta| = 0) = 1}$ gives

$$\langle \boldsymbol{E^2(\vec{\theta} = 0))} \rangle = \boldsymbol{\sigma_0^2}, \tag{9.9}$$

and for $\boldsymbol{\vec{\theta}_a = \vec{\theta}_b}$,

$$\langle \boldsymbol{E^2(\vec{\theta} = 0))} \rangle \ = \sum_{\boldsymbol{K=0}}^{\boldsymbol{K_{\max}}} \boldsymbol{\sigma_K^2 f_K^2}(|\vec{\boldsymbol{\theta}}|) \left(\frac{|\vec{\boldsymbol{\theta}}|^2}{\boldsymbol{\Lambda^2}}\right)^{\boldsymbol{K}}. \tag{9.10}$$

To this point, the form is completely general once one requires that the variance above is independent of the direction of $\boldsymbol{\vec{\theta}}$. I.e. the function $\boldsymbol{f_K(\vec{\theta})}$ could be any function satisfying the constraint, $\boldsymbol{f_K(0) = 1}$, and $\boldsymbol{\sigma_K^2}$ could have any function of $\boldsymbol{K}$. Below, we illustrate how different choices for $\boldsymbol{f}$ or for $\boldsymbol{\sigma_K}$ affect the emulator by comparing several variations. First, we define the default form.

9.2 Alternate Forms

As stated above, once the form is to provide variances that are independent of the direction of $\boldsymbol{\vec{\theta}}$, the general form going forward is

$$\boldsymbol{E(\vec{\theta})} = \sum_{\boldsymbol{\vec{n}, \text{s.t. } K(\vec{n}) < K_{\max}}} \boldsymbol{f_K}(|\vec{\boldsymbol{\theta}}|) \left(\frac{\boldsymbol{K(\vec{n})!}}{\boldsymbol{n_1! \cdots n_N!}}\right)^{\boldsymbol{1/2}} \boldsymbol{A_{\vec{n}}} \left(\frac{\boldsymbol{\theta_1}}{\boldsymbol{\Lambda}}\right)^{\boldsymbol{n_1}} \left(\frac{\boldsymbol{\theta_2}}{\boldsymbol{\Lambda}}\right)^{\boldsymbol{n_2}} \cdots \left(\frac{\boldsymbol{\theta_N}}{\boldsymbol{\Lambda}}\right)^{\boldsymbol{n_N}}, \tag{9.11}$$

$$\boldsymbol{P(\vec{A}_n)} = \frac{\boldsymbol{1}}{\boldsymbol{(2\pi\sigma_K^2)^{1/2}}} \boldsymbol{e^{-|A_{\vec{n}}|^2/2\sigma_K^2}}.$$

Variations from the general form involve adjusting either the $\boldsymbol{K}$-dependence of the $|\vec{\boldsymbol{\theta}}|$-dependence of $\boldsymbol{f_K}(|\vec{\boldsymbol{\theta}}|)$, or adjusting the $\boldsymbol{K}$-dependence of $\boldsymbol{\sigma_K}$.

Default Form
Here, we assume $\boldsymbol{f_K}(|\vec{\boldsymbol{\theta}}|)$ is independent of $|\vec{\boldsymbol{\theta}}|$, and that $\boldsymbol{\sigma_K}$ is independent of $\boldsymbol{K}$. Further, the $\boldsymbol{K}-$dependence of $\boldsymbol{f^2}$ is assumed to be $\boldsymbol{1/K!}$. With this choice

$$\boldsymbol{E(\vec{\theta})} = \sum_{\boldsymbol{\vec{n}, K(\vec{n}) \leq K_{\max}}} \frac{\boldsymbol{1}}{\boldsymbol{\Lambda^K}} \frac{\boldsymbol{A_{\vec{n}}}}{\boldsymbol{\sqrt{n_1! n_2! \cdots n_N!}}} \boldsymbol{\theta_1^{n_1} \cdots \theta_N^{n_N}}, \tag{9.12}$$

$$\boldsymbol{P(A_{\vec{n}})} \sim \boldsymbol{e^{-A_{\vec{n}}^2/2\sigma^2}}.$$

With this form the variance increases with $|\vec{\boldsymbol{\theta}}|$,

$$\langle \boldsymbol{E^2(\vec{\theta})} \rangle \ = \boldsymbol{\sigma^2 e^{|\vec{\theta}|^2/\sigma^2}}. \tag{9.13}$$

If the function is trained in a region where the function is linear, the emulator's extrapolation outside the region will continue to be follow the linear behavior, albeit with variation from the higher order coefficients.

The choice of $\boldsymbol{f_K^2 = 1/K!}$ ensures that the sum defining $\boldsymbol{E(\vec{\theta})}$ converges as a function of $\boldsymbol{K}$ as long as $\boldsymbol{K_{\max}}$ is rather large compared to $|\vec{\boldsymbol{\theta}}|/\boldsymbol{\Lambda}$.

Variant A: Letting $\boldsymbol{\sigma_K}$ have a $\boldsymbol{K}$ dependence
One reasonable alteration to the default choice might be to allow the $\boldsymbol{K = 0}$ term to take any value, i.e. $\boldsymbol{\sigma_{K=0} = \infty}$, while setting all the other $\boldsymbol{\sigma_K}$ terms equal to one another. This would make sense if our prior expectation of smoothness meant that we expect the $\boldsymbol{K = 2}$ terms to be less important than the $\boldsymbol{K = 1}$ terms, by some factor $\boldsymbol{|\vec{\theta}|/\Lambda}$, but that the variation of the $\boldsymbol{K = 1}$ term is unrelated to the size of the $\boldsymbol{K = 0}$ term. This would make the emulator independent of redefinition of the emulated function by adding a constant. This may well be a reasonable choice for many circumstances.

Variant B: Suppressing correlations for large $\boldsymbol{\Delta\vec{\theta}}$
This form for $\boldsymbol{f}$ causes correlations to fall for points far removed from one another.

$$\boldsymbol{f_K(|\vec{\theta}|) = \frac{1}{\sqrt{K!}} \left\{ \sum_{K=0}^{K_{\max}} \frac{1}{\sqrt{K!}} \left(\frac{|\vec{\theta}|^2}{2\Lambda^2} \right)^K \right\}^{-1/2}}.$$

In the limit that $\boldsymbol{K_{\max} \to \infty}$ the form is a simple exponential,

$$\boldsymbol{f_K(|\vec{\theta}|)\Big|_{K_{\max}\to\infty} = \frac{1}{\sqrt{K!}} e^{-|\vec{\theta}|^2/2\Lambda^2}}. \tag{9.14}$$

With this form the same-point correlations remain constant over all $\boldsymbol{\vec{\theta}}$,

$$\boldsymbol{\langle E(\vec{\theta}) E(\vec{\theta}) \rangle = \sigma^2}, \tag{9.15}$$

while the correlation between separate positions fall with increasing separation. This is especially transparent for the $\boldsymbol{K_{\max} \to \infty}$ limit,

$$\boldsymbol{\langle E(\vec{\theta}_a) E(\vec{\theta}_b) \rangle_{K_{\max}\to\infty} = \sigma^2 e^{-|\vec{\theta}_a - \vec{\theta}_b|^2/2\Lambda^2}}.$$

In this limit one can also see that

$$\boldsymbol{\langle [E(\vec{\theta}) - E(\vec{\theta'})]^2 \rangle_{K_{\max}\to\infty} = 2\sigma^2 \left(1 - e^{|\vec{\theta} - \vec{\theta'}|^2/2\Lambda^2} \right)}. \tag{9.16}$$

If one trains such an emulator in one region, then extrapolates to a region separated by $\boldsymbol{|\vec{\theta} - \vec{\theta'}| >> \Lambda}$, the average predictions will return to zero. Thus, if the behavior would appear linear in some region the emulator's distribution of predictions far away (extrapolations) would center at zero. This behavior is similar to a Gaussian-process emulator.

Variant C: Eliminating the $\boldsymbol{1/K!}$ weight
Clearly, eliminating the $\boldsymbol{1/K!}$ weights in $\boldsymbol{f_K}$ would more emphasize the contributions from larger $\boldsymbol{K}$. But, for and $\boldsymbol{|\vec{\theta}| > \Lambda}$ the contribution to the variance would increase as $\boldsymbol{K}$ increases and the sum would not converge if $\boldsymbol{K_{\max}}$ were allowed to approach infinity. An example of a function that expands without factorial suppression is $\boldsymbol{1/(1-x) = 1 + x + x^2 + x^3 \cdots}$, which diverges as $\boldsymbol{x \to 1}$. If such behavior is not expected, then this choice would be unreasonable.

9.3 Tuning the Emulator

Here, we illustrate how an emulator of the form above can be constrained given training points. We consider $\boldsymbol{M}$ full model runs at positions $\boldsymbol{\vec{\theta}_{m=1,M}}$, with values $\boldsymbol{F_{m=1,M}}$. The functional form

has a large number of coefficients, $\boldsymbol{A_{\vec{n}}}$, which is much larger than the number of training points, $\boldsymbol{M}$. However, the dependence of $\boldsymbol{A}$ is purely linear. One can enumerate the coefficients as $\boldsymbol{A_c}$ with $\boldsymbol{c = 1 \cdots C}$, where $\boldsymbol{C > M}$. One can randomly set the coefficients $\boldsymbol{A_M}$ through $\boldsymbol{A_C}$, then solve for the first $\boldsymbol{M}$ coefficients by solving a linear equation. One can then apply a weight based on the values of $\boldsymbol{A}$ consistent with the prior likelihood of $\boldsymbol{A}$ and the constraints. One can then generate a representative set of $\boldsymbol{A}$, perhaps a dozen samples. Each sample function will go through all the training points, but will vary further from the training points. Averaging over the $\boldsymbol{N_{\mathbf{sample}}}$ sets of coefficients can be used to make a prediction for the emulator at some point $\boldsymbol{\vec{\theta}}$, and the variance of those $\boldsymbol{N_{\mathbf{sample}}}$ points would represent the uncertainty of the emulator.

Here, we present two different methods for generating samples of points that are consistent with the training constraints and with the prior range of parameters. We do this for the default method, but this can be easily extended to the other model variants listed in the previous section. In addition to varying the coefficients, we will additionally vary the width parameters, $\boldsymbol{\sigma}$.

First, we need to describe how the weights are generated. The probability of a set of coefficients is initially

$$\boldsymbol{P(A_c) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-A_c^2/2\sigma^2}.} \tag{9.17}$$

If we also vary $\boldsymbol{\sigma}$, we can state that

$$\boldsymbol{Q(\sigma) = \frac{1}{\pi} \frac{\Gamma/2}{\sigma^2 + (\Gamma/2)^2}.} \tag{9.18}$$

Here, the half width of the distribution should be set to some large value to encompass the degree to which the emulated function might vary. The Lorentzian form accommodates a large uncertainty. The result should not depend strongly on $\boldsymbol{\Gamma}$. The joint probability is $\boldsymbol{Q(\sigma) \prod_c P(A_c)}$.

The constrained probability is

$$\begin{aligned} \boldsymbol{dP} &= \boldsymbol{d\sigma Q(\sigma) \prod_c [dA_c\ P(A_c)] \prod_{m=1}^{M} \delta(E(A, \sigma, \vec{\theta}_m) - F(\vec{\theta}_m))} \\ &= \boldsymbol{d\sigma Q(\sigma) \prod_{c=1}^{M} P(A_c) \frac{1}{|J|} \prod_{c=M}^{C} [dA_c\ P(A_c)].} \end{aligned} \tag{9.19}$$

Here, $\boldsymbol{|J|}$ is the Jacobian, i.e. it is the determinant of the $\boldsymbol{M \times M}$ matrix

$$\boldsymbol{J_{mc} = \frac{\partial E(\vec{\theta}_m)}{\partial A_c},} \tag{9.20}$$

For the default form in the previous section,

$$\boldsymbol{J_{mc} = \frac{1}{\sqrt{n_{c1}! \cdots n_{cN}!}} (\theta_{m1})^{n_{c1}} \cdots (\theta_{mN})^{n_{cN}}.} \tag{9.21}$$

Because the Jacobian depends only on the position of the training points, it can be treated as a constant.

One can now sample $\boldsymbol{A}$ and $\boldsymbol{\sigma}$ according to the weights above. Here, we list two methods.

a) **Keep and Reject Method**
One can generate the coefficients $\boldsymbol{A_{c=M} \cdots A_C}$ according to the Gaussian distribution with width $\boldsymbol{\sigma}$, after generating $\boldsymbol{\sigma}$ according to the Lorentzian. The residual weight is then

$$\boldsymbol{w = \prod_{c=1}^{M} P(A_c).} \tag{9.22}$$

For each attempt, one could keep or reject the attempt with probability $\boldsymbol{w}$. This generates perfectly independent samples, but with the caveat that in a high dimensional space the rejection level might be quite high.

b) **Metropolis Exploration of $\boldsymbol{A}$ and $\boldsymbol{\sigma}$**
Beginning with any configuration $\boldsymbol{A}$ and width $\boldsymbol{\sigma}$, one can take a random step $\boldsymbol{\delta A}$ and $\boldsymbol{\delta\sigma}$. In the absence of any weight this would consider all $\boldsymbol{A}$ or $\boldsymbol{\sigma}$ with values from $\boldsymbol{-\infty}$ to $\boldsymbol{\infty}$. The residual weight would then be

$$\boldsymbol{w = Q(\sigma) \prod_{c=1}^{C} P(A_c).} \tag{9.23}$$

One then keeps the step if the new weight exceeds the previous one, or if the ratio of the new weight to the old weight is greater than a random number between zero and unity. After some number of steps, $\boldsymbol{N_{steps}}$, one saves the configuration as a representative sample for the emulator. The disadvantage of this approach is that many steps may be needed to ensure that the samples are independent, and thus faithfully represent the variation in the emulator.

10 Theoretical Basis of *Simplex Sampler*

Here, we imagine a spherically symmetric prior, e.g. one that is purely Gaussian, and where the parameters are scaled so that the prior distribution is invariant to rotations. If one believe the function were close to linear, a strategy would be to find $\boldsymbol{M = N + 1}$ points in parameter space placed far apart from another. One choice is the $\boldsymbol{N}-$dimensional simplex. Examples are an equilateral triangle in two-dimensions or a tetrahedron in three dimensions. For a simplex, one places the $\boldsymbol{M = N + 1}$ training points at a uniform distance from the origin, $\boldsymbol{r}$, with equal separation between each point. One can generate an $\boldsymbol{N}-$dimensional simplex from an $\boldsymbol{N - 1}$ dimensional arrangement. In the $\boldsymbol{N - 1}$ dimensional arrangement, the points are arranged equidistant from one another using the coordinates $\boldsymbol{x_1 \cdots x_N}$. The points would all be placed at a radius $\boldsymbol{r_N}$ from the origin in this system, and the separation would be $\boldsymbol{d}$. In the $\boldsymbol{N}-$dimensional system all these $\boldsymbol{N-1}$ points had coordinate $\boldsymbol{x_N = -a}$. The $\boldsymbol{(N + 1)^{th}}$ point is then placed at position $\boldsymbol{x_1 \cdots x_N = 0}$ and $\boldsymbol{x_{N+1} = Na}$. This keeps the center of mass at zero. One then chooses $\boldsymbol{a}$ such that the new point is equally separated from all the other points by the same distance $\boldsymbol{d}$,

$$\boldsymbol{d^2 = r_{N-1}^2 + N^2 a^2,} \tag{10.1}$$
$$\boldsymbol{a = \sqrt{\frac{d^2}{N^2} - r_{N-1}^2}.}$$

Now, each of the $\boldsymbol{N}$ points is located a distance $\boldsymbol{Na}$ away from the center of the $\boldsymbol{N}-$dimensional origin. This procedure can applied iteratively to generate the vertices of the simplex.

One might also wish to use enough training points to uniquely determine the emulator in the case that the function is quadratic. There are $\boldsymbol{N(N+1)/2}$ additional points, which is exactly the number of segments connecting the $\boldsymbol{N+1}$ equally-spaced vertices of the $\boldsymbol{N}-$dimensional simplex. If placed at the midpoints of the segments, these points would be closer to the origin than the vertices. One of the simplex options is to place these points at the midpoints, then double their radii while maintaining their direction. This would result in arrays of points at two different radii, with $\boldsymbol{N+1}$ points positioned at the lower radius and $\boldsymbol{N(N+1)/2}$ points being placed at the larger radius.

Choosing the training points depends on prior expectation of the emulated function. The simplex choice for the first $\boldsymbol{M = N+1}$ points seems logical. Even if another method, such as a Gaussian process emulator is to implemented, such methods are often based on first understanding the linear behavior. The simplex strategy would seem a good way to pick the first $\boldsymbol{N+1}$ training points.

One issue with the simplex is that the first set of $\boldsymbol{N+1}$ training points would all be placed at the same radius. If the prior parameter distribution is uniform within an $\boldsymbol{N}-$dimensional hyper cube, the training points could be rather far from the corners in that space. Issues with such priors are discussed in the next section.

10.1 The Pernicious Nature of Step-Function Priors in High Dimension

For purely Gaussian priors, one can scale the prior parameter space to be spherically symmetric. Unfortunately, that is not true for step function priors (uniform within some range). In that case the best one can do (if the priors for each parameter are independent) is to scale the parameter space such that each parameter has the constraint, $\boldsymbol{-1 < \theta_i < 1}$. If the number of parameters is $\boldsymbol{N}$, the hyper-cube has $\boldsymbol{2N}$ faces and $\boldsymbol{2^N}$ corners, a face being defined as one parameter being $\boldsymbol{\pm 1}$ while the others are zero, while a corner has each parameter either $\boldsymbol{\pm 1}$. For 10 parameters, there are 1024 corners, and for 15 parameters there are 32678 corners. Thus, it might be untenable to place a training point in each corner.

One can also see the problem with placing the training points in a spherically symmetric fashion as is done with the *Simplex Sampler*. The hyper-volume of the parameter-space hyper-cube is $\boldsymbol{2^N}$, whereas the volume of an $\boldsymbol{N}-$dimensional hyper-sphere of radius $\boldsymbol{R = 1}$ is

$$\boldsymbol{V_{\rm sphere} = \Omega_N \int_0^R dr\ r^{N-1} = \Omega_N \frac{R^N}{N}.} \tag{10.2}$$

The solid angle, $\boldsymbol{\Omega_N}$ in $\boldsymbol{N}$ dimensions is

$$\boldsymbol{\Omega_N = \frac{2\pi^{N/2}}{\Gamma(N/2)},} \tag{10.3}$$

and after putting this together, the fraction of the hyper-cube's volume that is within the hyper-sphere is

$$\boldsymbol{\frac{V_{\rm sphere}}{V_{\rm cube}} = \begin{cases} \frac{(\pi/2)^{N/2}}{N!!}, & N = 2,4,6,8\cdots \\ \frac{(\pi/2)^{(N-1)/2}}{N!!}, & N = 3,5,7,\cdots \end{cases}} \tag{10.4}$$

In two dimensions, the ratio is $\boldsymbol{\pi/4}$, and in three dimensions it is $\boldsymbol{\pi/6}$. In 10 dimensions it is $\boldsymbol{2.5 \times 10^{-3}}$. For high dimensions only a small fraction of the parameter space can ever lie inside inside a sphere used to place points. And, if the model is expensive, it may not be tenable to run the full model inside every corner.

One can also appreciate the scope of the problem by considering the radius of the corners vs. the radius of the sphere. The maximum value of $|\vec{\boldsymbol{\theta}}|$ is $\sqrt{\boldsymbol{N}}$. So, for 9 parameters, if the training points were all located at positions $|\vec{\boldsymbol{\theta}}| < \mathbf{1}$, one would have to extrapolate all the way to $|\vec{\boldsymbol{\theta}}| = \mathbf{3}$. Thus, unless the model is exceptionally smooth, one needs to devise a strategy to isolate the portion of likely parameter space using some original set of full-model runs, then augment those runs in the likely region.

A third handle for viewing the issue in $\boldsymbol{N}$ dimensions is to compare the r.m.s. radii of the hyper-sphere to that of the hyper-cube. For the cube where each side has length $\boldsymbol{2a}$,

$$\left(\boldsymbol{R}_{\mathbf{r.m.s.}}^{\mathbf{(cube)}}\right)^{\mathbf{2}} = \boldsymbol{a}^{\mathbf{2}}\frac{\boldsymbol{N}}{\mathbf{3}}. \tag{10.5}$$

whereas for a sphere of radius $\boldsymbol{a}$,

$$\left(\boldsymbol{R}_{\mathbf{r.m.s.}}^{\mathbf{(sphere)}}\right)^{\mathbf{2}} = \boldsymbol{a}^{\mathbf{2}}\frac{\boldsymbol{N}+\mathbf{2}}{\boldsymbol{N}}. \tag{10.6}$$

The ratio of the radii is then

$$\frac{\boldsymbol{R}_{\mathbf{r.m.s.}}^{\mathbf{(sphere)}}}{\boldsymbol{R}_{\mathbf{r.m.s.}}^{\mathbf{(cube)}}} = \sqrt{\frac{\mathbf{3}}{\boldsymbol{N}+\mathbf{2}}}. \tag{10.7}$$

Thus, in 10 dimensions, if the training points are placed at a distance $\boldsymbol{a}$ from the origin, the r.m.s. radii of the interior space would be half that of the entire space. Further, the r.m.s. radii of the points in the cube, $\boldsymbol{a}\sqrt{\boldsymbol{N/3}}$, would be about 83% higher than the radius of the training points.

Of course, these problems would be largely avoided if the number of parameters was a half dozen or fewer, or if one was confident that the function was extremely smooth. In the first two sections of this paper, the smoothness parameter, $\boldsymbol{\Lambda}$, was set to a constant. There might be prior knowledge that certain parameters affect the observables only weakly. In that case, the response to these parameters can be considered as linear. This could be done by scaling those parameters so that they vary over a smaller range. If a parameter varies only between $\boldsymbol{-0.1}$ and $\boldsymbol{0.1}$, that effectively applies a smoothness parameter in those directions that is ten times higher. Unfortunately, the choice of which parameters to rescale in this fashion would likely vary depending on which observable is being emulated. Because all the observables might be calculated in a full model run, one needs to identify parameters that would likely have weak response on all observables.

11 Tests of the *Smooth Emulator* using *Simplex Sampler* for Training

First, we show some results of one-parameter emulators. For a linear fit, two parameters (slope and intercept) would suffice. But because higher-rank contributions exist, there is a spread of functions

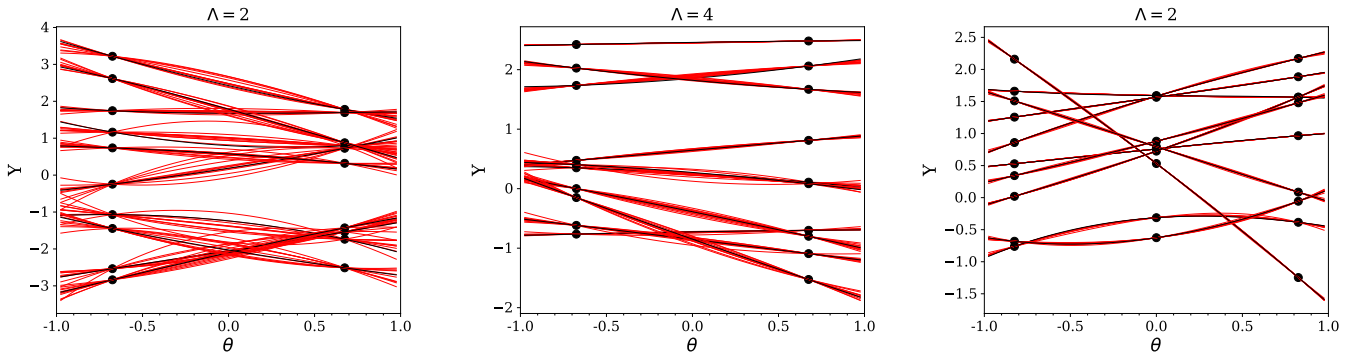


Figure 11.1: Real functions (black lines) are emulated with the *Smooth Emulator* (default settings). Ten different functions are emulated. Each emulation is sampled ten times (red lines), with the spread of the lines representing the uncertainty.

is that narrows with increasing smoothness parameter as shown in Fig. 11.1. Figure 11.1 also shows how increasing the number of training points, from two to three, also narrows the range of possible values.

Next, we repeat the same test with six parameters. The prior distribution of parameters was uniform in the region $-\mathbf{1} < \boldsymbol{\theta_i} < \mathbf{1}$. In this case, the *Simplex Sampler* was used to choose the training points. The "real" model, $\boldsymbol{F}$, was constructed from sample smooth functions, with the coefficients generated randomly and a smoothness parameter set to three or six. The $\boldsymbol{A_{\vec{n}=0}}$ coefficient for the real model was set to zero to better accommodate viewing results. The emulator was not given that information. Twenty instances of real models were emulated. For each real model five random point in parameter space were chosen. The emulator value and its uncertainty are plotted alongside the real-model values in Fig. 11.2. The 100 comparisons show that the emulator accurately predicts the uncertainty. This is not surprising, because the emulator used the same smoothness parameter as was used to construct the real models. The width parameters, $\boldsymbol{\sigma}$, were explored stochastically, and remarkably they seem to fluctuate around the same value used to construct the real model, albeit with fluctuations of the order of 30%.

Figure 11.2 first shows the 100 comparisons for the case with the smoothness parameter, $\boldsymbol{\Lambda = 3}$. The simplest simplex form was applied, which has seven parameters, the same number one would use for a linear fit. Uncertainties were provided by finding 16 independent samplings of $\boldsymbol{A}$ coefficients and of $\boldsymbol{\sigma}$, then using the variance of the 16 samples to define the uncertainty. Variant *a* of the default emulator was used, i.e. the coefficient $\boldsymbol{A_0}$ was not given a constraining prior distribution, whereas all other coefficients were weight by a Gaussian of width $\boldsymbol{\sigma}$. In the other two panels of Fig. 11.2, the procedure was repeated but once with a higher smoothness parameter, and once with a the 28 training points, placed according to the procedure mentioned in Sec. 3, where an additional set of points in parameter space was generated by choosing points that bisect all the lines connecting points in the original simplex, then doubling their radius. As expected, smoother functions are more easily emulated with a given number of training points, and using more training points also improves the accuracy.

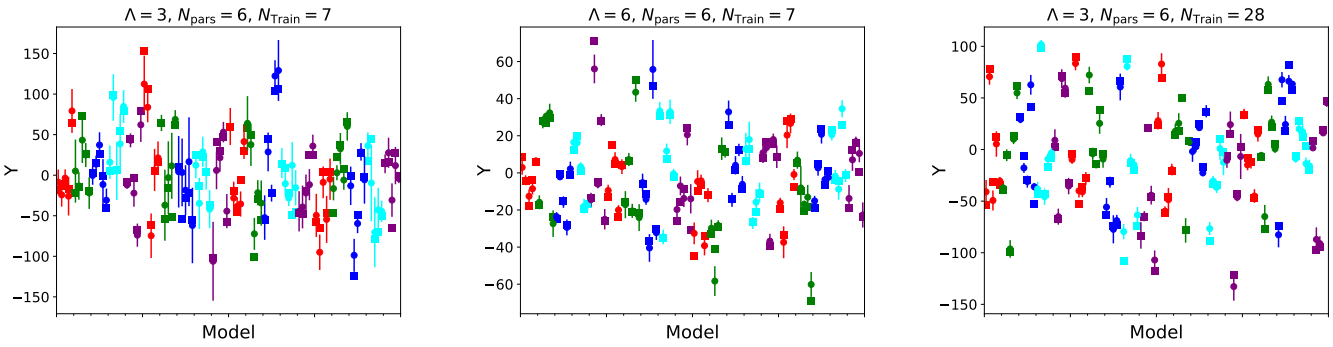


Figure 11.2: Using 20 instances of real models using six parameters, choosing 5 random points in parameter space for each model, the emulator (circles) and its uncertainty were compared to the real values (squares). Neighboring points of the same color emulated the same real model. The accuracy improves if a smoother function is considered (middle panel), or if more training points are used (right panel). Estimates of the uncertainty seem reasonable given that the uncertainties illustrated in the figure represent one standard deviation. It should be emphasized that this consistency depends critically on the fact that the emulator chose the same smoothness parameter as was used to generate the real models.