Smooth Emulator & Simplex Sampler

User Manual

A BAND Collaboration Project

https://bandframework.github.io

Scott Pratt, Eren Erdogan, Ekaksh Kataria

Department of Physics and Facility for Rare Isotope Beams

Michigan State University, East Lansing Michigan, 48824

February 26, 2024

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 $N_p + 1$, where 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 $N_p + 1$ 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, $(N_p + 1)(N_p + 2)/2$. 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)
- 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 installation. 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 *homebrew*,

```
% brew install eigen
```

2.2 Downloading the Repository and Setting the GITHOME_BAND_SMOOTH Environment Variable

The software requires downloading the BAND framework software repository into some directory. Should that be in the User's home directory, the User might enter

```
    /Users/CarlosSmith% git clone https://github.com/bandframework/bandframework.git
```

The User needs to set an environmental variable, `GITHOME_BAND_SMOOTH`, to the full path of the directory where the software is located, e.g.

```
    % export GITHOME_BAND_SMOOTH="/Users/CarlosSmith/bandframework/software/SmoothEmulator"
```

It is recommended to copy this command into the user's `.bashrc` (or equivalent) file to avoid redefining it each time one needs to recompile. Throughout the manual the phrase `GITHOME_BAND_SMOOTH` will refer to this directory.

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_BAND_SMOOTH/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 main distribution, i.e. outside the `bandframework/` path.

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_BAND_SMOOTH/templates/mylocal MY_LOCAL
```

Here `MY_LOCAL` will hence forth refer to the path of this directory. This directory should be outside the main distribution, i.e. outside the `bandframework/` path.

For the remainder of this manual, `GITHOME_BAND_SMOOTH`, `MY_LOCAL` and `MY_PROJECT` will be used to denote the location of these directories.

2.3 Directory and File Structure

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 `SmoothEmulator/` directory contains codes that are used to create libraries specific to the sampler and emulator. The executables are stored in `MY_LOCAL/bin`. The short main program source files are located in `MY_LOCAL/main_programs/`. It is not envisioned that the User would edit files in the `SmoothEmulator/software` directory, but that the User may well wish to create custom versions of the short main programs in `MY_LOCAL/main_programs/`. The main programs are compiled using the CMake files in `MY_LOCAL/build/`. The User may find it convenient to add `MY_LOCAL/bin/` to their path.

2.4 Compiling Libraries

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

```
% cd GITHOME_BAND_SMOOTH/software
GITHOME_BAND_SMOOTH/software% cmake .
GITHOME_BAND_SMOOTH/software% make
```

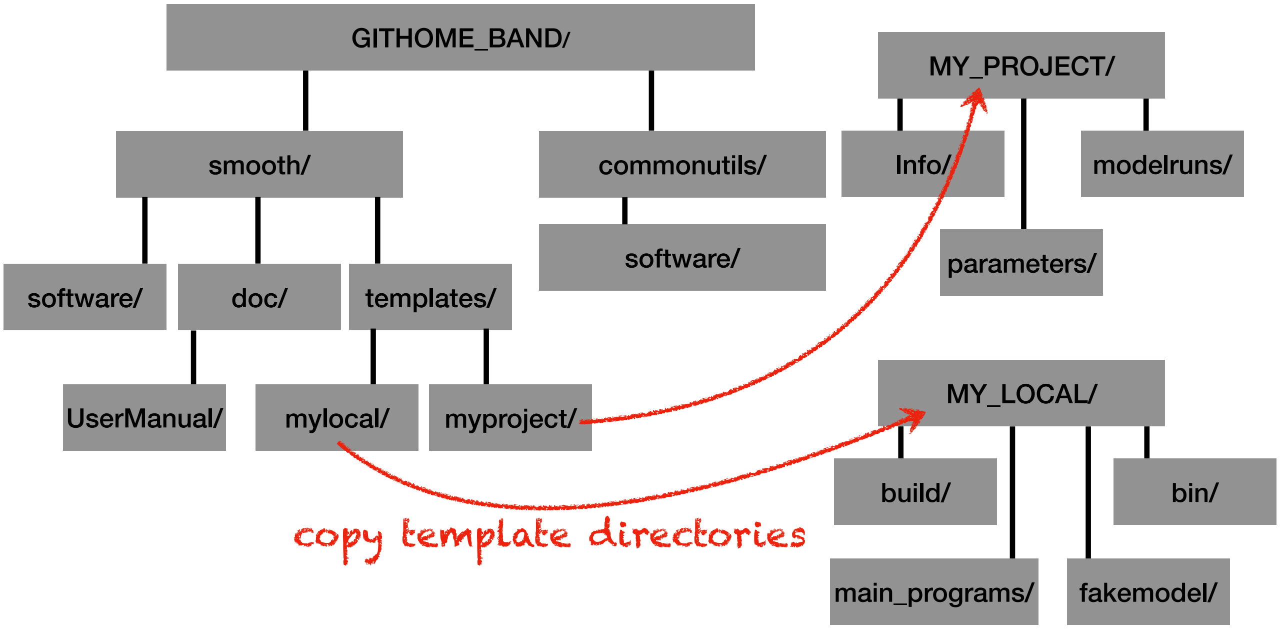

Figure 2.1: The directory structure: The User clones two repositories into some location, which will be referred to as `GITHOME_BAND`. The User can then copy two template directories into the User's choices of locations outside the path of the BAND repositories. The name of those two directories will be referred to as `MY_LOCAL`, which will contain the main programs and executables, and `MY_PROJECT` which contains the data files related to a given project. The programs are designed to be run from within the `MY_PROJECT/` directory.

There seems to be a common problem that cmake misreports the path the Eigen installation. If the User should get an error stating that the Eigen header files cannot be found, the User can set one of the following environmental variables,

```
% export EIGEN3_INCLUDE_DIR=/usr/local/include/eigen3
```

The final arguments may need to be changed depending on the User's location of the packages.

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_BAND_SMOOTH/local/build, using the existing entries as an example. The executables should appear in MY_LOCAL/bin/.

2.5 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/Info/modelpar_info.txt, and information about the observables is stored in MY_PROJECT/observable_info.txt. 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/emulator_parameters.txt. The MY_PROJECT/modelruns directory will store information for each full-model run. The directories MY_PROJECT/modelruns/run0/, MY_PROJECT/modelruns/run1/, · · · , 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 the $\boldsymbol{n}-$dimensional 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 first 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 stored in the file `${MY_PROJECT}/parameters/simplex_parameters.txt`, where the path is either absolute or 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 commented, output to screen
Simplex_TrainType             1                 # Must be 1 or 2
Simplex_ModelRunDirName       modelruns         # Directory with training
                                                          point information
                                                          lea
```

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" or "2". 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. For "2", points are added at the half-way points between each vertex of the tetrahedron. The points at the bisection points are scaled to a different radius than those at the vertices. This provides the precise number of training points to exactly determine both the linear and quadratic terms.
2. **Simplex_ModelRunDirName**
   This sets the path to the directory in which the run files will be created. The default name is `./modelruns`, but the User can change this to anything they want. The path is relative to the project directory, i.e. the directory from which you run the *simplex* command.

3. **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, x_0, and the fourth entry describes the Gaussian width, σ_0, where the prior distribution is $\propto \exp\{-(x-x_0)^2/2\sigma_0^2\}$. Simplex will read the information to determine the number of parameters. It will then assign the n points, $\theta_{1\cdots n}$ assuming each dimension of θ varies from -1 to 1, for uniform distributions, or proportional to $e^{-\theta^2/2}$ for Gaussian distributions. The points θ_i are each then converted into x_i by scaling and translating the values according to the ranges/priors defined in the `modelpar_info.txt` file.

3.4 Training Types

3.4.1 Type 1

Depending on the number of parameters, n, the program creates a simplex in n dimensions. This simplex's vertices will be used to generate $N_{\rm train} = n + 1$ 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, if the model is perfectly linear, this option provides perfect emulation.

3.4.2 Type 2

Depending on the number of parameters, the program first creates a simplex in 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. The net number of training points is then $N_{\rm train} =$

$\boldsymbol{n+1+n(n+1)/2}$. Thus, if the model is perfectly quadratic, this option provides perfect emulation.

3.4.3 Type 3

This training type creates two different simplexes, both centered at the origin. The second simplex is 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. The net number of training points is $\boldsymbol{N_{\rm train} = 2n + 3}$.

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 MY_LOCAL/main_programs/ directory and enter the following command,

```
${MY_LOCAL}/main\_programs% cmake .
${MY_LOCAL}/main\_programs% make simplex
```

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

```
${MY_PROJECT}% ${MY_LOCAL}/bin/simplex
```

Here `${MY_LOCAL}/bin` is the path to where the User compiles the main programs into executables.

Simplex will read parameters from the `./parameters/simplex_parameters.txt` file and from the `./Info/modelpar_info.txt` files. It will then 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 `runI/mod_parameters.txt` for the I$^{\rm th}$ 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.

4 Performing Full Model Runs

Once the training points are generated, the User must 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. ??.

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 `MODEL_RUN_DIRNAME/runI/obs.txt`. The directory `MODEL_RUN_DIRNAME/`. Typically this directory is `MY_PROJECT/moderundirs`, but that can be changed by the User. 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. ??. 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 User might not wish to constrain the emulator to exactly reproduce the training point observables at the training points. The principal danger being, that if two training points are very close to one another, but with a finite fluctuation, exactly producing the training points might require very high slopes to exactly reproduce the training points. If the training points are far apart from one another, and if the random uncertainties are not large, it should be safe to ignore the random uncertainty and constrain the emulator to exactly reproduce the model values. Currently, if one wishes to account for the random uncertainty the User must set the following parameters in `parameters/emulator_parameters.txt`:

a) Set either `SmoothEmulator_TuneChooseMCMC` or `SmoothEmulator_TuneChooseMCMCPerfect` to `true`.

b) Set `SmoothEmulator_MCMCUseSigmaY` to `true`.

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. The process of finding those coefficients is referred to as "tuning". For a given observable, a particular sample set of coefficients gives the following emulated function:

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

Here, $\theta_1\theta_2\cdots$ represent the original model parameters, $\vec{X}$, but are scaled. If their initial prior is uniform, they are scaled so that their priors range from -1 to +1, and if they have Gaussian priors, they are scaled so that their variance is one third. The degeneracy factor, $d(\vec{n})$ is the number of different ways to sum the powers n_i to a given rank,

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

As described in Sec. ??, the coefficients are chosen weighted by the distribution,

$$P(\vec{A}) = \prod_n \frac{1}{\sqrt{2\pi\sigma_A^2}} e^{-A_n^2/2\sigma_A^2}, \tag{5.3}$$

where σ_A is varied to maximize the overall probability given the constraint of reproducing the training points. More discussion is provided in Sec. ??. Whereas *Smooth Emulator* does a nice job of finding an optimum value for σ_A, the smoothness parameter Λ is unfortunately difficult to optimize. For the moment, this is treated purely as prior knowledge, or expectation. If the User expects the full model to be very smooth, i.e. the quadratic contributions to be much smaller than the linear contributions and so on, a larger value (e.g 3.0), might be chosen. If the full-model output might be almost wavy, then a smaller value (e.g. 1.5) might be chosen. The emulator uncertainties will be smaller for larger Λ.

By setting parameters, as described below, *Smooth Emulator* can tuned one of three different ways

a) Find the optimum set of coefficients. If evaluated a the training points, the emulator will exactly produce the full model. when it predicts the observable at a new $\vec{\theta}$ it provides an uncertainty.

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

c) The third mode also provides $N_{\rm sample}$ predictions, but rather than exactly reproducing the training values the emulator merely comes close to the training points with a distribution $\sim e^{-\Delta y^2/2\sigma_y^2}$, where σ_y represents the random error of the full model. This mode should be chosen if the full model has significant random error, and especially if the training points are close to one another.

Method (a) is by far the quickest, and will probably be used the most often.

If methods (b) or (c) are chosen *Smooth Emulator* solves for the $\boldsymbol{N}_{\mathbf{sample}}$ sets of coefficients from the training data, then stores $\boldsymbol{N}_{\mathbf{sample}}$ sets of coefficients, along with the averaged coefficients in files for later use. If (a) is chosen, *Smooth Emulator* stores the set of "best" coefficients along with some other arrays used for rapid calculation of the uncertainty. *Smooth Emulator* can emulate either the full-model observables directly, or their principal components. Training the emulator follows the same steps for either approach.

The executables based on *Smooth Emulator* are located in the User's `${MY_LOCAL}/bin` directory. Examples of such executables are `smoothy_tune` or `smoothy_calcobs`. These functions must be executed from within the User's project directory.

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 real observables.

5.2 Preparing Files for *Smooth Emulator*

Before training the emulator, one must first run the full model at a given set of training points. In addition to a parameter file (described in the next sub-section), which sets numerous options, the User must provide the following:

1. A file listing the names of observables and an estimate of the variance of each observable throughout the model-parameter space, $\boldsymbol{\sigma_A}$. This file is named `Info/observable_info.txt`, where the path is relative to the project directory. The file might look like

   ```
    obsname1  12.3
    obsname2  23.4
    obsname3  34.5
    obsname4  45.6
        ⋮
   ```

   The initial $\boldsymbol{\sigma_A}$ is only relevant if one is using one of the Monte Carlo tuning methods, (b) or (c) above, as it provides an initial guess for the parameter $\boldsymbol{\sigma_A}$ above.
2. A file listing the names of the model parameters that also describes their priors. This file is `Info/modelpar_info.txt`. The file might have the following form:

   ```
   parname1 uniform   0      1.0E-3
   parname2 uniform   -50.0  100.0
   parname3 gaussian  0      24.6
   parname4 uniform   30.0   50.0
         ⋮
   ```

   If the prior is `uniform` the two following numbers provide the minimum and maximum of the interval. If the prior is `gaussian` the two subsequent values represent the center and r.m.s. width of the Gaussian. This same file was required for running *Simplex Sampler*.

3. A list of the model-parameter values, $\vec{\boldsymbol{\theta}}_{\mathbf{train}}$, at each training point. These points can be generated by *Simplex Sampler*, as described in Sec. **??**, or they can be generated by hand. If the number of full-model runs performed is $\boldsymbol{N}_{\mathbf{train}}$, Smooth emulator requires files for each run. Each file is named `${MODEL_RUN_DIRNAME}/runI/mod_parameters.txt`, where $\mathbf{0} \leq \boldsymbol{I} < \boldsymbol{N}_{\mathbf{train}}$, and $\boldsymbol{I}$ denotes the point in parameter space for the $\boldsymbol{I}^{\mathbf{th}}$ full-model training run. The directory `${MODEL_RUN_DIRNAME}/` is typically `${MY_PROJECT}/modelruns`, but can be defined otherwise (see below). For example the file `modelruns/run0/mod_parameters.txt` might look like

   ```
   parname1  8.34E-4
   parname2  -30.5375
   parname3  36.238
   parname4  39.34
       ⋮
   ```

4. At each training point, the User must provide the full model's value for each observable. In the same directory where the model-parameter values are stored, *Smooth Emulator* requires the observables calculated at the training points mentioned above. This information is provided in `${MODEL_RUN_DIRNAME}/runI/obs.txt`. An example of such a file is:

   ```
   obsname1  -51.4645   2.5
   obsname2  166.837    0.9
   obsname3  -47.9877   0.0
   obsname4  -2.34526   0.03
       ⋮
   ```

   The first number is the calculated value of the observable, and the second is the random error. This is only the random error, i.e. that which represents that if the model were rerun at the same training point, the value might be different. This should only be non-zero if the full-model has some Monte Carlo feature. For example, the full model might involve simulating a small number of events. Other types of uncertainty are accounted for by including them into the experimental uncertainty.

   Once the emulator is tuned and before it is applied to a Markov Chain investigation of the likelihood, the software needs know the experimental measurement and uncertainty. That information must be entered in the `Info/experimental_info.txt` file. The file should have the format:

   ```
   obsname1  -12.93    0.95   0.5
   obsname2  159.3     3.0    2.4
   obsname3  -61.2.    1.52   0.9
   obsname4  -1.875    0.075  0.03
       ⋮
   ```

   The first number is the measured value and the second is the experimentally reported uncertainty. The third number is the uncertainty inherent to the theory, due to missing physics. For example, even if a model has all the parameters set to the exact value, e.g. some parameter of the standard value, the full-model can't be expected to exactly reproduce a correct measurement given that some physics is likely missing from the full model. For the MCMC software, the relevant uncertainty incorporates both, and only the combination of both, added in quadrature, affects the outcome. We emphasize that this last file is not needed to train and tune the emulator. It is needed once one performs the MCMC search of parameter space.

5.3 *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 # comment out for interactive running
 SmoothEmulator_LAMBDA 2.0 #  Smoothness parameter
 SmoothEmulator_MAXRANK 5
 SmoothEmulator_ConstrainA0 true
 SmoothEmulator_ModelRunDirName modelruns
 SmoothEmulator_TrainingPts 0-27
 SmoothEmulator_UsePCA   false
 SmoothEmulator_TuneChooseExact true
#
# These are only used if you are using MCMC tuning rather than Exact method
 SmoothEmulator_TuneChooseMCMC false # set false if NPars<5
 SmoothEmulator_MCMC_NASample 8  # No. of coefficient samples
 SmoothEmulator_MCMC_StepSize 0.01
 SmoothEmulator_TuneChooseMCMCPerfect false #
 SmoothEmulator_MCMC_UseSigmaY false # Emulator only fits training data to within model ra
 SmoothEmulator_MCMC_CutoffA false # Used only if SigmaA constrained by SigmaA0
 SmoothEmulator_MCMC_SigmaAStepSize 1.0  #
 SmoothEmulator_MCMC_NMC 20000  # Steps between samples
#
# This is for the MCMC search of parameter space (not for the emulator tuning)
MCMC_METROPOLIS_STEPSIZE 0.01
RANDY_SEED.  1234
```

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

1. **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 and the program will run interactively. Otherwise, the output will be recorded in the specified file. Most often, one will wish the program to run interactively.
2. **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.

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_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`.

5. **SmoothEmulator_ModelRunDirName**
   This gives the directory in which the training data from the full model runs is stored. The default is `modelruns`, which is the same default `Simplex Sampler` uses for writing the coordinates of the training points.

6. **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 `run15`, which would be found in the directory denoted by the `SmoothEmulator_ModelRunDirName` parameter.

7. **SmoothEmulator_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. **??**.

8. **SmoothEmulator_TuneExact**
   This is set to `true` by default. In this case *Smooth Emulator* finds the optimum set of coefficients and also records some other arrays necessary for calculating the emulator uncertainty. This is tuning type (a) above.

9. **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)`.

   ### None of the parameters below are relevant if `SmoothEmulator_TuneExact` is `true`.

10. **SmoothEmulator_TuneChooseMCMC**
    This is tuning type (b) above. Rather than finding the optimum set of coefficients, *Smooth Emulator* finds $\boldsymbol{N_{\rm sample}}$ sets of coefficients consistent with the probability. All the sets exactly reproduce training observables. As this runs as a Markov chain, the independence of the sample may require a large number of steps between samplings. The default is `false`.

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

12. **SmoothEmulator_MCMC_NMC**
This is the number of Markov Chain steps between samples. A larger number is required if the samplings are to be independent. The default is 20,000.

13. **SmoothEmulator_MCMC_StepSize**
This is the size of the MCMC stepsize in $\boldsymbol{\theta}$ space. MCMC approaches are most efficient if the success probability is near 0.5. If the probability is much higher, then the step size should be increased, and if it is much lower, the step size should be decreased. This affects only the efficiency, not the accuracy. The default of 0.01.

14. **SmoothEmulator_TuneChooseMCMCPerfect**
If there are a small number of model parameters, perhaps less than a half dozen, then rather than performing a Markov Chain one can choose the coefficients by a keep-or-reject method. The advantage of this approach is that $\boldsymbol{N_{\rm sample}}$ coefficients are perfectly independent. The disadvantage is that the the percentage of "keeps" falls rapidly with an increasing number of parameters. For larger numbers of parameters it is usually more efficient to set this to its default, `false`.

15. **SmoothEmulator_MCMC_UseSigmaY**
If the real model has significant random noise, the emulator should not be constrained to exactly reproduce the observables at the training points. This is tuning type (c) above.

16. **SmoothEmulator_MCMC_CutoffA**
This applies an additional multiplicative weight to the weight for $\boldsymbol{A}$ above:

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

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`.

5.4 Running the *Smooth Emulator* Program

The source code for several *Smooth Emulator* main programs can be found in the `${MY_LOCAL}/main_programs/` directory. They are separated from the bulk of the software, which is in the `GITHOME_BAND/SmoothEmulator/so` directory. The main programs are designed so that the User can easily copy and edit them to create versions that might be more appropriate to the User's specific needs. When compiled, from the `${MY_LOCAL}/build/` directory, the executables appear in the `${MY_LOCAL}/bin/` directory. Two of the source codes that come with the distributions are ${MY_LOCAL} and `${MY_LOCAL}/main_programs/smoothy_calc_main.cc`. Once compiled the corresponding executables are `${MY_LOCAL}/bin/smoothy_tune` and `${MY_LOCAL}/bin/smoothy_calcobs`.

```
using namespace std;
```

```
int main(){
    CparameterMap *parmap=new CparameterMap();
    parmap->ReadParsFromFile(string(argv[1]));
    CSmoothMaster master(parmap);
    master.ReadTrainingInfo();
    master.GenerateCoefficientSamples();
    master.WriteCoefficientsAllY();
    return 0;
}
```

Similarly, there is a code `${MY_LOCAL}/main_programs/smoothy_calcobs_main.cc`, which provides an example of how one might read the coefficients and generate predictions for the emulator at specfied points in parameter space.

From within the `${MY_LOCAL}/build/` directory, one can compile the two programs with the commands:

```
 MY_LOCAL/build % cmake .
 MY_LOCAL/build % make smoothy_tune
 MY_LOCAL/build % make smoothy_calcobs
```

The executables `smoothy_tune` and `smoothy_calcobs` should now appear in the `${MY_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
```

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, i.e. `coefficients/OBS_NAME/sampleI.txt`. Here, , where `OBS_NAME` is the name for each observable, and if there are $\boldsymbol{N}_{\mathbf{sample}}$ sets of coefficients, $\mathbf{0} \leq \boldsymbol{I} < \boldsymbol{N}_{\mathbf{sample}}$. Along with the coefficients, in the same directory *Smooth Emulator* writes a file for each observable. These files are named `coefficients/OBS_NAME/meta.txt`. This file 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 plateau once the Metropolis procedure has settled on the most likely region.

For later us, e.g. when performing the MCMC to sampler the posterior, the User would need to generate predictions for specified values of the parameters. The executable `${MY_LOCAL}/bin/smoothy_calcobs`, is such an example. It is compiled from the main program, `${MY_LOCAL}/main_programs/smoothy_calcobs.cc`

```
int main(int argc,char *argv[]){
    if(argc!=2){
        CLog::Info("Usage smoothy_calcobs emulator parameter filename");
        exit(1);
    }
    CparameterMap *parmap=new CparameterMap();
    parmap->ReadParsFromFile(string(argv[1]));
    CSmoothMaster master(parmap);
    // Reads Emulator Coefficients for all observables
    master.ReadCoefficientsAllY();
    master.priorinfo->PrintInfo();
    //modpars carries info about single point
    CModelParameters *modpars=new CModelParameters(master.priorinfo);
    // Prompt user for model parameter values
    vector<double> X(modpars->NModelPars);
    for(int ipar=0;ipar<modpars->NModelPars;ipar++){
        cout << "Enter value for " << master.priorinfo->GetName(ipar) << ":\n";
        cin >> X[ipar];
    }
    modpars->SetX(X);
    //  Calc Observables Y[iy] for X
    CObservableInfo *obsinfo=new CObservableInfo("Info/Observable_Info.txt");
    vector<double> Y(obsinfo->NObservables);
    vector<double> SigmaY(obsinfo->NObservables);
    master.CalcAllY(modpars,Y,SigmaY);
    for(int iY=0;iY<obsinfo->NObservables;iY++){
        cout << obsinfo->GetName(iY) << " = " << Y[iY] << " +/- " << SigmaY[iY] << endl;
    }
    return 0;
```

The User can hopefully use this template programs as a base for calling *Smooth Emulator* to calculate the emulator values for a specified point in space. Note that `SigmaY` is the emulator uncertainty, not that from experiment or from the theoretical model.

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_BAND_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_BAND/local/build` and compile/install the program
`GITHOME_BAND/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_BAND/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

A template project directory is provided that the User may copy to their own space, then use this as a foundation from which to embark on their own analysis. This directory includes information files, describing the parameter priors and the observables, that correspond to an artificial model that is also provided as a template. Working through the steps in this section constitutes a tutorial, both for running *Simplex Sampler* and for running *Smooth Emulator*.

This section describes the steps of how the User would

1. Copy the required files from the template directory to the User's space, and compile the main programs.
2. Set up the information files describing the priors and observable names.
3. Run *Simplex Sampler* to generate the model-parameter values at which the full model will be trained.
4. Run a "fake" full model to generate the observables for each of the full-model runs.
5. Tune *Smooth Emulator* and write the coefficients to file.
6. Run a program that prompts the User for the coordinates of a point in parameter space, then returns the emulator prediction with its uncertainty.

8.2 Installation and Compilation

After completing the necessary prerequisites listed in section ??[Installation] and following the steps outlined in section ??[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 ??[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 ??[Downloading] and [Compiling Libraries].

Then, the user can establish a personalized project directory by duplicating the project_template directory onto their computer. The User should copy the directories `GITHOME_BAND_SMOOTH/templates/mylocal` and `GITHOME_BAND_SMOOTH/templates/myproject` to a location in their personal space. We will refer to the User's two new directories as `${MY_LOCAL}/` and `${MY_PROJECTS}/`. For the purpose of this tutorial, the User must compile three main programs. This requires first changing into the `${MY_LOCAL}/main_programs/` directory and entering:

```
${MY\_LOCAL}/main_programs% cmake .
${MY\_LOCAL}/main_programs% make simplex
${MY\_LOCAL}/main_programs% make smoothy_tune
${MY\_LOCAL}/main_programs% make smoothy_calcobs
```

The reason these are compiled in the User's space, separate from the main libraries, is that the User may well wish to create their own main programs, and this arrangement allows the User to compile their own versions, while leaving the original programs from the templates directory unchanged.

For the purpose of the tutorial, there are also some "fake" models included in the distribution. For the User's project the fake model, which is very fast numerically, will be replaced by their own numerically intensive model. To compile the fake model used in the tutorial the User should change into the `${MY_LOCAL}/main_programs/` directory and enter:

```
${MY\_LOCAL}/fakemodels% cmake .
${MY\_LOCAL}/fakemodels% make fakerhic
```

This particular fake model has six model parameters and six observables, all with names in common use by the RHIC community. The output has absolutely no physical motivation, other than providing some arbitrary functions to emulate. The executable should appear in `${MY_LOCAL}/bin/`.

8.3 Creating Necessary Info Files

The User will run the software from the `${MY_PROJECTS}/` directory. Before a User can run *Simplex Sampler* they must create information files that describe the model-parameter priors and list the observable names. Both files are in the `${MY_PROJECTS}` directory. The first file is `${MY_PROJECTS}/Info/modelpa`
For the purposes of this tutorial, a file already exists,

```
compressibility        uniform  150   300
etaovers               uniform  0.05  0.32
initial_flow           uniform  0.3   1.2
initial_screening      uniform  0.0   1.0
quenching_length       uniform  0.5   2.0
initial_epsilon        uniform  15.0  30.0
```

This implies that the model has four parameters. The names, without much inspiration, are `par1`, `par2`, `par3` and `par4`. These names would normally be more descriptive, e.g. `NuclearCompressibility`. The second entry in each line is either `uniform` or `gaussian`. If the parameter is `uniform`, the last two numbers represent the range of the uniform prior, $\boldsymbol{x}_{\mathbf{min}}$ and $\boldsymbol{x}_{\mathbf{max}}$. If the second entry is `gaussian` the third entry represents the center of the Gaussian distribution and the fourth represents the width. For a real model, the User would replace this model with one appropriate for their own model.

The second file is `${MY_PROJECTS}/Info/observable_info.txt`. This describes output values from the model. In the template the file is

```
meanpt_pion    100
meanpt_kaon    200
meanpt_proton  300
Rinv           1.0
v2             0.2
RAA            0.5
```

The first entry in each line simply provides the names of the observable which will be processed in the Bayesian analysis. The second entry is used by `Smooth Emulator` during tuning, but only if a Monte Carlo method is used, and then is only used to seed the Monte Carlo search. If the analytical method is used for tuning (which is recommended) this parameter is irrelevant.

8.4 Running *Simplex Sampler*

Both *Simplex Sampler* and *Smooth Emulator* have options. These are provided in parameter files. For this tutorial, the provided parameter file is `${MY_PROJECTS}/parameters/simplex_parameters.txt`. The provided file is

```
#Simplex_LogFileName    simplexlog.txt # comment out to direct output to screen
Simplex_TrainType       2              # Must be 1 or 2
Simplex_ModelRunDirName modelruns      # Directory with training pt. info
```

Because the first line is commented, the output of *Simplex Sampler* will be to the screen. Otherwise it would go to the specified file. By setting `Simplex_TrainType=1`, the sampler will choose $\boldsymbol{n+1}$ training points, where $\boldsymbol{n=4}$ is the number of model parameters. Each point corresponds to the vertices of an $\boldsymbol{n+1}$ dimensional simplex. Finally, the parameter `Simplex_ModelRunDirName` is set to "`modelruns`". This informs `Simplex Sampler` to write the coordinates of each training point and the corresponding observables in the directory `${MY_PROJECTS}/rhic/modelruns/`.

Now the user can run `Simplex Sampler`, which must be run from the project directory. The only output is the number of training points.

```
${MY_PROJECTS}/rhic% ${MY_LOCAL}/bin/simplex
NTrainingPts=28
```

If one had set `Simplex_TrainType=1`, only seven training points would have been created. The programs writes information about the training points in the `modelruns/` directory. Changing into that directory, there should now be 28 sub-directories, corresponding to the 28 training points: `modelruns/run0`, `modelruns/run1`, `modelruns/run2`, `modelruns/`$\cdots$. Each directory has one text file describing the training points. For example, the `modelruns/run0/mod_parameters.txt` file might be

```
compressibility 190.282
etaovers 0.14892
initial_flow 0.664958
initial_screening 0.426807
quenching_length 1.16036
initial_epsilon 21.7424
```

This describes the six model parameters, which will serve as the input for the first full model run. The next step will be to run the full model for the parameters in each directory. Thus for `Simplex_Traintype=1`, one would need 7 full-model runs, and for `Simplex_Traintype=2`, one would need to do 28 full-model runs. The corresponding observables will be written in the files `modelruns/runI/obs.txt`

8.5 Running the Fake Full Model

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. For the tutorial, a fake model is provided. It reads the model-parameter values in each `modelruns/runI/mod_parameters.txt` file and writes the corresponding observables in `modelruns/runI/obs.txt`. The output should be as follows:

```
${MY_PROJECTS}/rhic% ${MY_LOCAL}/bin/fakerhic
NTraining Pts=28
NPars=6
```

The output simply verifies the number of model parameters and the number of training points created by simplex.

Inspecting the `modelruns/run0/obs.txt` file,

```
meanpt_pion   418.821195  1.000000
meanpt_kaon   715.592889  2.000000
meanpt_proton 1079.482871 3.000000
Rinv          5.004248    0.010000
v2            0.178353    0.002000
RAA           0.553416    0.005000
```

The second entry of each line is the value of the specified observable for that specific training point. The last entry is the random uncertainty associated with the full model. This is only relevant if the model has random fluctuations, meaning the re-running the model at the same point might result in different output. For this tutorial, the emulator will not consider such fluctuations (there is an emulator parameter that can be set to either consider the randomness or ignore it), so the third entry on each line is superfluous.

8.6 Running *Smooth Emulator*

To tune the emulator, the User will run `${MY_LOCAL}/bin/SmoothEmulator_tune` which should have been compiled in the directions above. The User needs to edit one additional file a this point, the parameter file that sets numerous options for *Smooth Emulator*. For the template used in this tutorial, that file is

```
#SmoothEmulator_LogFileName smoothlog.txt # comment out for interactive running
  SmoothEmulator_LAMBDA 2.0 # smoothness parameter
  SmoothEmulator_MAXRANK 5
  SmoothEmulator_ConstrainA0 false
  SmoothEmulator_ModelRunDirName modelruns
  SmoothEmulator_TrainingPts 0-27
  SmoothEmulator_UsePCA   false
  SmoothEmulator_TuneExact true
```

```
#
# These are only used if you are using MCMC tuning rather than Exact method
 SmoothEmulator_TuneChooseMCMC false # set false if NPars<5
 SmoothEmulator_TuneChooseMCMCPerfect false #
 SmoothEmulator_MCMC_NASample 8  # No. of coefficient samples
 SmoothEmulator_MCStepSize 0.01
 SmoothEmulator_MCMC_CutoffA false # Used only if SigmaA constrained by SigmaA0
 SmoothEmulator_MCSigmaAStepSize 1.0  #
 SmoothEmulator_MCMCUseSigmaY false # If false, also varies SigmaA
 SmoothEmulator_MCMC_NMC 20000  # Steps between samples
#
# This is for the MCMC search of parameter space (not for the emulator tuning)
MCMC_METROPOLIS_STEPSIZE 0.01
```

The parameters are described in detail in Sec. ??. Because `SmoothEmulator_TuneExact` is set to `true`, the Monte Carlo methods are not invoked and none of the parameters with `MCMC` in their names are relevant. The most relevant parameter is setting the smoothness parameter. Also, it is important to make sure that `SmoothEmulator_TrainingPts` is set to the correct number of training points. The Constrain A0 parameter decides where the first term of the Taylor expansion is used to estimate the variance of the coefficients, which then affects the emulator's estimate of its uncertainty.

Now, running `smoothy_tune`, produces the following output,

```
${MY_PROJECTS}/rhic% ${MY_LOCAL}/bin/smoothy_tune
 ---- Tuning for meanpt_pion ----
 ---- Tuning for meanpt_kaon ----
 ---- Tuning for meanpt_proton ----
 ---- Tuning for Rinv ----
 ---- Tuning for v2 ----
 ---- Tuning for RAA ----
```

The program generates Taylor coefficients which are saved in the `coefficients/` directory. Each observable has its own sub-directory with its name. In this case, `smoothy_tune` created the directories, `coefficients/rhic/RAA`, `coefficients/Rinv`, `coefficients/menapt_kaon`, `coefficients/meanpt_pion`, `coefficients/meanpt_proton` and `coefficients/v2`. Within each of these sub-directories `smoothy_tune` created files `meta.txt`, `ABest.txt` and `BetaBest.txt`.The number or parameters, the maximum rank of the Taylor expansion and the overall number of Taylor coefficients are give in `meta.txt`. The file `ABest.txt` provides the actual coefficients of the Taylor expansion, and `BetaBest.txt` gives an array used to calculate the uncertainty. If one of the Monte Carlo methods is chosen, rather than the default analytic tuning method, the file BetaBest.txt is replaced by several files, `sample0.txt, sample1.txt···`, which provide several samples of Taylor coefficients. For the tutorial, the parameter file `parameters/emulator_parameters.txt` has the parameters set to use apply analytic tuning rather than Monte Carlo tuning.

9 Testing the Emulator at the Training Points

Smooth Emulator should return the training values at the training points. If one runs the executable `smoothy_train_test`, it will first read in the coefficient information along with the training information. The program then emulates the model at the training points and compares the emulated value to the training value. Running the program gives the output:

```
${MY_PROJECTS}/rhic% ${MY_LOCAL}/bin/smoothy_train_test
 --- TESTING AT TRAINING POINTS ----
 ------ itrain=0 --------
 Y[0]= 4.188e+02 =?  4.188e+02,    SigmaY_emulator= 1.78365e-07
 Y[1]= 7.156e+02 =?  7.156e+02,    SigmaY_emulator= 2.81059e-07
 Y[2]= 1.079e+03 =?  1.079e+03,    SigmaY_emulator= 4.08783e-07
 Y[3]= 5.004e+00 =?  5.004e+00,    SigmaY_emulator= 3.15227e-09
 Y[4]= 1.784e-01 =?  1.784e-01,    SigmaY_emulator= 1.08732e-10
 Y[5]= 5.534e-01 =?  5.534e-01,    SigmaY_emulator= 4.26570e-10
 ------ itrain=1 --------
 Y[0]= 4.744e+02 =?  4.744e+02,    SigmaY_emulator= 1.82174e-07
 Y[1]= 7.156e+02 =?  7.156e+02,    SigmaY_emulator= 2.87061e-07
 Y[2]= 1.066e+03 =?  1.066e+03,    SigmaY_emulator= 4.17513e-07
 Y[3]= 5.004e+00 =?  5.004e+00,    SigmaY_emulator= 3.21959e-09
 Y[4]= 1.784e-01 =?  1.784e-01,    SigmaY_emulator= 1.11054e-10
 Y[5]= 5.533e-01 =?  5.533e-01,    SigmaY_emulator= 4.35679e-10
 ------ itrain=2 --------
 Y[0]= 4.437e+02 =?  4.437e+02,    SigmaY_emulator= 3.01087e-07
 Y[1]= 7.846e+02 =?  7.846e+02,    SigmaY_emulator= 4.74437e-07
 Y[2]= 1.073e+03 =?  1.073e+03,    SigmaY_emulator= 6.90041e-07
 Y[3]= 5.004e+00 =?  5.004e+00,    SigmaY_emulator= 5.32114e-09
 Y[4]= 1.784e-01 =?  1.784e-01,    SigmaY_emulator= 1.83543e-10
 Y[5]= 6.175e-01 =?  6.175e-01,    SigmaY_emulator= 7.20065e-10
 ------ itrain=3 --------
 Y[0]= 4.457e+02 =?  4.457e+02,    SigmaY_emulator= 2.47694e-07
 Y[1]= 6.842e+02 =?  6.842e+02,    SigmaY_emulator= 3.90304e-07
 Y[2]= 1.182e+03 =?  1.182e+03,    SigmaY_emulator= 5.67674e-07
```

⋮ The observables, $\boldsymbol{Y}[\mathbf{0}]\cdots\boldsymbol{Y}[\mathbf{27}]$ should be identical and the uncertainties at the training points should be zero. The fact that the uncertainties are not exactly zero derives from the numerical accuracy of the linear algebra routines.

10 Generating Emulated Observables at Given Points

Finally, now that the emulator is tuned, one may wish to generate emulated values for the observables for specified points in model-parameter space. A sample program, `${MY_LOCAL}/bin/smoothy_calcobs` is provided to illustrate how this can be accomplished. If one invokes the executable, using the same parameters as those used by `smoothy_tune`, the User is prompted to enter the coordinates of a point in model-parameter space, after which `smoothy_calcobs` prints out the observables. In this case, for the case where `compressibility=205`, `etaovers=0.2`, `initial_flow=0.7`, `initial_screening=0.4`, `quenching_length=1.2` and`initial_epsilon=23.0`

```
${MY_PROJECTS}/rhic% ${MY_LOCAL}/bin/smoothy_calcobs
 Prior Info
 #   ParameterName Type   Xmin_or_Xbar  Xmax_or_SigmaX
  0: compressibility     uniform        150        300
  1: etaovers            uniform        0.05       0.32
  2: initial_flow        uniform        0.3        1.2
  3: initial_screening   uniform          0          1
  4: quenching_length    uniform        0.5          2
  5: initial_epsilon     uniform         15         30
 Enter value for compressibility:
 205
 Enter value for etaovers:
 .2
 Enter value for initial_flow:
 .7
 Enter value for initial_screening:
 0.4
 Enter value for quenching_length:
 1.2
 Enter value for initial_epsilon:
 23.0
 ---- EMULATED OBSERVABLES ------
 meanpt_pion = 425.843 +/- 2.15299
 meanpt_kaon = 747.019 +/- 3.39257
 meanpt_proton = 1084.91 +/- 4.93428
 Rinv = 5.17076 +/- 0.03805
 v2 = 0.181905 +/- 0.00131247
 RAA = 0.59458 +/- 0.00514898
```

Note that the uncertainties for the emulation are not effectively zero, as each set of the 8 sets of coefficients provides an an emulator that exactly reproduces the training points.

Of course, it is unlikely the User will wish to enter model parameters interactively as was done above. To incorporate `Smooth Emulator` into other programs, the User should inspect the main programs, e.g. `${MY_LOCAL}/main_programs/smoothy_calcobs_main.cc`. The User can then design their own program based on this source code, and compile and link it by editing `${MY_LOCAL}/main_programs/CMakeList` By editing the CMake file, replacing the lines unique to `smoothy_calcobs`, one can easily compile new executables based on the User's main programs. To understand what might be involved, the source code in `${MY_LOCAL}/main_programs/SmoothEmulator_calcobs_main.cc` is

```
#include "msu_smoothutils/parametermap.h"
```

```
#include "msu_smooth/master.h"
#include "msu_smoothutils/log.h"
using namespace std;
int main(){
   NMSUUtils::CparameterMap *parmap=new CparameterMap();
   parmap->ReadParsFromFile("parameters/emulator_parameters.txt");
   NBandSmooth::CSmoothMaster master(parmap);
   master.ReadCoefficientsAllY();
   NBandSmooth::CModelParameters *modpars=new NBandSmooth::CModelParameters(); // contains
   modpars->priorinfo=master.priorinfo;
   master.priorinfo->PrintInfo();

   // Prompt user for model parameter values
   vector<double> X(modpars->NModelPars);
   for(unsigned int ipar=0;ipar<modpars->NModelPars;ipar++){
      cout << "Enter value for " << master.priorinfo->GetName(ipar) << ":\n";
      cin >> X[ipar];
   }
   modpars->SetX(X);

   //  Calc Observables
   NBandSmooth::CObservableInfo *obsinfo=master.observableinfo;
   vector<double> Y(obsinfo->NObservables);
   vector<double> SigmaY(obsinfo->NObservables);
   master.CalcAllY(modpars,Y,SigmaY);
   cout << "---- EMULATED OBSERVABLES ------\n";
   for(unsigned int iY=0;iY<obsinfo->NObservables;iY++){
      cout << obsinfo->GetName(iY) << " = " << Y[iY] << " +/- " << SigmaY[iY] << endl;
   }

   return 0;
}
```

This illustrates how one can write a code that

a) Reads the parameter file
b) Creates a *master* emulator file (called master because it includes an emulator for each observable)
c) Creates a model-parameters object, `modpars`, that stores the coordinates of the model-parameter point
d) Reads in the model parameters interactively
e) Calculates the observables from the emulator
f) Prints out the emulated observable and the uncertainty for for the emulator

9 Underlying Theory of *Smooth Emulator*

The choice of model emulators, $E(\vec{\theta})$, depends on the prior understanding of the model being emulated, $M(\vec{\theta})$. 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:

- $E(\vec{\theta}_t) = M(\vec{\theta}_t)$ at the training points, $\vec{\theta}_t$.
- 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, $\theta_i,\ i = 1 \cdots N$
- 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 $E(\vec{\theta})$ is chosen,

$$E(\vec{\theta}) = \sum_{\vec{n},\text{s.t. } K(\vec{n}) \leq K_{\max}} d_{\vec{n}} f_{K(\vec{n})}(|\vec{\theta}|) A_{\vec{n}} \left(\frac{\theta_1}{\Lambda}\right)^{n_1} \left(\frac{\theta_2}{\Lambda}\right)^{n_2} \cdots \left(\frac{\theta_N}{\Lambda}\right)^{n_N}. \tag{9.1}$$

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

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

The variance, σ_K^2, is allowed to vary as a function of K.

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

First, we consider the variance of the emulator at a given point, $\vec{\theta}$. Requiring that the variance is independent of the direction of $\vec{\theta}$ will fix $d_{\vec{n}}$. For example, transforming θ_1 and θ_2 to parameters $(\theta_1 \pm \theta_2)/\sqrt{2}$ should not affect the accuracy or uncertainty of the emulator.

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

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

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

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

Choosing $f_{K=0}(0) = 1$ and $d_{n_i=0} = 1$, the variance of E is indeed σ_0^2. The variance at some point $\vec{\theta} \neq 0$ is

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

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

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

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

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

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

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

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

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

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

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

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

To this point, the form is completely general once one requires that the variance above is independent of the direction of $\vec{\theta}$. I.e. the function $f_K(\vec{\theta})$ could be any function satisfying the constraint, $f_K(0)=1$, and σ_K^2 could have any function of K. Below, we illustrate how different choices for f or for σ_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 $\vec{\theta}$, the general form going forward is

$$\begin{aligned} E(\vec{\theta}) &= \sum_{\vec{n},{\rm s.t.}\;K(\vec{n})<K_{\rm max}} f_K(|\vec{\theta}|)\left(\frac{K(\vec{n})!}{n_1!\cdots n_N!}\right)^{1/2} A_{\vec{n}}\left(\frac{\theta_1}{\Lambda}\right)^{n_1}\left(\frac{\theta_2}{\Lambda}\right)^{n_2}\cdots\left(\frac{\theta_N}{\Lambda}\right)^{n_N}, \\ P(\vec{A}_n) &= \frac{1}{(2\pi\sigma_K^2)^{1/2}}e^{-|A_{\vec{n}}|^2/2\sigma_K^2}. \end{aligned} \tag{9.11}$$

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

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

$$\begin{aligned} E(\vec{\theta}) &= \sum_{\vec{n},K(\vec{n})\leq K_{\rm max}} \frac{1}{\Lambda^K}\frac{A_{\vec{n}}}{\sqrt{n_1!n_2!\cdots n_N!}}\theta_1^{n_1}\cdots\theta_N^{n_N}, \\ P(A_{\vec{n}}) &\sim e^{-A_{\vec{n}}^2/2\sigma^2}. \end{aligned} \tag{9.12}$$

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

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

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

Variant A: Letting σ_K have a K dependence
One reasonable alteration to the default choice might be to allow the $K = 0$ term to take any value, i.e. $\sigma_{K=0} = \infty$, while setting all the other σ_K terms equal to one another. This would make sense if our prior expectation of smoothness meant that we expect the $K = 2$ terms to be less important than the $K = 1$ terms, by some factor $|\vec{\theta}|/\Lambda$, but that the variation of the $K = 1$ term is unrelated to the size of the $K = 0$ 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 $\Delta\vec{\theta}$
This form for f causes correlations to fall for points far removed from one another.

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

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

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

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

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

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

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

In this limit one can also see that

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

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

9.3 Tuning the Emulator via MCMC

Here, we illustrate how an emulator of the form above can be constrained given training points. We consider $N_{\rm train}$ full model runs at positions $\vec{\theta}_a,\ a = 1, N_{\rm train}$, with values Y_a. The functional form has a large number of coefficients, $A_{\vec{n}}$. However, the dependence of A is purely linear. One can denote the coefficients as A_i with $i = 1 \cdots N_{\rm coef}$, where $N_{\rm coef} >> N_{\rm train}$. One can randomly set the coefficients $A_i,\ i = N_{\rm train} + 1 \cdots N_{\rm coeff}$ then determine the first $N_{\rm train}$ coefficients by solving a linear equation. One can then apply a weight based on the values of A consistent with the prior likelihood of A and the constraints. One can then generate a representative set of 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 $N_{\rm sample}$ sets of coefficients can be used to make a prediction for the emulator at some point $\vec{\theta}$, and the variance of those $N_{\rm sample}$ 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, σ.

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

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

If we also vary σ, we can choose a prior probability for σ. E.g.

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

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 Γ. One can also choose a flat distribution. The joint probability is $Q(\sigma) \prod_c P(A_c)$.

The constrained probability is

$$\begin{aligned} dP &= d\sigma Q(\sigma) \prod_{c=1}^{N_{\rm coef}} \left[dA_c\, P(A_c)\right] \prod_{m=1}^{N_{\rm train}} \delta(E(A, \sigma, \vec{\theta}_m) - F(\vec{\theta}_m)) \qquad (9.19) \\ &= d\sigma Q(\sigma) \prod_{c \leq N_{\rm train}} P(A_c) \frac{1}{|J|} \prod_{c=N_{\rm train}}^{N_{\rm coef}} \left[dA_c\, P(A_c)\right]. \end{aligned}$$

Here, $|J|$ is the Jacobian, i.e. it is the determinant of the $N_{\rm train} \times N_{\rm train}$ matrix

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

For the default form in the previous section,

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

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

One can now sample A and σ according to the weights above. Here, we list two methods.

a) **Keep and Reject Method**
One can generate the coefficients, $A_i,\ i > N_{\rm train}$, according to the Gaussian distribution with width σ, after generating σ according to the Lorentzian. The residual weight is then

$$w = \prod_{c=1}^{N_{\rm coef}} P(A_c). \tag{9.22}$$

For each attempt, one could keep or reject the attempt with probability w. This generates perfectly independent samples, but with the caveat that in a high dimensional space the rejection level becomes untenably high.

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

$$w = Q(\sigma)\prod_c P(A_c). \tag{9.23}$$

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, $N_{\rm steps}$, 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.

9.4 Exact Solution for the Most Probable Coefficients and the Uncertainty

Here, we demonstrate how one can solve for the set of most probable coefficients, $A_{\vec{n}}$, given the the training values, $y_t(\vec{\theta}_t)$, where $\vec{\theta}_t$ are the points at which the emulator was trained. Again, we consider $N_{\rm train}$ to be the number of training points and $N_{\rm coef}$ to be the number of coefficients, which is much larger than $N_{\rm train}$. Given the coefficients, $\vec{A}_i$ for $i > N_{\rm train}$, the first coefficients, $i = 1\cdots N_{\rm train}$, are found by solving the linear equations for $A_a, a \leq N_{\rm train}$. For shorthand, we define the function $T_a(\vec{\theta})$,

$$E(\vec{\theta}) \;=\; \sum_{i=1}^{N_{\rm coeff}} A_i\mathcal{T}_i(\vec{\theta}), \tag{9.24}$$

The functions $\mathcal{T}_i(\vec{\theta})$ reference the factors in Eq. (??) if the default form is used. Evaluated at the $N_{\rm train}$ training points, we define a $N_{\rm train} \times N_{\rm coef}$ matrix,

$$T_{ai} \equiv \mathcal{T}_i(\vec{\theta}_a), \tag{9.25}$$

where $\vec{\theta}_a$ labels the training points, $a \leq 0 < N_{\rm train}$, and $N_{\rm train} << N_{\rm coef}$. Further, moving forward it is convenient to define the $N_{\rm train} \times N_{\rm train}$ sub-matrix,

$$\tilde{T}_{ab} \;=\; T_{ab},\;\; a, b < N_{\rm train}. \tag{9.26}$$

Given the location of the training points, one knows T_{ta}. One can then solve the linear system of equations to find $A_{a\leq N_{\rm train}}$ if one knows $y_{t\leq N_{\rm train}}$. To fit the training points,

$$A_a \;=\; \sum_t \tilde{T}^{-1}_{at}\left(y_t - \sum_{i>N_{\rm train}} T_{ti}A_i\right), \tag{9.27}$$

for $a = 1\cdots N_{\rm train}$. As mentioned above, the matrix $\tilde{T}_{ab}$ is a $N_{\rm train}\times N_{\rm train}$ square matrix. For the purposes of brevity going forward, we define the vector $\alpha_a, a \leq N_{\rm train}$ and $\beta_{a,i},\ i > N_{\rm train}$, so that After making the following definitions,

$$\begin{aligned}\alpha_a &\equiv \sum_{b\leq N_{\rm train}} \tilde{T}^{-1}_{ab} y_b, \\ \beta_{ai} &\equiv \sum_{t'\leq N_{\rm train}} \tilde{T}^{-1}_{at'} T_{t'i},\end{aligned} \tag{9.28}$$

the expression needed to solve for $A_{a\leq N_{\rm train}}$ can be expressed as

$$A_a \;=\; \alpha_a - \sum_{i>N_{\rm train}} \beta_{ai}A_i. \tag{9.29}$$

All the dependence on the full model is contained within α_a, whereas β_{ai} depends only on the positions of the training points.

9.4.1 Finding the Optimum Coefficients

The goal is to find the coefficients, $A_{i>N_{\rm train}}$, that maximizes the probability given that the first $N_{\rm train}$ coefficients are determined by Eq. (??),

$$\begin{aligned}P(\vec{A}) &= \frac{1}{(2\pi\sigma^2)^{N_{\rm coef}/2}} \int \prod_{t\leq N_{\rm coef}} \delta[y_m(\vec{\theta}_t) - E(\vec{\theta}_t)] \prod_i dA_i\ \exp\left\{-\frac{1}{2\sigma^2}A_i^2\right\} \\ &= \frac{1}{(2\pi\sigma^2)^{N_{\rm coef}/2}} \int e^{-\sum_j A_j^2/2\sigma^2}\frac{1}{|J|}\prod_{i>N_{\rm train}} dA_i.\end{aligned} \tag{9.30}$$

Here, $|J|$ is the Jacobian, i.e. it is the determinant of the $N_{\rm train}\times N_{\rm train}$ matrix

$$J_{mc} = \frac{\partial E(\vec{\theta}_m)}{\partial A_c}, \tag{9.31}$$

For the default form in the previous section,

$$J_{mc} \;=\; \tfrac{1}{\sqrt{n_{c1}!\cdots n_{cN}!}}(\theta_{m1})^{n_{c1}}\cdots(\theta_{mN})^{n_{cN}}. \tag{9.32}$$

Because the Jacobian depends only on the position of the training points, it can be treated as a constant. As was discussed earlier, the Jacobian depends on the positions of the training points, $\vec{\theta}_t$, but does not depend on $\vec{A}$. In this case we will also fix σ, so we need merely minimize the argument of the exponential. The argument of the exponential, which is to be minimized, then becomes

$$\begin{aligned} \mathrm{MIN} &= \sum_{i>N_{\rm train}} A_i^2 + \sum_{a\leq N_{\rm train}} \left[\sum_t \tilde{T}^{-1}_{at}\left(y_t - \sum_{i>N_{\rm train}} T_{ti}A_i\right)\right]^2 \qquad (9.33)\\ &= \sum_{i>N_{\rm train}} A_i^2 + \sum_{a\leq N_{\rm train}} \left[\alpha_a - \sum_{i>N_{\rm train}} \beta_{ai}A_i\right]^2 \end{aligned}$$

One next needs to minimize this for each coefficient, $A_{i>N_{\rm train}}$. This gives

$$0 = A_i - \sum_{a\leq N_{\rm train}}\left[\alpha_a - \sum_{j>N_{\rm train}} \beta_{aj}A_j\right]\beta_{ai}. \qquad (9.34)$$

$$\sum_{a\leq N_{\rm train}} \alpha_a\beta_{ai} = \sum_{j>N_{\rm train}}\left(\delta_{ij} + \sum_{a\leq N_{\rm train}} \beta_{ai}\beta_{aj}\right)A_j \qquad (9.35)$$

This would be a straight forward solution, but if there were large numbers of parameters, one might have tens of thousands of coefficients. To avoid solving such a large number of simultaneous equations, one can realize that if one thinks of $A_{i>N_{\rm train}}$ as a high dimensional vector, the only other vectors in that space, i.e. those with index $i > N_{\rm train}$, that can represent the solution are the $N_{\rm train}$ vectors β_{ai}. Thus the solution should have the form

$$A_i = \sum_a \gamma_a\beta_{ai}. \qquad (9.36)$$

One can then solve for γ_a, which will require solving $N_{\rm train}$ simultaneous equations. For the sake of brevity, gall the indices labels, a, b, c or t will vary from $1 - N_{\rm train}$ and those labeled i, j will vary from $N_{\rm train} + 1 \cdots N_{\rm coef}$. One must minimize

$$\begin{aligned} \mathrm{MIN} &= \sum_i\left[\sum_a \gamma_a\beta_{ai}\right]^2 + \sum_a\left[\left(\alpha_a - \sum_i \beta_{ai}\sum_b \gamma_b\beta_{bi}\right)\right]^2 \qquad (9.37)\\ &= \sum_{ab}\gamma_a B_{ab}\gamma_b + \sum_a(\alpha_a - \sum_b B_{ab}\gamma_b)^2, \end{aligned}$$

where B_{ab} is defined.

$$B_{ab} \equiv \sum_i \beta_{ai}\beta_{bi}. \qquad (9.38)$$

Setting the derivative w.r.t. γ_a to zero,

$$\begin{aligned} 0 &= \sum_b B_{ab}\gamma_b + \sum_{bc} B_{ab}B_{bc}\gamma_b - B_{ab}\alpha_b, \qquad (9.39)\\ \alpha_a &= (\delta_{ab} + B_{ab})\,\gamma_b. \end{aligned}$$

Thus, once one has run the full model $N_{\rm train}$ times at the training points, one can find y_t and T_{ai}. Following the prescription above, one then finds α_a and β_{ai}, which then gives B_{ab}. One then solves Eq. (??) for γ_b, which when inserted into Eq. (??) gives the coefficients A_i for $i > N_{\rm train}$. To find the coefficients, A_a for $a \leq N_{\rm train}$, one then applies Eq. (??).

Note that this algorithm never involves a double sum with both indices over all $N_{\rm coef}$. Nor does it involve any matrix manipulations, or linear equation solving involving matrices of $N_{\rm coef} \times N_{\rm coef}$. Assuming that the number of training points is no more than a few hundred, this algorithm should be rather quick.

9.4.2 Finding the Uncertainty when Calculating $E(\vec{\theta})$

Next, one needs to express the uncertainty at a point in parameter space $\sigma_E^2(\vec{\theta}) = \vec{\theta}$, $\langle(\delta E(\vec{\theta}))^2\rangle$. This is given by

$$\sigma_E^2 = \frac{\partial E}{\partial A_i}\frac{\partial E}{\partial A_j}\langle \delta A_i \delta A_j\rangle. \tag{9.40}$$

Once again, the indices i, j refer to components with $i > N_{\rm train}$. We begin with

$$\begin{aligned} E(\vec{\theta}) &= T_a(\vec{\theta})A_a + T_i(\vec{\theta})A_i, \\ T_a(\vec{\theta}) &= \frac{1}{\Lambda^K}\frac{\theta_1^{n_{1a}}\theta_2^{n_{2a}}\cdots}{\sqrt{n_{1a}!n_{2a}!\cdots}}. \end{aligned} \tag{9.41}$$

If one chooses $\vec{\theta}$ equal to one of the training points, t, then $T_a(\vec{\theta}_t) = T_{ta}$. The differential of $E(\vec{\theta})$ is

$$\delta E(\vec{\theta}) = \sum_a T_a(\vec{\theta})\delta A_a + \sum_i T_i(\vec{\theta})\delta A_i. \tag{9.42}$$

Substituting for A_a in terms of A_a,

$$\delta E(\vec{\theta}) = \left[T_i(\vec{\theta}) - T_a(\vec{\theta})\beta_{ai}\right]\delta A_i. \tag{9.43}$$

Next, one needs to calculate $\langle \delta A_i \delta A_j\rangle$. Expanding around the minimum, the exponential of the probability becomes

$$\begin{aligned} P(\delta\vec{A}) &\propto \exp\left\{-\frac{1}{2\sigma^2}\left(\sum_a(\delta A_a)^2 + \sum_i(\delta A_i)^2\right)\right\} \\ &= \exp\left\{-\frac{1}{2\sigma^2}\sum_{ij}\left(\delta_{ij} + \beta_{ai}\beta_{aj}\right)\delta A_i \delta A_j\right\}. \end{aligned} \tag{9.44}$$

Defining

$$D_{ij} = \delta_{ij} + \sum_a \beta_{ai}\beta_{aj}, \tag{9.45}$$

the fluctuation of the coefficients is

$$\langle \delta A_i \delta A_j \rangle \;= D^{-1}_{ij}. \tag{9.46}$$

Again, if there are many coefficients, inverting the matrix could be numerically costly. But one can realize that the inverse matrix should be of the form

$$D^{-1}_{ij} \;=\; \delta_{ij} + \sum_{ab} \psi_{ab}\beta_{ai}\beta_{bj}, \tag{9.47}$$

and solve for the $N_{\rm train} \times N_{\rm train}$ values of ψ_{ab}. This leads to

$$D^{-1}_{ik}D_{kj} \;=\; \delta_{ij} + \sum_{ab} \beta_{ai}\beta_{bj}(\delta_{ab}+\psi_{ab}) + \sum_{abc} \beta_{ai}\beta_{ak}\beta_{ck}\beta_{bj}\psi_{cb}. \tag{9.48}$$

For this to be satisfied for any pair of i, j,

$$\delta_{ab} \;=\; -\psi_{ab} - \sum_c B_{ac}\psi_{bc}. \tag{9.49}$$

This gives

$$\begin{aligned} \psi_{ab} &= -Q^{-1}_{ab}, \\ Q_{ab} &= \delta_{ab} + B_{ab}. \end{aligned} \tag{9.50}$$

Finally, this gives

$$\langle (\delta y)^2 \rangle \;=\; \left\{ T_i(\vec{\theta}) - T_a(\vec{\theta})\beta_{ai} \right\} D^{-1}_{ij} \left\{ T_j(\vec{\theta}) - T_b(\vec{\theta})\beta_{bj} \right\}. \tag{9.51}$$

One can look at the first term in Eq. (??), $(T_i(\vec{\theta}) - T_a(\vec{\theta})\beta_{ai})$, and see that if θ is chosen to be one of the training points, that the resulting width will be zero. In that case, $T_a(\vec{\theta}_b)$ becomes the matrix element T_{ab}, and when multiplied by $\beta_{ai} = T^{-1}_{ab}T_{bi}$, becomes equal to T_{ai}. I.e.,

$$T_a(\vec{\theta}_b)\beta_{ai} \;= T_{ba}T^{-1}_{ac}T_{ci} = T_{bi}. \tag{9.52}$$

Inserting Eq. (??) and putting this all together, one has 8 terms altogether.

$$\sigma^2_E(\vec{\theta}) \;=\; \sigma^2_1 + \sigma^2_2 + \sigma^2_3 + \sigma^2_4 + \sigma^2_5 + \sigma^6_2 + \sigma^2_7 + \sigma^2_8. \tag{9.53}$$

The individual terms are

$$\begin{aligned} \sigma^2_1 &= T_i(\vec{\theta})T_i(\vec{\theta}), \\ \sigma^2_2 &= -T_a(\vec{\theta})\beta_{ai}T_i(\vec{\theta}) = -T_a(\vec{\theta})S_a(\vec{\theta}), \\ \sigma^2_3 &= \sigma^2_2, \\ \sigma^2_4 &= T_a(\vec{\theta})B_{ab}T_b(\vec{\theta}), \\ \sigma^2_5 &= T_i(\vec{\theta})\beta_{ai}\psi_{ab}\beta_{bj}T_j(\vec{\theta}) = S_a(\vec{\theta})\psi_{ab}S_b(\vec{\theta}), \\ \sigma^2_6 &= -T_a(\vec{\theta})B_{aa'}\psi_{a'b'}\beta_{b'j}T_j(\vec{\theta}) = -T_a(\vec{\theta})B_{aa'}\psi_{a'b'}S_{b'}(\vec{\theta}), \\ \sigma^2_7 &= \sigma^2_6, \\ \sigma^2_8 &= T_a(\vec{\theta})B_{aa'}\psi_{a'b'}B_{b'b}T_b(\vec{\theta}). \end{aligned} \tag{9.54}$$

Here, we have defined the quantity

$$S_a(\vec{\theta}) \equiv \beta_{ai} T_i(\vec{\theta}). \tag{9.55}$$

By first calculating $S_a(\vec{\theta})$, this allows one to avoid any sums over two indices i and j. One can also avoid expedite the calculations of σ_6^2 and σ_8^2 by calculating and storing

$$\begin{aligned} H_{ab}^{(6)} &= B_{aa'}\psi_{a'b}, & (9.56)\\ H_{ab}^{(8)} &= B_{aa'}\psi_{a'b'}B_{b'b}. & (9.57)\end{aligned}$$

These quantities need be calculated only once as they do not depend on $\vec{\theta}$. One then has

$$\begin{aligned} \sigma_6^2 &= -T_a(\vec{\theta})H_{ab}^{(6)}S_b(\vec{\theta}), & (9.58)\\ \sigma_8^2 &= T_a(\vec{\theta})H_{ab}^{(8)}T_b(\vec{\theta}). & (9.59)\end{aligned}$$

This avoids any $N_{\rm train}^3$ or $N_{\rm train}^4$ calculations. Thus, assuming $N_{\rm coef} >> N_{\rm train}$, the largest numeric penalty in calculating σ^2 is in the $N_{\rm train} \times N_{\rm coef}$ loop required to calculate $S_a(\vec{\theta})$ for all a.

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 $N_{\rm train} = N + 1$ points in parameter space placed far apart from another. One choice is the $N-$dimensional simplex. Examples are an equilateral triangle in two-dimensions or a tetrahedron in three dimensions. For a simplex, one places the $N_{\rm train} = N + 1$ training points at a uniform distance from the origin, r, with equal separation between each point. One can generate an $N-$dimensional simplex from an $N - 1$ dimensional arrangement. In the $N - 1$ dimensional arrangement, the points are arranged equidistant from one another using the coordinates $x_1 \cdots x_N$. The points would all be placed at a radius r_N from the origin in this system, and the separation would be d. In the $N-$dimensional system all these $N - 1$ points had coordinate $x_N = -a$. The $(N + 1)^{\rm th}$ point is then placed at position $x_1 \cdots x_N = 0$ and $x_{N+1} = Na$. This keeps the center of mass at zero. One then chooses a such that the new point is equally separated from all the other points by the same distance d,

$$\begin{aligned} d^2 = r_{N-1}^2 + N^2a^2, \qquad (10.1)\\ a = \sqrt{\frac{d^2}{N^2} - r_{N-1}^2}. \end{aligned}$$

Now, each of the N points is located a distance Na away from the center of the $N-$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 $N(N + 1)/2$ additional points, which is exactly the number of segments connecting the $N + 1$ equally-spaced vertices of the $N-$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 $N+1$ points positioned at the lower radius and $N(N+1)/2$ 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 $N_{\rm train} = N+1$ 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 $N+1$ training points.

One issue with the simplex is that the first set of $N+1$ training points would all be placed at the same radius. If the prior parameter distribution is uniform within an $N-$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, $-1 < \theta_i < 1$. If the number of parameters is N, the hyper-cube has $2N$ faces and 2^N corners, a face being defined as one parameter being ± 1 while the others are zero, while a corner has each parameter either ± 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 2^N, whereas the volume of an $N-$dimensional hyper-sphere of radius $R=1$ is

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

The solid angle, Ω_N in N dimensions is

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

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

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

In two dimensions, the ratio is $\pi/4$, and in three dimensions it is $\pi/6$. In 10 dimensions it is 2.5×10^{-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 $|\vec{\theta}|$ is $\sqrt{N}$. So, for 9 parameters, if the training points were all located at positions $|\vec{\theta}| < 1$, one would have to extrapolate all the way to $|\vec{\theta}| = 3$. 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 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 $2a$,

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

whereas for a sphere of radius a,

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

The ratio of the radii is then

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

Thus, in 10 dimensions, if the training points are placed at a distance 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, $a\sqrt{N/3}$, 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, Λ, 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 -0.1 and 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 is that narrows with increasing smoothness parameter as shown in Fig. ??. Figure **??** also shows

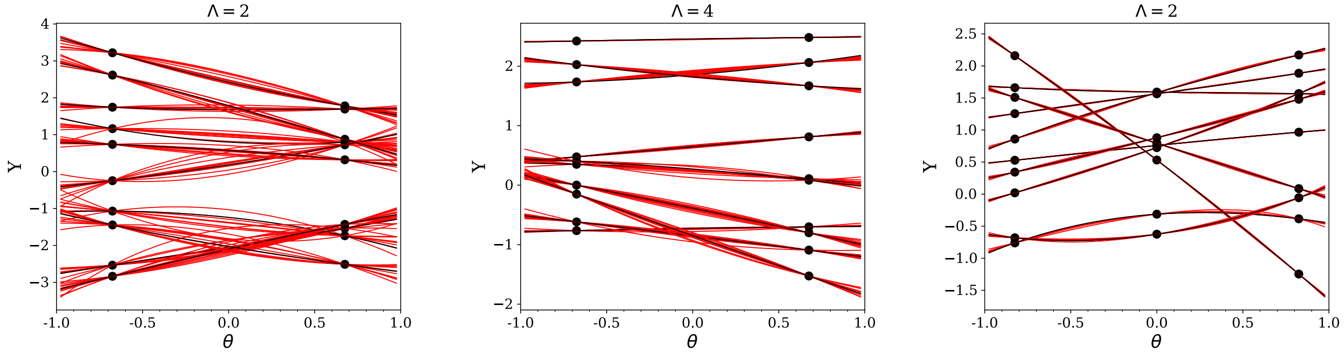

Figure 11.1: Real functions (black lines) are emulated with the *Smooth Emulator* (default settings). Ten different functions were generated sampled by the Monte Carlo method. The spread of the lines represents the uncertainty. The spread is reduced by either reducing the smoothness parameter, $\boldsymbol{\Lambda}$, or by increasing the number of sampling points.

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 $-1 < \theta_i < 1$. In this case, the *Simplex Sampler* was used to choose the training points. The "real" model, F, was constructed from sample smooth functions, with the coefficients generated randomly and a smoothness parameter set to three or six. The $A_{\vec{n}=0}$ 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 points in parameter space were chosen. The emulator value and its uncertainty are plotted alongside the real-model values in Fig. **??**. 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 **??** first shows the 100 comparisons for the case with the smoothness parameter, $\boldsymbol{\Lambda} = \mathbf{3}$. 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 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 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. **??**, 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. **??**, 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.

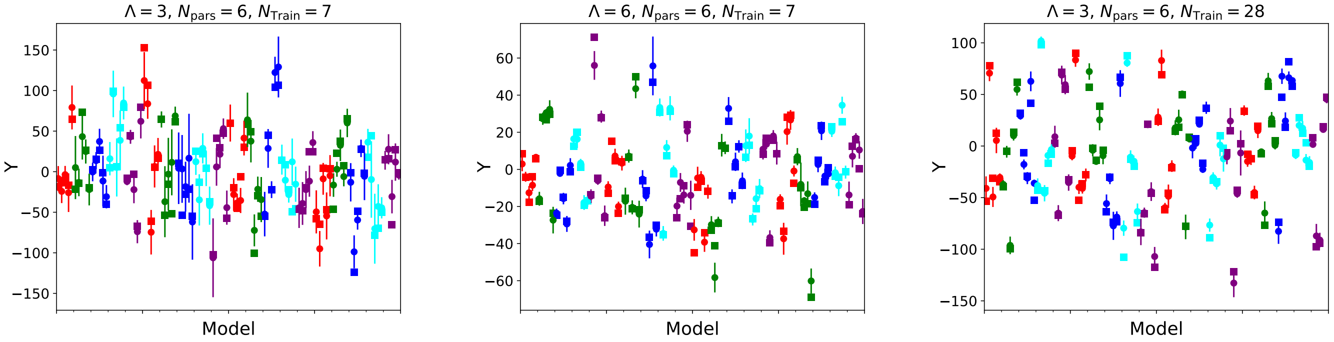

Figure 11.2: Using 20 instances of real models using six parameters, choosing 5 random points in parameter space for each model, the emulator (circles) and its uncertainty were compared to the real values (squares). Neighboring points of the same color emulated the same real model. The accuracy improves 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.