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

March 14, 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

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
```

Within the repository, there will be a directory `/Users/CarlosSmith/bandframework/software/SmoothEmulator/`. All the relevant functionality of *Smooth Emulator* is contained within this directory. Throughout the manual the phrase `${GITHOME_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_SMOOTH}/templates/myproject ${MY_PROJECT}
```

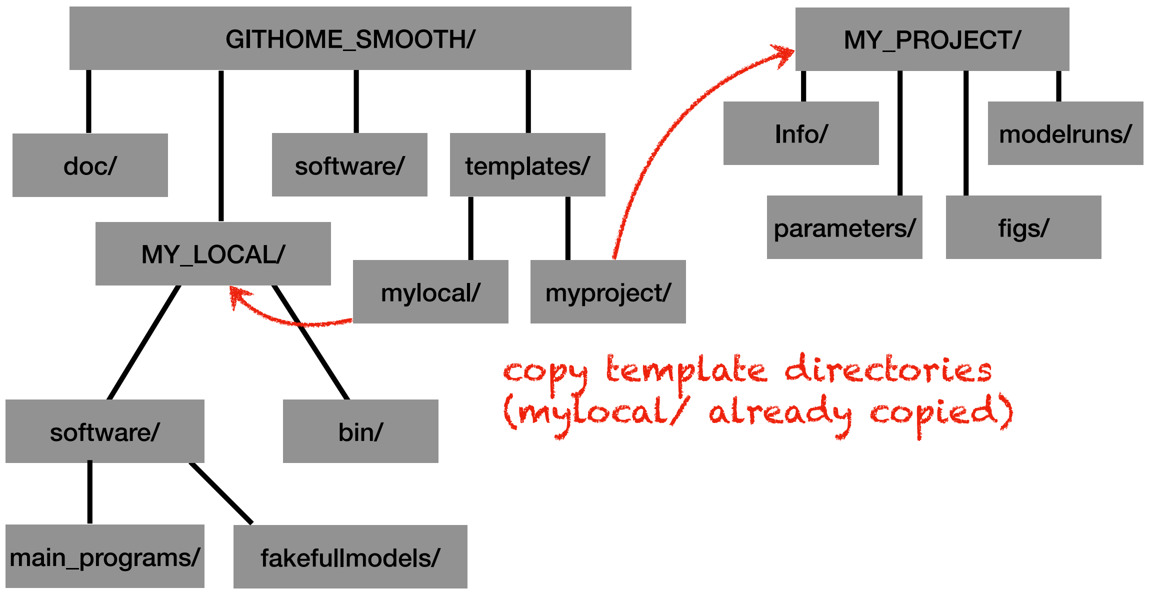


Figure 2.1: The directory structure: The User clones the repository into some location, which will be referred to as `${GITHOME_SMOOTH}`. The User can then copy the `${MY_PROJECT}` directory to the User's choices of locations outside the path of the BAND repository. The programs are designed to be run from within the `MY_PROJECT/` directory, and the User can make multiple copies of `${MY_PROJECT}` whenever they wish to run and save different instances of *Smooth Emulator*

.

Hence forth, `${MY_PROJECT}` is a name chosen by the User, and will refer to the directory, including the path, from which the User will perform most of the analysis. The User may wish to have several such directories, located according to the User's preference. Any time a new analysis is performed with new parameters, and if the User wishes to save the previous analysis, a new `${MY_PROJECT}` directory should be created. These directories should be outside the main distribution, i.e. outside the `bandframework/` path so that the User's work does not interfere with the repository.

Within the `templates/` directory, there is also a directory `../templates/mylocal/`. The directory `${GITHOME_SMOOTH}/mylocal/` is an exact copy. The `mylocal/` directory contains the main programs and the *Smooth Emulator* executables will be stored in `${GITHOME_SMOOTH}/mylocal/bin`. Because the User may well wish to edit the main programs, or to add similar programs, this provides the User a space to make such edits, all while leaving an original copy of the directory in the `templates/` directory.

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

2.3 Directory and File Structure

The `${GITHOME_SMOOTH}/software/` directory contains codes that are used to create libraries specific to the sampler and emulator. The User can change to The executables are stored in `${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}/main_programs/`. 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_SMOOTH}/software
${GITHOME_SMOOTH}/software% cmake .
${GITHOME_SMOOTH}/software% make
```

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

```
% 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. If the User wishes to choose a specific C++ compiler, the `cmake` command should be replaced with `cmake -D YOUR_PREFERRED_COMPILER`.

At this point all the libraries are built, but this does not include the main programs. The main programs are short, and are meant to serve as examples which the User might copy and edit at will. Below, this illustrates how to compile the programs used for generating training points with Simplex and for tuning the emulator with Smoothy:

```
% cd ${MY_LOCAL}/software
${MY_LOCAL}/software% cmake .
${MY_LOCAL}/software% make
.
.
```

The `cmake` command will also recompile the libraries in `${GITHOME_SMOOTH}/software/` if necessary. Several source codes for main programs can be found in `${MY_LOCAL}/software/main_programs/`. If you build your own main programs (probably using these as examples), you can edit the `CMakeList.txt` file in `${MY_LOCAL}/software/main_programs/`, using the existing entries as an example. The executables should appear in `${MY_LOCAL}/bin/`.

2.5 The `MY_PROJECT` Directory

Within `${MY_PROJECT}/` there are four sub-directories (assuming it was created from the template). The first is `${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`, $\cdots$, have files describing the model parameters for each run, along with the output required by the emulator for each specific full-model run. For example, the `${MY_PROJECT}/modelruns/run1/` directory has the files `mod_parameters.txt` and `obs.txt`. The first file stores the model parameter values for that particular training run. The User then runs their full model based on those parameters and stores the corresponding observables in `obs.txt`. The User may generate the `mod_parameters.txt` files using *Simplex Sampler*, or the user might generate them according to some other prescription. Once the User has then generated the `obs.txt` files, *Smooth Emulator* can then build and tune the emulator. Finally the `${MY_PROJECT}/figs` directory contains python scripts for plotting results.

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, or to use their own method to generate training 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 $N_{\rm train} = n+1+n(n+1)/2$. Thus, if the model is perfectly quadratic, this option provides perfect emulation. Rather arbitrarily, the points generated from the midpoints of the original simplex pairs are all pushed out to a large radius, and those from the original simplex are brought somewhat inward.

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$^{\mathrm{th}}$ 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.

3.6 Replacing *Simplex Sampler* with Other Choices of Training Points

If the User desires to use their own procedure for training points, or if the User wishes to augment the *Simplex Sampler* points with additional training points, the User can write their own files `runI/mod_parameters.txt`. The emulator should work fine, though the User should remember that when training the emulator (see Sec. 5) the emulator parameter describing with `runI` directories to apply should be modified.

There is one warning to be issued when choosing training points for tuning *Smooth Emulator*. When solving for the Taylor coefficients, the emulator uses the factor of those coefficients (products of $\boldsymbol{\theta_i}$) in the linear algebra routine. However, if those coefficients are not linearly independent, the solution becomes undetermined. For example, if two of the training points were exactly the same, it would fail.

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`, where $\boldsymbol{I = 0, 1, 2, \cdots}$. These files could have been generated by *Simplex Sampler*, but could have been generated by any other means, including by hand. The files should be of the form,

```
par1_name  par1_value
par2_name  par2_value
par3_name  par3_value
     ⋮
```

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

The User must then perform full model runs using the model-parameter values as defined in each sub-directory. The full model runs should record results by writing a list of observable values in each run directory. Each file must be named `${MODEL_RUN_DIRNAME}/runI/obs.txt`, where $\boldsymbol{I = 0, 1, 2, \cdots}$. 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. 5. The values are the observable values as calculated by the full model for the model-parameter values listed in the corresponding `mod_parameters.txt` file in the same directory.

The random uncertainties refer only to those uncertainties due to noise in the full model. Random noise is that, which if the full model would be rerun at the same model-parameter values, would represent the variation in the observable values. In most cases this would be set to zero. But, if the full model has some aspect of sampling to it, for example generating observables from event generators with a finite number of events, that variation should be listed here. This variation is required for the emulator. If there is such a variation, the 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:

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

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. 9, 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. 9. Whereas *Smooth Emulator* does a nice job of finding an optimum value for σ_A if Λ is known, 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 be tuned one of three different ways

a) Find the optimum set of coefficients. If evaluated at 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 full model at the training points. Aside from the constraint, the coefficients are chose randomly, but weighted according to Eq. (5.3). The User sets the number of sets of coefficients, typically of order $N_{\text{sample}} \approx 10$, 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_{sample} predictions.

c) The third mode also provides N_{sample} predictions, but rather than exactly reproducing the training values the emulator merely comes close to the training points with a distribution $\sim e^{-\Delta y^2/2\epsilon_y^2}$, where ϵ_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 N_{sample} sets of coefficients from the training data, then stores N_{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, σ_A. This file is named `Info/observable_info.txt`, where the path is relative to the project directory. The file might look like

```
 obsname1  SigmaA1_init
 obsname2  SigmaA2_init
 obsname3  SigmaA3_init
 obsname4  SigmaA4_init
     ⋮
```

   The initial σ_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 σ_A above when using one of the Monte Carlo methods.

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, $\vec{\theta}_{\rm train}$, at each training point. These points can be generated by *Simplex Sampler*, as described in Sec. 3, or they can be generated by hand. If the number of full-model runs performed is $N_{\rm train}$, Smooth emulator requires files for each run. Each file is named `${MODEL_RUN_DIRNAME}/runI/mod_parameters.txt`, where $0 \leq I < N_{\rm train}$, and I denotes the point in parameter space for the $I^{\rm th}$ 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.

5.3 Experimental Measurement Information

Once the emulator is tuned and before it is applied to a Markov Chain investigation of the likelihood, the software needs to 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.

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

Smooth Emulator requires a parameter file, `${MY_PROJECT}/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 random error
 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 $\boldsymbol{\Lambda}$. It sets the relative importance of terms of various rank. If $\boldsymbol{\Lambda}$ 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. 6.

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 N_{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 N_{sample} sets of coefficients. Each set reproduces the training points, but differs away from the training points. Setting $N_{\mathrm{sample}} \sim 10$ 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 θ 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 N_{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 full 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 A above:

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

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

5.5 Running *Smooth Emulator* Programs

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

To begin, we consider the source code `${MY_LOCAL}main_programs/smoothy_tune_main.cc`. Once compiled the corresponding executable is `${MY_LOCAL}/bin/smoothy_tune`. The source code for `smoothy_tune` is:

```
int main(){
  // First create CSmoothMaster
  CparameterMap *parmap=new CparameterMap();
  CSmoothMaster master(parmap);
  // Read in Training Info
  master.ReadTrainingInfo();
  // Tune for all observables "Y"
  master.TuneAllY();
  // Write coefficients to file
  master.WriteCoefficientsAllY();
  return 0;
}
```

From within the `${MY_LOCAL}/main_programs/` directory, one can compile the program with the command:

```
 MY_LOCAL/main\_programs % cmake .
 MY_LOCAL/main\_programs % make smoothy_tune
```

The executable `smoothy_tune` should now appear in the `${MY_LOCAL}/bin/` directory. If one enters `make` without the name of the program, all the main-program source files will be compiled. Assuming the directory `${MY_LOCAL}/bin/` has been added to the User's path, the User may switch to the User's project directory, and enter:

```
  ~/MY_PROJECT % smoothy_tune
```

This should read in all the necessary information, tune the emulators for all the observables and write the Taylor-expansion coefficients to file. The optimum coefficients are stored in the file `coefficients/OBS_NAME/ABest.bin`, where `OBS_NAME` is the observable name. If a Monte Carlo tuning method was applied, there are several sets of coefficients stored,
`coefficients/OBS_NAME/sampleI.bin`, where $\boldsymbol{0 \leq I < N_{\rm sample}}$. 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. For the non-Monte-Carlo method, files `coefficients/OBS_NAME/BetaBest.bin` are stored for later use in calculating the uncertainties.

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 specified points in parameter space:

```
int main(){
  // Create Smooth Master Object
  NMSUUtils::CparameterMap *parmap=new CparameterMap();
  NBandSmooth::CSmoothMaster master(parmap);
  // Read in coefficients
  master.ReadCoefficientsAllY();
  // Create a CModelParameters object to store information
    // about a single point in parameter space
  NBandSmooth::CModelParameters *modpars=new NBandSmooth::CModelParameters();
     // contains info about single point
  modpars->priorinfo=master.priorinfo;
  master.priorinfo->PrintInfo(); // print info about priors
  // 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 and print out their values and uncertainties
  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;
}
```

Smooth Emulator programs will often output lines describing their progress, either to the screen or to a file, as specified by the `SmoothEmulator_LogFile` parameter described above. For example, with the Monte Carlo tuning methods the output includes a report on the percentage of Metropolis 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. All output, except for some explicit code in main programs, is directed in this manner.

5.6 Useful Functionalities of `CSmoothMaster` Object

Smooth Emulator was designed so that the User can write their own main programs and access the functionality mainly through references to the `CSmoothMaster` object. Additionally, the User will find it useful to access the `CModelParameters`, `CparameterMap`, `CPriorInfo`, and `CObservableInfo` objects. Here is a compendium of calls to the `CSmoothMaster`:

- `CSmoothMaster(CparameterMap *parmap)`
  This is the constructor. CparameterMap object stores temulator parameters (not model parameters). CparameterMap functionality described below.
- `void ReadTrainingInfo()`
  This reads the training point information to be used for tuning.
- `TuneAllY()`
  // Tune all observables, $\boldsymbol{Y}$.
- `TuneY(string obsname)`
  Tunes one observable, by observable name.
- `TuneY(unsigned int iY)`
  Tune one observable, referenced by index.
- `CalcAllY(CModelParameters *modelpars,vector<double> &Y, vector<double> &SigmaY_emulator)`
  Calculates all observables. CModelParameters object stores information about a single point in model-parameter space. Object described further below.
- `CalcY(unsigned int iY,CModelParameters *modelpars,double &Y, double &SigmaY_emulator)`
  Calculates observable referenced by index.
- `CalcY(string obsname,CModelParameters *modelpars,double &Y, double &SigmaY_emulator)`
  Calculates observable referenced by observable name.
- `CalcAllYdYdTheta(CModelParameters *modelpars,vector<double> &Y, vector<double> &SigmaY_emulator,vector<vector<double>> &dYdTheta)`
  Also calculates derivatives w.r.t. $\vec{\boldsymbol{\theta}}$. Especially useful for some Markov chain searches in parameter space, e.g. Langevin approaches.
- `CalcYdYdTheta(string obsname,CModelParameters *modelpars,double &Y, double &SigmaY_emulator,vector<double> &dYdTheta)`
  Same, but for one observable referenced by observable name.
- `CalcYdYdTheta(unsigned int iY,CModelParameters *modelpars,double &Y, double &SigmaY_emulator,vector<double> &dYdTheta)`
  Same, but by index.
- `CalcAllLogP()`
  Prints technical information the User may find helpful in evaluating whether the choice of $\boldsymbol{\Lambda}$ is reasonable. For each observable, it calculates the ratio of the r.m.s. coefficients of rank-two to those of rank-one. One would roughly expect the ratio to be unity if $\boldsymbol{\Lambda}$ is appropriate, but from testing the measure is so noisy that it is not useful on a observable-by-observable basis. It also calculates the probability for the optimum coefficient set. This would be maximized for best choices of $\boldsymbol{\Lambda}$, but again is highly sensitive to fluctuations. Thus this information is not recommended for actual tuning $\boldsymbol{\Lambda}$.
- `TestAtTrainingPts()`
  Compares emulator predictions to full model calculations at training points. Observables should match and uncertainties should vanish.

- `TestAtTrainingPts(string obsname)`
  Same but for a single observable referenced by observable name.
- `TestAtTrainingPts(unsigned int iY)`
  Same but for a single observable referenced by index.
- `TestVsFullModel()`
  This is useful for comparing emulator to model evaluated at points not used for training. For the full model, make a sub-directory within the project directory called `fullmodeltestdata/`. Within that subdirectory, for each observable create a file called `OBSNAME.txt`, where `OBSNAME` is the name for each observable. Each file should have the format

  ```
  X_1  X_2  ...  X_N  Y
  X_1  X_2  ...  X_N  Y
  X_1  X_2  ...  X_N  Y

           ⋮
  ```

  Each line describes a point in parameter space and the observable calculated by the full model. `master.TestVsFullModel()` will calculate the emulated value at each $\vec{\boldsymbol{X}}$ and compare the full-model value to the emulator value. It will also give the uncertainty. If the uncertainty is well represented, 68 % of the emulated values should be within the uncertainty.
- `WriteCoefficientsAllY()`
  Writes the Taylor coefficients for all observables.
- `WriteCoefficients(string obsname)`
  Writes only those for one observable referenced by observable name.
- `WriteCoefficients(unsigned int iY)`
  Writes only for one observable reference by index.
- `ReadCoefficientsAllY()`
  Reads all Taylor coefficients from file.
- `ReadCoefficients(string obsname)`
  Reads Taylor coefficients for one observable referenced by observable name.
- `ReadCoefficients(unsigned int iY)`
  Reads Taylor coefficients for one observable referenced by observable index.

5.7 Other Potentially Useful *Smooth Emulator Objects*

Within the main program one may also wish to apply or access other objects with the *Smooth Emulator* framework. Their functionalities are described here:

- `CparameterMap`
  This object stores the emulator parameters described above. It is a simple inheritization of a map, linking string labels to values of various types. The object can read a parameter list from a file, e.g.

```
CparameterMap parmap;
parmap.ReadParsFromFile("parameters/emulator_parameters.txt")
```

The argument can be either a C++ string or a character array. The CSmoothMaster constructor takes a pointer to a CparameterMap object as a argument. If one wishes to print the parameters, the function is `parmap.PrintPars()`. To set a parameter within a program, `parmap.set(string,value)`, where `value` can be any of several types, e.g. bool, double, an integer, long long integer, string, $\cdots$. To retrieve a value from the map, the commands are `getB, getI, getLongI, getS, getD`, $\cdots$, for `bool, int, long long int` $\cdots$ types. For example,

```
parmap.getD("SmoothEmulator_LAMBDA",2.0);
```

would retrieve the value of $\mathbf{\Lambda}$, and if the map did not include `SmoothEmulator_LAMBDA`, it would return a default value of 2.0.

- `CPriorInfo`
  This object stores information about the prior. The `CSmoothMaster` object includes such an object, and automatically, during its construction, the `CSmoothMaster` object reads in the `Info/modelpar_info.txt` file and creates the object. The functionalities of potential interest might be addressed from a main program via:

```
smoothmaster.priorinfo.PrintInfo();  // prints out the parameter priors
unsigned int ipar=smoothmaster.priorinfo.GetIPosition(PARAMETER_NAME);
  // finds position given name of parameter (string)
string parname=smoothmaster.priorinfo.GetName(I);
  // finds name given position {\tt I} (unsigned int)
```

- `CModelParameters`
  This object stores the information describing a single point in the model-parameter space. It has two vectors storing the true $\vec{\boldsymbol{X}}$ and the scaled $\vec{\boldsymbol{\theta}}$ parameters. As a static variable, it stores a pointer to a `CPriorInfo` object so that it can translate back and forth from $\vec{\boldsymbol{X}}$ to $\vec{\boldsymbol{\theta}}$. The functionalities are fairly easily seen from the definition of its members header file. The ones most likely to be accessed by the User are:

```
vector<double> X;   // model parameters
vector<double> Theta; // scaled model parameters
void TranslateTheta_to_X();  // Given Theta, fill out X
void TranslateX_to_Theta(); // Given X, fill out Theta
void Print();    // Prints model parameters
void Write(string filename);    // Writes model parameters
void Copy(CModelParameters *mp);  // Copies from another object
void SetTheta(vector<double> &theta);  // Sets model parameters from vector, also tra
void SetX(vector<double> &X);  // Sets model parameters from vector, also translates
```

The emulator software stores all model-parameter information in these objects. There is a `CTrainingInfo` object within `CSmoothMaster` that stores a vector of

- `CObservableInfo`
  This object stores general information about all the observables, including their experimental value and experimental uncertainty. The object does not store observable information as calculated by the emulator. These objects are also fairly self-explanatory and the functionality can be ascertained by looking at some of the lines in the header file.

  ```
  unsigned int NObservables;
  vector<string> observable_name;
  vector<double> SigmaA0;
    // initial guess for spread of coefficients (only used for MC tuning methods)
  map<string,unsigned int> name_map;
  unsigned int GetIPosition(string obsname);
    // finds position given name of observable
  string GetName(unsigned int iposition);
    // finds name give position
  void ReadObservableInfo(string filename);
    // reads information about observable, either from
    //Info/observable_info.txt o Info/pca_info.txt
  void ReadExperimentalInfo(string filename);
    // reads experimental measurement and uncertainty (used in MCMC)
  vector<double> YExp,SigmaExp;
    // experimental measurement information
  void PrintInfo();
  ```

  There is such an object in the `CSmoothMaster` class. For example, to print the information about the observables from a main program, one would include the line:
  `master->observableinfo->PrintInfo();`

6 Emulating Principal Components

User Beware: the PCA software is not yet fully tested.

6.1 Overview

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_calctransformation` program included with the distribution. This will create files that shadow those used to emulate the observables. This will create a file `PCA_Info/observable_info.txt` which shadows the `Info/observable_info.txt`. The difference is that the observables will be named `z1,z2···`. 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/`.

6.2 Compiling and Running the PCA Programs

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, `${MY_LOCAL}/main_programs/pca_calctransformation_main.cc`:

```
int main() {
  CparameterMap parmap;
  NBandSmooth::PCA *pca = new NBandSmooth::PCA(&parmap);
  pca->CalcTransformationInfo();
}
```

To compile the program,

```
${MY_LOCAL}/main_programs% make pca_calctransformation
```

Before running one needs to have set up the usual files defining the training points, e.g. running *Simplex Sampler*, and running the full model at the training points. One can now run the program that creates files containing all the observable information, but translated into PCA components:

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

One can check the directory `${MY_PROJECT}/PCA_Info/` to make sure that one sees the files `experimental_info.txt`, `observable_info.txt` and `transformation_info.txt`. The latter file includes the transformation information so that one can readily translate from the nominal observables to the PCA components.

Before performing the tuning, one needs to edit the `parameters/emulator_parameters.txt` file and change the parameter `SmoothEmulator_UsePCA` to `true`. The output when tuning the emulator should look something like:

```
${MY_PROJECT}% ${MY_LOCAL}/bin/smoothy_tune
 ---- Tuning for z0 ----
 ---- Tuning for z1 ----
 ---- Tuning for z2 ----
 ---- Tuning for z3 ----
 ---- Tuning for z4 ----
      ⋮
```

The Taylor-expansion coefficients for the PCA observables are in the directory `${MY_PROJECT}/coefficients_p`

Another example program for PCA analysis is `${MY_LOCAL}/main_programs/pca_readinfo_calcy_main.cc`:

```
 int main() {
CparameterMap parmap;
    NBandSmooth::PCA *pca = new NBandSmooth::PCA(&parmap);
    pca->ReadTransformationInfo();

vector<double> Z,Y,SigmaZ_emulator,SigmaY_emulator;
Z.resize(pca->Nobs);
Y.resize(pca->Nobs);
SigmaZ_emulator.resize(pca->Nobs);
SigmaY_emulator.resize(pca->Nobs);
printf("---- Start with these values of Z ----\n");
for(unsigned int iobs=0;iobs<pca->Nobs;iobs++){
Z[iobs]=iobs;
printf("Z[%u]=%g\n",iobs,Z[iobs]);
}
printf("---- Translated values of Y ----\n");
SigmaZ_emulator[0]=SigmaZ_emulator[1]=SigmaZ_emulator[2]=SigmaZ_emulator[3]=SigmaZ_emulator

pca->TransformZtoY(Z,SigmaZ_emulator,Y,SigmaY_emulator);
for(unsigned int iobs=0;iobs<pca->Nobs;iobs++){
printf("%10.5f %10.5f\n",Y[iobs],SigmaY_emulator[iobs]);
}
printf("---- (Re)Translated values of Z ----\n");
pca->TransformYtoZ(Y,SigmaY_emulator,Z,SigmaZ_emulator);
for(unsigned int iobs=0;iobs<pca->Nobs;iobs++){
printf("%10.5f %10.5f\n",Z[iobs],SigmaZ_emulator[iobs]);
}
printf("---- Retranslated values of Z should match original and SigmaZ should all be unity
 }
```

This program illustrates how one can insert the PCA transformations into another program. After defining the `pca` object, the program reads in the transformation information with the `pca->ReadTransformati` command. The remainder of the program simply test the transformation by making a fake vector of the $\boldsymbol{Z}$ (PCA) components, then translating them to $\boldsymbol{Y}$ (observable) values and then back. The original values for $\boldsymbol{Z[i]}$ are set to $\mathbf{0, 1, 2, \cdots}$, which should then match the values after translating back and forth as illustrated.

To compile the program:

```
${MY_LOCAL}/main_programs% make pca_readinfo_calcy
```

Running the program:

```
${MY_PROJECT}% ${MY_LOCAL}/bin/pca_readinfo_calcy
 ---- Start with these values of Z ----
Z[0]=0
Z[1]=1
Z[2]=2
Z[3]=3
Z[4]=4
Z[5]=5
---- Translated values of Y and SigmaY ----
  -2.09833   24.99999
  -1.03694   33.33329
  -0.26087   41.66668
  -5.36371    0.16667
   3.49439    0.08333
   2.91093    0.08333
---- (Re)Translated values of Z and SigmaZ ----
   0.00000    1.00000
   1.00000    1.00000
   2.00000    1.00000
   3.00000    1.00000
   4.00000    1.00000
   5.00000    1.00000
---- Retranslated values of Z should match original and SigmaZ should all be unity ----
```

Note that for the PCA observables the scaling ensures they all have the same uncertainty, 1.0.

6.3 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*.
3. **SmoothEmulator_PCAMinVariance**
   This tells the emulator to not emulate PCA observables where the corresponding eigen value to the $\langle \boldsymbol{\delta Y_a/\sigma_a \delta Y_b/\sigma_b} \rangle$ matrix are below this amount. For such variables the emulated value of $\boldsymbol{Z_a}$ is simply set to zero.

6.4 Performing MCMC search with PCA components

Once the parameter `SmoothEmultor_UsePCA` is set to `true` and the parameter `SmoothEmulator_PCAMinVariance` is set to be greater than zero, the MCMC procedure should work in the MCMC programs. Both these parameters are set in the `Info/emulator_parameters.txt` file. For example, if `SmoothEmulator_PCAMinVariance` is set to 0.01, then the emulator will not bother emulating any PCA observable where, in the training set, $\langle \boldsymbol{\delta Z_a \delta Z_a} \rangle$ is less than 0.01. Given that the PCA variables are scaled by their uncertainties, this then ignores PCA observables that vary less than one percent of their uncertanty.

The User should be warned again that all this functionality is not yet fully tested.

7 Markov-Chain Monte Carlo (MCMC) Generation of the Posterior

The final step in Bayesian analyses is to generate MCMC traces through parameter space. *Smooth Emulator* software is designed for the MCMC programs to be run from within the `${MY_PROJECT}` directory. 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.

The log-likelihood, $\boldsymbol{LL}$, for the MCMC generation is assumed to be of a simple form. Summing over the observables $\boldsymbol{I}$,

$$\begin{aligned}
\boldsymbol{\sigma^2_{I,\mathrm{tot}}} &= \boldsymbol{\sigma^2_{I,\mathrm{exp}} + \sigma^2_{I,\mathrm{theory}} + \sigma^2_{I,\mathrm{emulator}}},\\
\boldsymbol{LL} &= \boldsymbol{-\sum_I \frac{(Y_{I,\mathrm{exp}} - Y_{I,\mathrm{emu}})^2}{2\sigma^2_{I,\mathrm{tot}}} - \ln(\sigma_{I,\mathrm{tot}})}.
\end{aligned}$$

The main program that runs the mcmc code is `${MY_LOCAL}/main_programs/mcmc_main.cc`. To compile the program:

```
${MY_LOCAL}/main_programs% make mcmc
```

Next, one needs to edit the parameter file `${MY_PROJECT}/parameters/mcmc_parameters.txt`. An example file is:

```
MCMC_LANGEVIN false
MCMC_METROPOLIS_STEPSIZE 0.04
MCMC_LANGEVIN_STEPSIZE 0.5
MCMC_NBURN  100000
MCMC_NTRACE 100000
MCMC_NSKIP  5
RANDY_SEED  12345
```

The program can run either a Metropolis algorithm or a Langevin algorithm. If `MCMC_LANGEVIN` is `false`, the Metropolis algorithm is invoked. The Langevin algorithm is faster when one has large numbers of parameters, but it ignores the emulator uncertainty (because it has a $\boldsymbol{\theta}$ dependence). Thus, if the emulator uncertainty is significant, the Metropolis method is recommended. For the Metropolis method, the stepsize of the random steps should be adjusted so that the Metropolis success probability is on the order of one half. If the probability is only a few percent, or if it is very close to 100% the efficiency of the method suffers. When one runs `mcmc` it will report the success percentage. If the percentage is worrisome, one should reset `MCMC_METROPOLIS_STEPSIZE` and re-run. The parameter `MCMC_NBURN` is the number of Metropolis steps the program attempts during the burn-in stage, i.e. before the trace is reported when the trace might be outside the likely region. The trace then records `MCMC_NTRACE` steps, taking `MCMC_NSKIP` steps between writing down the information about the point in model parameter space. Thus, after burn-in, the chain performs `MCMC_NSKIP`$\times$`MCMC_NTRACE` samplings, recording `MCMC_NTRACE` values of $\vec{\boldsymbol{\theta}}$ and $\vec{\boldsymbol{X}}$. At the current time, the Langevin method is not fully tested and should be avoided.

Then, from the project directory, run the program. Output should look something like this:

```
${MY_PROJECT}% ${MY_LOCAL}/bin/mcmc
 At beginning of Trace, LL=-15.526125
 At end of trace, best LL=-0.131612
 Metropolis success percentage=72.030000
 finished burn in
 At beginning of Trace, LL=-2.904881
 finished 10%
 finished 20%
 finished 30%
 finished 40%
 finished 50%
 finished 60%
 finished 70%
 finished 80%
 finished 90%
 finished 100%
 At end of trace, best LL=-0.075831
 Met This is list of $\vec{\theta}$ for each point in the trace.
\item {\tt XbarThetabar.txt}: Same as the output above
\item {\tt sigma2.txt}: This gives the $N_{\rm par}\times N_{\rm par}$ covariance matrix $\
\item {\tt sigma2\_eigenvalues.txt}: That eigenvalues of that matrix
\item {\tt sigma2\_eigenvectors.txt}: The eigenvectors
\end{itemize}
The information for $\langle\delta \theta_i\delta\theta_{b}\rangle$ might be useful later o

Finally, we review the source code in {\tt \$\{MY\_LOCAL\}/main\_programs/mcmc\_main.cc}:
{\tt
\begin{verbatim}
int main(){
  CparameterMap *parmap=new CparameterMap();
  CSmoothMaster master(parmap);
  CMCMC mcmc(&master);
  master.ReadCoefficientsAllY();
  unsigned int Nburn=parmap->getI("MCMC_NBURN",1000);  // Steps for burn in
  unsigned int Ntrace=parmap->getI("MCMC_NTRACE",1000); // Record this many points
  unsigned int Nskip=parmap->getI("MCMC_NSKIP",5); // Only record every Nskip^th point
  mcmc.PerformTrace(1,Nburn);
  CLog::Info("finished burn in\n");
  mcmc.PruneTrace(); // Throws away all but last point
  mcmc.PerformTrace(Ntrace,Nskip);
  mcmc.EvaluateTrace();
  mcmc.WriteTrace();
  return 0;
}
```

This is mostly self-explanatory. If one wishes to avoid writing out the trace, the line `mcmc.WriteTrace` can be deleted. If the User wishes to run the code in batch mode, the output can be directed to a file, `mcmc_log.txt`, by adding the line

```
   MCMC_LogFileName mcmc_log.txt
```

to the parameter file `parameters/mcmc_parameters.txt`.

8 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 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

Installation and compilation is described in Sec. 2. As was defined in that section, the tutorial will refer to three locations with the short hand:

`${GITHOME_SMOOTH}`	Location of Git Repository, e.g. `/Users/CarlosSmith/bandframework/software/SmoothEmulator`
`${MY_LOCAL}`	Should be a subdirectory of `${GITHOME_SMOOTH}/` Directory including executables, `${MY_LOCAL}/bin` and main programs, `${MY_LOCAL}/software/main_programs`
`${MY_PROJECT}`	Work space where parameter files, data, results and figures are created and stored. User may have several different such directories

Then, the user should have established a personalized project directory by duplicating the `${GITHOME_SMOOTH}/` directory onto their computer. The User should have also compiled the main libraries

```
${GITHOME_SMOOTH}/software% cmake .
```

and the main programs,

```
${MY_LOCAL}/software% cmake .
${MY_LOCAL}/software% make
```

This will compile all the main source programs in `${MY_LOCAL}/software/main_programs/*.cc` and a "fake" full model to be used in the tutorial `${MY_LOCAL}/software/fakefulmodels/fakerhic.cc`. The repository was organized to encourage Users to edit any files in `${MY_LOCAL}/`. If the User wishes to restore any original files, a copy can be found at `${GITHOME_SMOOTH}/templates/mylocal`.

Several executables should now appear in `${MY_LOCAL}/bin/`, `simplex, smoothy_tune, mcmc, smoothy_testattrainingpts, smoothy_testvsfullmodel, pca_calctransformation, pca_readinfo_cal` and `fakerhic`. The User might find it convenient to add `${MY_LOCAL}/bin` to their path. 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 and the lower-level source code unchanged.

8.3 Creating Necessary Info Files

The User will run the software from the `${MY_PROJECTS}/` directory. Before a User can run *Simplex Sampler* they must create information files that describe the model-parameter priors and list the observable names. Both files are need to be in the `${MY_PROJECTS}` directory. The first file is `${MY_PROJECTS}/Info/modelpar_info.txt`, which describes the model parameters and their priors. 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
```

Thus, the model has six parameters. The second entry in each line is either `uniform` or `gaussian`. If the entry 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 full 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 provided 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 (but still shoud be listed).

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 $n+1$ training points, where $n = 6$ is the number of model parameters. Each point corresponds to the vertices of an $n+1$ 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/`. Here, `Simplex_TrainType=2`, which adds points half-way between each pair of simplex points. These additonal points are then moved outward and the original simplex points are brought inward. This method has precisely the number of training points as the number of coefficients necessary for a quadratic fit.

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 run a full model based on the given structure, tailored to address their specific problem. For the tutorial, a fake full 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 usually superfluous.

Additionally, `fakerhic` created a directory `${MY_PROJECTS}/fullmodel_testdata/` which stores information about full-model runs at 50 randomly chosen points in the model-parameter space. These points are not used for tuning. This data can be used later to test the emulator.

8.6 Running *Smooth Emulator*

Before building and tuning the emulator, the User needs to edit one additional file, 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 2.5 # 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. 5. 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.

To get an idea of how one might build one's own main program to access the capabilities used above, the source code for `${MY_LOCAL}/software/main_programs/smoothy_tune_main.cc` is:

```
#include "msu_smoothutils/parametermap.h"
#include "msu_smooth/master.h"
#include "msu_smoothutils/log.h"
using namespace std;
int main(){
NMSUUtilsCparameterMap *parmap=new CparameterMap();
NBandSmooth::CSmoothMaster master(parmap);
master.ReadTrainingInfo();
master.TuneAllY();
master.WriteCoefficientsAllY();
return 0;
}
```

Hopefully, the User will find this and the other main-program source codes to be fairly self-explanatory. Nonetheless, detailed explanations can be found in Sec. 5.

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_testattrainingpts
 --- Y_train     Y_emulator    Sigma_emulator ----
------ itrain=0 --------
Y[0]= 4.026e+02 =?  4.026e+02  +/-  1.14910e-07
Y[1]= 7.049e+02 =?  7.049e+02  +/-  1.41579e-07
Y[2]= 1.112e+03 =?  1.112e+03  +/-  1.90839e-07
Y[3]= 4.953e+00 =?  4.953e+00  +/-  2.05171e-09
Y[4]= 3.720e-01 =?  3.720e-01  +/-  2.85035e-10
Y[5]= 6.226e-01 =?  6.226e-01  +/-  4.36786e-10
------ itrain=1 --------
Y[0]= 4.667e+02 =?  4.667e+02  +/-  7.32386e-08
Y[1]= 7.758e+02 =?  7.758e+02  +/-  9.02363e-08
Y[2]= 1.094e+03 =?  1.094e+03  +/-  1.21633e-07
```

```
Y[3]= 5.110e+00 =?  5.110e+00  +/-  1.30767e-09
Y[4]= 4.620e-01 =?  4.620e-01  +/-  1.81669e-10
Y[5]= 7.040e-01 =?  7.040e-01  +/-  2.78389e-10
------ itrain=2 --------
Y[0]= 4.617e+02 =?  4.617e+02  +/-  2.17777e-07
Y[1]= 6.909e+02 =?  6.909e+02  +/-  2.68321e-07
Y[2]= 1.071e+03 =?  1.071e+03  +/-  3.61679e-07
Y[3]= 4.999e+00 =?  4.999e+00  +/-  3.88841e-09
Y[4]= 3.575e-01 =?  3.575e-01  +/-  5.40200e-10
Y[5]= 6.648e-01 =?  6.648e-01  +/-  8.27799e-10
------ itrain=3 --------
Y[0]= 4.574e+02 =?  4.574e+02  +/-  2.92485e-08
Y[1]= 7.132e+02 =?  7.132e+02  +/-  3.60366e-08
```

$\vdots$

```
------ itrain=27 --------
Y[0]= 4.527e+02 =?  4.527e+02  +/-  2.44311e-07
Y[1]= 8.317e+02 =?  8.317e+02  +/-  3.01012e-07
Y[2]= 1.112e+03 =?  1.112e+03  +/-  4.05745e-07
Y[3]= 6.135e+00 =?  6.135e+00  +/-  4.36216e-09
Y[4]= 3.312e-01 =?  3.312e-01  +/-  6.06017e-10
Y[5]= 4.445e-01 =?  4.445e-01  +/-  9.28657e-10
```

The observables, $\boldsymbol{Y}[\mathbf{0}]\cdots\boldsymbol{Y}[\mathbf{5}]$ 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 = 436.167 +/- 2.16724
meanpt_kaon = 692.539 +/- 2.67022
meanpt_proton = 1084.85 +/- 3.59929
Rinv = 5.00013 +/- 0.0386959
v2 = 0.361353 +/- 0.00537586
RAA = 0.545491 +/- 0.00823794
```

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/CMakeLists.txt`. 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/SmoothEmulat`
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();
   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;
}
```

The above 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) Read the Taylor coefficients that were written when the emulator was tuned
d) Creates a model-parameters object, `modpars`, that stores the coordinates of the model-parameter point
e) Reads in the model parameters interactively
f) Calculates the observables from the emulator
g Prints out the emulated observable and the uncertainty for for the emulator

10.1 Exploring the Posterior via Markov-Chain Monte-Carlo

Given the experimental information, which is stored in project directory in `Info/experimental_info.txt`, one can then use the tuned emulator to explore the posterior likelihood through MCMC, which works via a Metropolis algorithm. The file `Info/experimental_info.txt` provided in the template is:

```
meanpt_pion      481.179     20     0.0
meanpt_kaon      757.872     30     0.0
meanpt_proton    1113.3      40     0.0
Rinv             6.27842     0.3    0.0
v2               0.382973    0.1    0.0
RAA              0.558367    0.1    0.0
```

The first column is the list of observable names, which should be identical to those listed in `Info/observable_info.txt`. The second and third columns lists the experimental measurement, $\boldsymbol{Y_a}$, and the experimental uncertainty, $\boldsymbol{\sigma_a^{\rm exp}}$. The last column lists the additional uncertainty due to errors, $\boldsymbol{\sigma_a^{\rm theory}}$, i.e. missing physics, in the theoretical model. For the purposes of comparing theory to data, only the combination $(\boldsymbol{\sigma_a^{\rm exp}})^2 + (\boldsymbol{\sigma_a^{\rm exp}})^2$ comes into play, because this combination appears in the likelihood for the posterior,

$$\begin{aligned}\mathcal{L}(\vec{\theta}) &= \prod_a \frac{1}{\sqrt{2\pi(\sigma_a^{\rm tot})^2}} \exp\left\{-\frac{(Y_a(\vec{\theta}) - Y_a^{\rm exp})^2}{2(\sigma_a^{\rm tot})^2}\right\} \\ (\sigma_a^{\rm tot})^2 &= (\sigma_a^{\rm exp})^2 + (\sigma_a^{\rm theory})^2 + (\sigma_a^{\rm emu})^2.\end{aligned} \tag{10.1}$$

Whereas the emulator uncertainty, $\boldsymbol{\sigma_a^{\rm emu}}$, depends on the location in parameter space, $\vec{\boldsymbol{\theta}}$, the other two contributions are assumed to be independent of $\vec{\boldsymbol{\theta}}$.

There are special parameters for the MCMC. These are stored in `Info/mcmc_parameters.txt`. For the tutorial template, that file is

```
# This is for the MCMC search of parameter space
 # (not for the emulator tuning)
 MCMC_LANGEVIN false
 MCMC_METROPOLIS_STEPSIZE 0.05
 MCMC_LANGEVIN_STEPSIZE 0.5
 MCMC_NBURN  100000
 MCMC_NTRACE 100000
 MCMC_NSKIP  5
 RANDY_SEED  12345
```

The first parameter, `MCMC_LANGEVIN` should be set to `false`, as the Langevin MCMC (as opposed to the Metropolis version) is under development. The Metropolis stepsize should be adjusted so that the Metropolis success rate is approximately one half. The success rate prints out when the `mcmc` code runs. If the success rate is anywhere between 20 and 80%, this should be fine. But, if the rate is close to zero or 100%, the efficiency of the procedure suffers. It is recommended to run the MCMC code with a modest number of steps, then adjust the stepsize accordingly.

The parameter `MCMC_NBURN` sets the number of Metropolis steps to be used in the "burn-in" stage, i.e. before one begins to store the trace. The number of elements to store in the trace in `MCMC_NTRACE`, and `MCMC_NSKIP` sets the number of steps to skip before storing a new point in the trace. Thus, if `MCMC_NTRACE` is one million and if `MCMC_NSKIP`=5, then the procedure will perform 5 million steps, and store every fifth one, leading to one million stored points in the trace. Finally, `RANDY_SEED` sets the random number seed.

Running the MCMC program gives the following output:

```
 ${MY_PROJECTS}/rhic% ${MY_LOCAL}/bin/mcmc
 At beginning of Trace, LL=-9.538317
 At end of trace, best LL=-4.415930
 Best Theta=
 0.186889  0.075493  0.165236  0.221045  0.187867  0.219744
 Metropolis success percentage=73.550000
 finished burn in
 At beginning of Trace, LL=-6.743758
 finished 10%
 finished 20%
 finished 30%
 finished 40%
 finished 50%
 finished 60%
 finished 70%
 finished 80%
 finished 90%
 finished 100%
 At end of trace, best LL=-4.381435
 Best Theta=
 0.169135  0.141132  0.153260  0.212104  0.245712  0.193710
 Metropolis success percentage=75.474600
 writing, ntrace = 100001
```

Here `best LL` refers to the log-likelihood and `Best Theta` refers to the value of $\vec{\theta}$ that gave the maximum log-likelihood. Values of `Best Theta` and `best LL` are given after the burn-in and after the trace.

The trace is written to `mcmc_trace/trace.txt`. It is in the format

```
theta_1 theta_2 theta_3 theta_4 ...
theta_1 theta_2 theta_3 theta_4 ...
theta_1 theta_2 theta_3 theta_4 ...
```

$\vdots$ If `MCMC_NTRACE` is set to a million, there would be a million lines in the file. The program also calculated various covariances: $\langle\langle\delta\theta_i\delta\theta_j\rangle\rangle$, $\langle\langle\delta\theta_i\delta Y_a\rangle\rangle$, and $\langle\langle\delta Y_a\delta Y_b\rangle\rangle$. The quantities $\langle\langle\ldots\rangle\rangle$ refer to averages over the posterior, i.e. averages over the trace. The eigenvalues and eigenvectors of $\langle\langle\delta\theta_i\delta\theta_j\rangle\rangle$ are also recorded. Hopefully, the User will find the file names in `mcmc_trace/` to be self-explanatory.

As described in Sec. 9 the covariances in the posterior can also be used to quantify the resolving power of specific observables to constrain specific parameters. One such measure is

$$\mathcal{R}_{ia} \equiv \frac{d\langle\langle\theta_i\rangle\rangle}{dY_a^{\rm exp}}\langle\delta Y_a^2\rangle^{1/2}. \tag{10.2}$$

This quantifies how the posterior value of θ_i changes as the experimental value changes if the experimental value, $Y_a^{\rm exp}$, changes an amount characteristic of the variance of Y_a across the prior. Given that there may not be a set of training points calculated uniformly across the prior, the final factor is estimated as described in Sec. 9. Higher values of $\mathcal{R}_{ia}$ for different a demonstrate the relative contributions of different observables a to constrain a model parameter i. The resolving power matrix is written to `mcmc_trace/ResolvingPower.txt`.

10.2 Making Plots

Three python scripts (using MATPLOTLIB) are provided to provide graphical insight into the posterior likelihood, into the resolving power and for viewing how the emulator uncertainty compares to the discrepancies between full-model runs (not used for tuning) and emulated value. One can visit the `${MY_PROJECTS}/figs/` directory and peruse the file `directions.txt` for more detailed instructions of how to create the plots below. First, one must create two data files. These provide the names of observables and model parameters to be used by the plots.

The two files are given the same names as two files in `${MY_PROJECTS}/Info/`. The first is `${MY_PROJECTS}/figs/modelpars_info.txt`. It differs from the one in the `Info/` directory in that it has only three columns, though the first column is identical. The provided file is

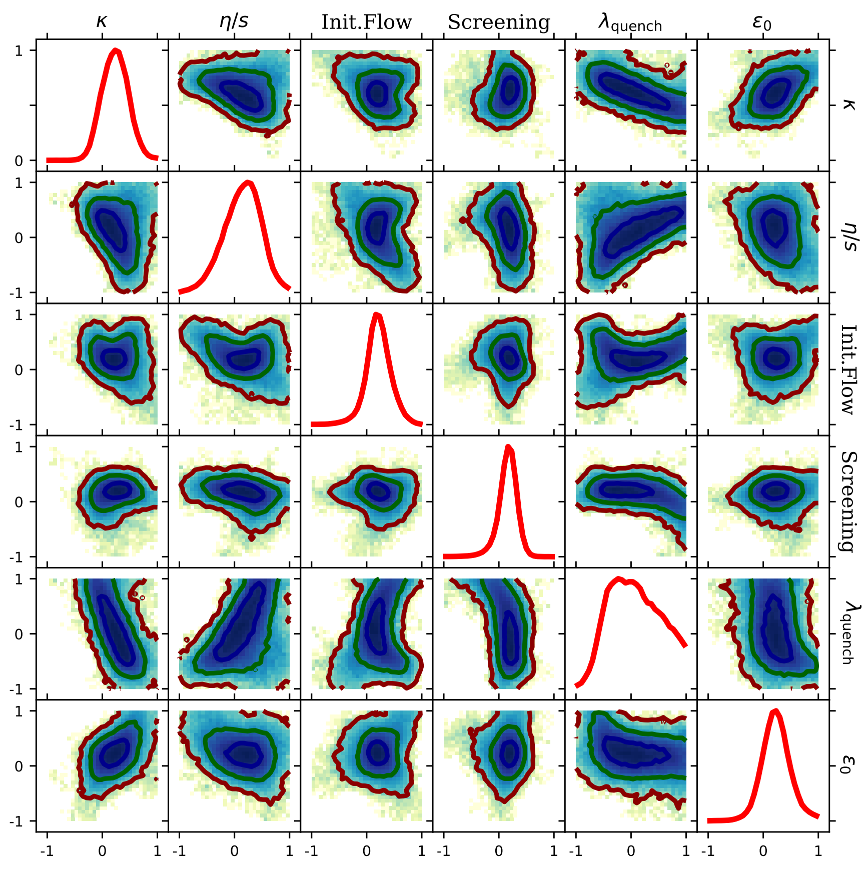
```
compressibility          uniform    $\kappa$
etaovers                 uniform    $\eta/s$
initial_flow             uniform    Init.Flow
initial_screening        uniform    Screening
quenching_length         uniform    $\lambda_{\rm quench}$
initial_epsilon          uniform    $\epsilon_0$
\end{verbatime}}
As one can see, the last column is used by MATPLOTLIB to label axes. The middle column is n

The second required file is {\tt \$\{MY\_PROJECTS\}/figs/observable\_info.txt}, and the pro
{\tt
\begin{verbatim}
meanpt_pion         $\langle p_t\rangle_{\pi}$
meanpt_kaon         $\langle p_t\rangle_{K}$
meanpt_proton       $\langle p_t\rangle_{p}$
Rinv                $R_{\rm inv}$
v2                  $v_2$
RAA                 $R_{AA}$
```

Again, the first column is identical to that in `Info/` and the second is used for labeling.

To graph the posterior likelihood, first be sure to run the `mcmc` program, then change into the `figs/posterior` directory and enter the command:

```
${MY_PROJECT}/figs/posterior% ln -s ../../mcmc_trace/trace.txt .
${MY_PROJECT}/figs/posterior% python3 posterior.py
```

One can replace "`ln -s`" with "cp" or "mv". The script should create a file `posterior.pdf`, which looks like:



Projections for the posterior likelihood from the MCMC trace. The contour lines represent $\mathbf{1-\sigma}$, $\mathbf{2-\sigma}$ and $\mathbf{3-\sigma}$ likelihoods.

The likelihood is projected for individual model parameters, or for pairs. The plot is in terms of the scaled variables, $\boldsymbol{\theta_i}$. To translate to the true model-parameter ranges, one can look at the `Info/modelpars_info.txt` file, which gives the prior ranges of the model parameters before they are scaled to the -1 to 1 range. The file `figs/directions.txt` shows how the User can alter plot. For example there is a line in the python script, `ParsToPlot=[1,2,3,4,5,0]`, which the User can edit to change the ordering of the model-parameters, and to choose which model parameters are considered.

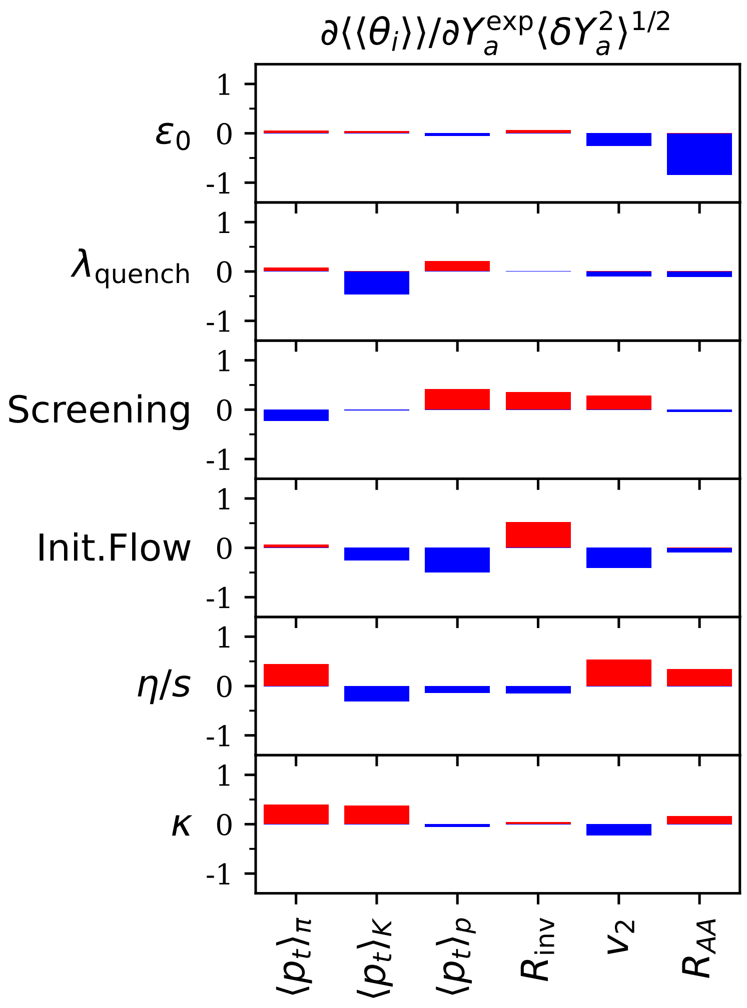
Similarly, one can plot the resolving power. The User can visit the directory `figs/resolvingpower/`. The User should copy the files `mcmc_trace/ResolvingPower.txt` and `Info/modelpar_info.txt` to this directory, editing the `Info/modelpar_info.txt` as was done for the posterior visualization figures described above. The User must also copy over the `Info/observable_info.txt` file and edit it in a similar fashion. In the template file is

```
meanpt_pion          $\langle p_t\rangle_{\pi}$
meanpt_kaon          $\langle p_t\rangle_{K}$
meanpt_proton        $\langle p_t\rangle_{p}$
Rinv                 $R_{\rm inv}$
v2                   $v_2$
RAA                  $R_{AA}$
```

The first column should be exactly the same as the first line in the original file. After running the `mcmc` program, the figure can be produced via the command

```
${MY_PROJECT}/figs/resolvingpower% ln -s ../../mcmc_trace/ResolvingPower.txt .
${MY_PROJECT}/figs/resolving power% python3 RP.py
```

The figure should look like:



Resolving Power. Red bars represent positive correlations with $\boldsymbol{Y_a^{\rm exp}}$ and $\boldsymbol{\theta_i}$. Larger bars suggest that the particular observable contributes more to the constraint of the particular model parameter.

The third provide python script is in `figs/YvY/` and it compares full-model runs (not used for tuning) to the emulator. This is useful for seeing whether the emulator's error estimates are reasonable. First, one must run the program that writes out the emulator predictions for the full-model runs. This is accomplished by the command:

```
${MY_PROJECT}% smoothy_testvsfullmodel
 meanpt_pion: 44 out of 50 points within 1 sigma
 meanpt_kaon: 40 out of 50 points within 1 sigma
 meanpt_proton: 36 out of 50 points within 1 sigma
 Rinv: 43 out of 50 points within 1 sigma
 v2: 46 out of 50 points within 1 sigma
 RAA: 44 out of 50 points within 1 sigma
```

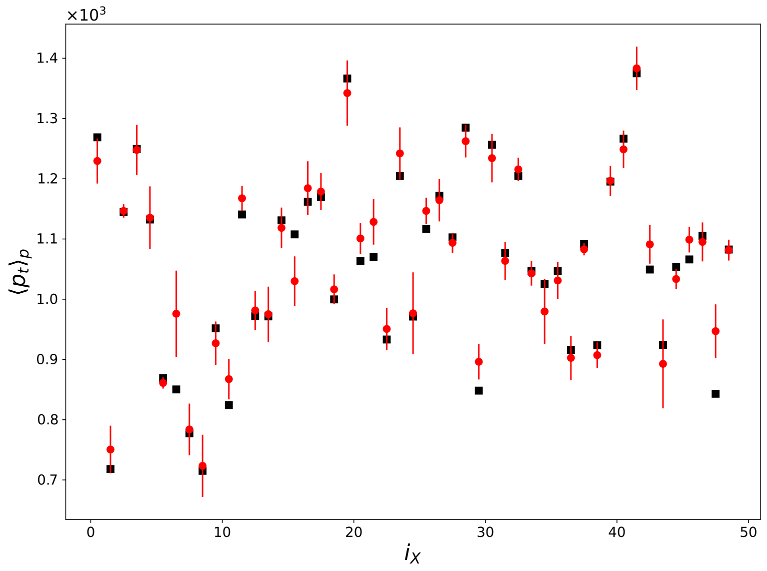
If the uncertainty were perfectly stated, 68% of the points would be within one standard deviation.

In this case the fraction was higher, which suggests that the uncertainty was somewhat overstated.

To make the plot, first change into the `figs/YvsY/` directory, and enter

```
${MY_PROJECT}/figs/YvsY% ln -s ../../fullmodel_testdata/ResolvingPower.txt .
${MY_PROJECT}/figs/YvsY% python3 YvsY.py
 ['meanpt_pion', 'meanpt_kaon', 'meanpt_proton', 'Rinv', 'v2', 'RAA']
 Enter iY: 2
 36 of 49  points within 1 sigma
```

The script will prompt the User for which observable to consider. In this case, choose 0-5 for the six possible observables. In this case '2' was entered and the chosen observable was `meanpt_proton`.



Comparison of full-model values (black squares) for 50 points in parameter space to emulator values (red circles). The uncertainties are solely those associated with the emulation. If the uncertainties were accurately expressed, 68% of the points would lie within the uncertainty intervals.

10.3 PCA Analysis

The PCA functionality needs further testing and is not included in the tutorial at this time.

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_{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. (9.1) 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. (9.1) 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

$$\langle E(\theta_1 = \theta_2 = \cdots \theta_N = 0)^2\rangle = d_{n_i=0}^2 \sigma_{K=0}^2 f_{K=0}^2(\vec{\theta} = 0). \tag{9.4}$$

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

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

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 $\vec{\theta}$,

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

In this limit one can also see that

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

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

Variant C: Eliminating the $1/K!$ 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 $K_{\max}$ 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 $x \to 1$. 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_{train} full model runs at positions $\vec{\theta}_a,\ a = 1, N_{\text{train}}$, 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

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

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 $A_a, a \le N_{\rm train}$. 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. (9.12) 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, $a \le 0 < N_{\rm train}$, 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,

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

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. (9.29),

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

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

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

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

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

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

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 $A_{i>N_{\text{train}}}$ as a high dimensional vector, the only other vectors in that space, i.e. those with index $i > N_{\text{train}}$, that can represent the solution are the N_{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_{train} simultaneous equations. For the sake of brevity, the index labels, a, b, c or t will vary from $1\cdots N_{\text{train}}$ and those labeled i, j will vary from $N_{\text{train}} + 1\cdots N_{\text{coef}}$. One must minimize

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

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,

$$\begin{aligned}
0 &= \sum_b B_{ab}\gamma_b + \sum_{bc} B_{ab}B_{bc}\gamma_b - B_{ab}\alpha_b, \qquad (9.39)\\
\alpha_a &= (\delta_{ab} + B_{ab})\,\gamma_b,\\
\gamma_a &= (\mathbb{1} + B)^{-1}_{ab}\alpha_b.
\end{aligned}$$

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. (9.39) for γ_b, which when inserted into Eq. (9.36) gives the coefficients A_i for $i > N_{\rm train}$. To find the coefficients, A_a for $a \le N_{\rm train}$, one then applies Eq. (9.29).

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. (9.51), $(T_i(\vec{\theta}) - T_a(\vec{\theta})\beta_{ai})$, 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. (9.47) and putting this all together, one has 8 terms altogether.

$$\sigma^2_E(\vec{\theta}) \;=\; \sigma^2_1 + \sigma^2_2 + \sigma^2_3 + \sigma^2_4 + \sigma^2_5 + \sigma^2_6 + \sigma^2_7 + \sigma^2_8. \tag{9.53}$$

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

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

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

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

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

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

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 $2a$,

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

whereas for a sphere of radius a,

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

The ratio of the radii is then

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

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. 11.1. Figure 11.1 also shows

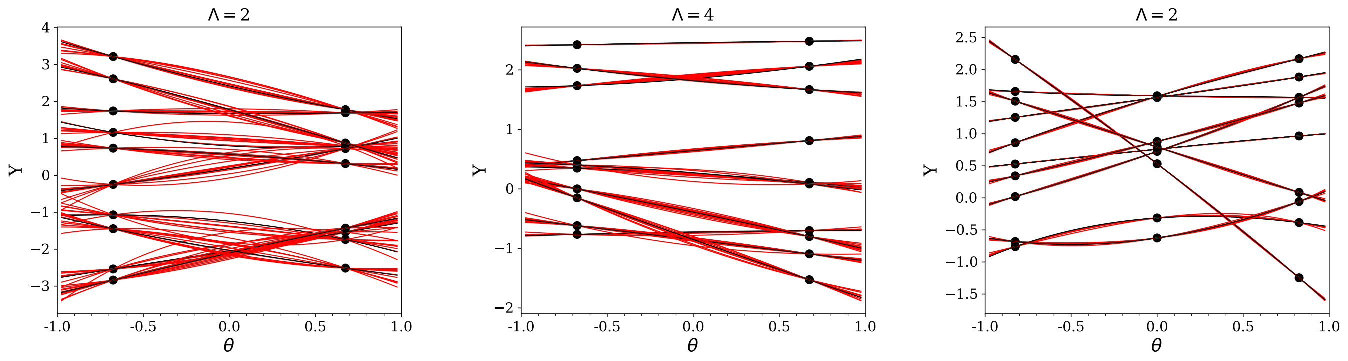


Figure 11.1: Real functions (black lines) are emulated with the *Smooth Emulator* (default settings). Ten different functions were generated sampled by the Monte Carlo method. The spread of the lines represents the uncertainty. The spread is reduced by either reducing the smoothness parameter, $\boldsymbol{\Lambda}$, 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 full model, $\boldsymbol{F}$, was constructed from sample smooth functions, with the coefficients generated randomly and a smoothness parameter set to three or six. The $\boldsymbol{A}_{\vec{n}=0}$ coefficient for the full model was set to zero to better accommodate viewing results. The emulator was not given that information. Twenty instances of full models were emulated. For each full model five random points in parameter space were chosen. The emulator value and its uncertainty are plotted alongside the full-model values in Fig. 11.2. The 100 comparisons show that the emulator accurately predicts the uncertainty. This is not surprising, because the emulator used the same smoothness parameter as was used to construct the full models. The width parameters, $\boldsymbol{\sigma}$, were explored stochastically, and remarkably they seem to fluctuate around the same value used to construct the full model, albeit with fluctuations of the order of 30%.

Figure 11.2 first shows the 100 comparisons for the case with the smoothness parameter, $\boldsymbol{\Lambda} = \mathbf{3}$. 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 $\boldsymbol{A}$ coefficients and of $\boldsymbol{\sigma}$, then using the variance of the 16 samples to define the uncertainty. Variant *a* of the default emulator was used, i.e. the coefficient $\boldsymbol{A}_0$ was not given a constraining prior distribution, whereas all other coefficients were weight by a Gaussian of width $\boldsymbol{\sigma}$. In the other two panels of Fig. 11.2, the procedure was repeated but once with a higher smoothness parameter, and once with a the 28 training points, placed according to the procedure mentioned in Sec. 3, where an additional set of points in parameter space was generated by choosing points that bisect all the lines connecting points in the original simplex, then doubling their radius. As expected, smoother functions are more easily emulated with a given number of training points, and using more training points also improves the accuracy.

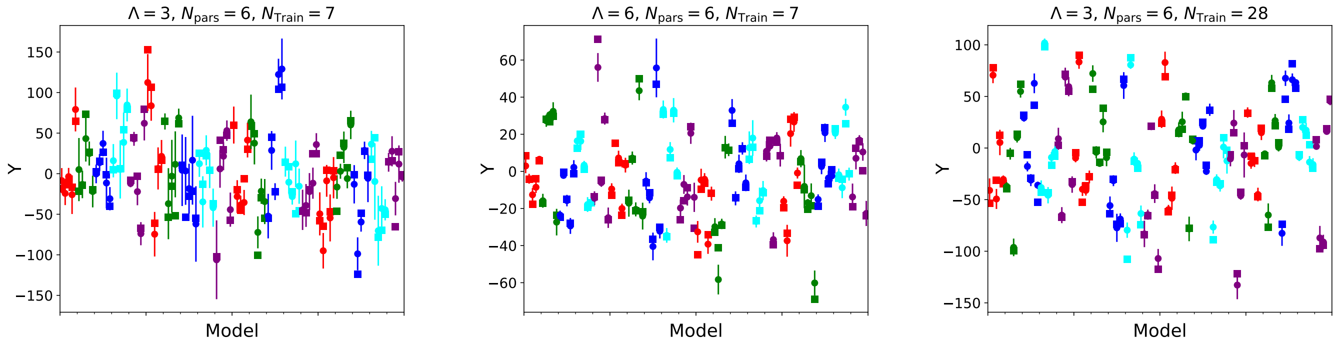


Figure 11.2: Using 20 instances of full models using six parameters, choosing 5 random points in parameter space for each model, the emulator (circles) and its uncertainty were compared to the full-model values (squares). Neighboring points of the same color emulated the same full model. The accuracy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