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. **??**. 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 `${MY_PROJECT}/` directory. Before a User can run *Simplex Sampler* they must create information files that describe the model-parameter priors and list the observable names. Both files are need to be in the `${MY_PROJECT}` directory. The first file is `${MY_PROJECT}/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_PROJECT}/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_PROJECT}/parameters/simplex_parameters.txt`. The provided file is

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

Because the first line is commented, the output of *Simplex Sampler* will be to the screen. Otherwise it would go to the specified file. By setting `Simplex_TrainType=1`, the sampler will choose $\boldsymbol{n+1}$ training points, where $\boldsymbol{n = 6}$ is the number of model parameters. Each point corresponds to the vertices of an $\boldsymbol{n+1}$ dimensional simplex. Finally, the parameter `Simplex_ModelRunDirName` is set to "`modelruns`". This informs `Simplex Sampler` to write the coordinates of each training point and the corresponding observables in the directory `${MY_PROJECT}/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_PROJECT}/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_PROJECT}/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_PROJECT}/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

```
#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. **??**. 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_PROJECT}/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`$\cdots$, 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. **??**.

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_PROJECT}/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
```

⋮

```
------ 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_PROJECT}/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. `${MY_LOCAL}/software/main_programs/smoothy_calcobs_main.cc`. The User can then design their own program based on this source code, and compile and link it by editing `${MY_LOCAL}/software/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}/software/main_programs/SmoothEmulator_calcobs_main.cc` is

```
#include "msu_smoothutils/parametermap.h"
#include "msu_smooth/master.h"
#include "msu_smoothutils/log.h"
using namespace std;
int main(){
   NMSUUtils::CparameterMap *parmap=new CparameterMap();
   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 $\boldsymbol{(\sigma_a^{\rm exp})^2 + (\sigma_a^{\rm exp})^2}$ comes into play, because this combination appears in the likelihood for the posterior,

$$\boldsymbol{\mathcal{L}(\vec{\theta}) = \prod_a \frac{1}{\sqrt{2\pi(\sigma_a^{\rm tot})^2}} \exp\left\{ -\frac{(Y_a(\vec{\theta}) - Y_a^{\rm exp})^2}{2(\sigma_a^{\rm tot})^2} \right\}} \tag{10.1}$$

$$\boldsymbol{(\sigma_a^{\rm tot})^2 = (\sigma_a^{\rm exp})^2 + (\sigma_a^{\rm theory})^2 + (\sigma_a^{\rm emu})^2.}$$

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

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_PROJECT}/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 $\vec{\boldsymbol{\theta}}$ 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 ...
                ⋮
```

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

As described in Sec. **??** 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

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

This quantifies how the posterior value of $\boldsymbol{\theta_i}$ changes as the experimental value changes if the experimental value, $\boldsymbol{Y_a^{exp}}$, changes an amount characteristic of the variance of $\boldsymbol{Y_a}$ 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. **??**. Higher values of $\boldsymbol{\mathcal{R}_{ia}}$ for different $\boldsymbol{a}$ demonstrate the relative contributions of different observables $\boldsymbol{a}$ to constrain a model parameter $\boldsymbol{i}$. 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_PROJECT}/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 `${MY_PROJECT}/Info/`. The first is `${MY_PROJECT}/figs/r`
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

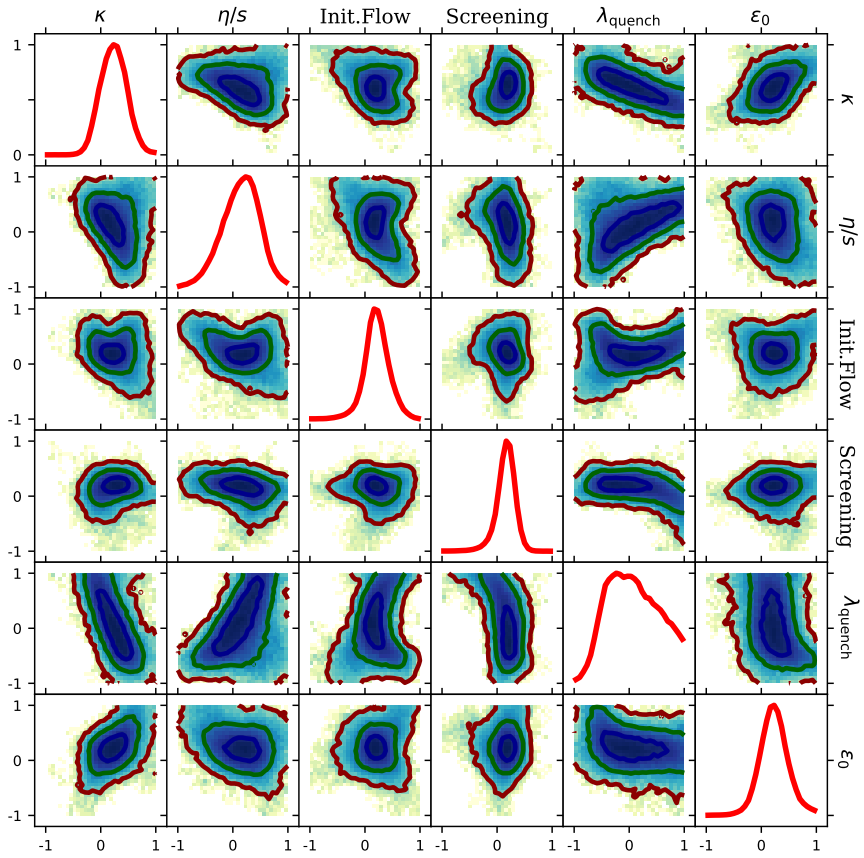
```
compressibility           uniform    $\kappa$
etaovers                  uniform    $\eta/s$
initial_flow              uniform    Init.Flow
initial_screening         uniform    Screening
quenching_length          uniform    $\lambda_{\rm quench}$
initial_epsilon           uniform    $\epsilon_0$
\end{verbatime}}
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\_PROJECT\}/figs/observable\_info.txt}, and the prov
{\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 $\mathbf{1}-\boldsymbol{\sigma}$, $\mathbf{2}-\boldsymbol{\sigma}$ and $\mathbf{3}-\boldsymbol{\sigma}$ 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.

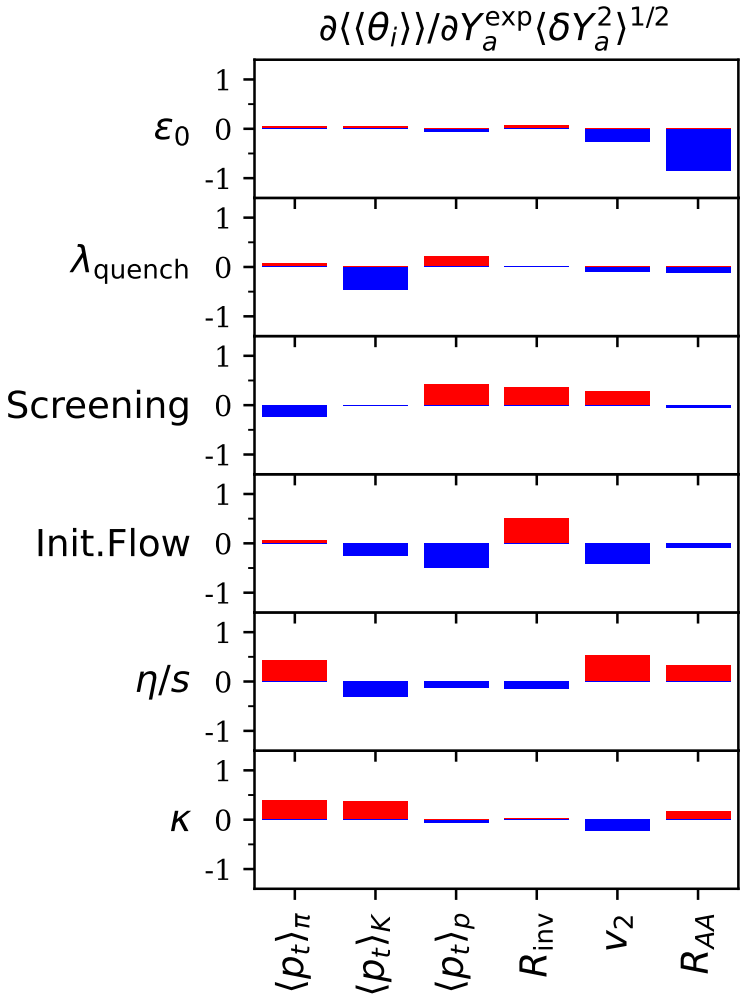
Similarly, one can plot the resolving power. The User can visit the directory `figs/resolvingpower/`. The User should copy the files `mcmc_trace/ResolvingPower.txt` and `Info/modelpar_info.txt` to this directory, editing the `Info/modelpar_info.txt` as was done for the posterior visualization figures 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
```

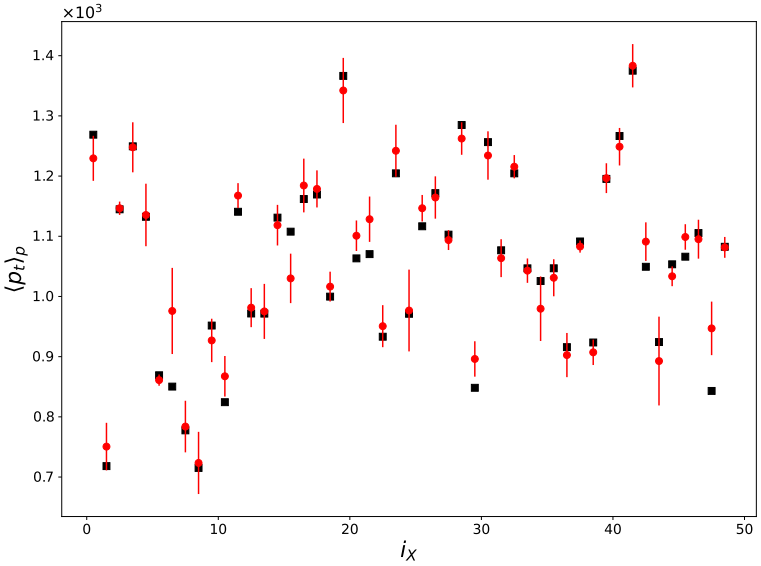
If the uncertainty were perfectly stated, 68% of the points would be within one standard deviation.

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.