

8 Template-Based Tutorial

8.1 Overview

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

This section describes the steps of how the User would

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

8.2 Installation

After completing the necessary prerequisites listed in section ??[Installation] and following the steps outlined in section ??[Prerequisites] to install the required cmake, eigen, and gsl libraries, and setting the Home Environment Variable by creating the Home Directory as described in section ??[Making Home Directory and Setting Home Environment Variable], the user must proceed to clone the smooth and commonutils directories and compile the libraries, as explained in sections ??[Downloading] and [Compiling Libraries].

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

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

8.3 Creating Necessary Info Files

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

par1	uniform	0.000	100.0
par2	uniform	0.000	100.0
par3	uniform	0.000	100.0
par4	uniform	0.000	100.0

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

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

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

The first entry in each line simply provides the names of the observable which will be processed in the Bayesian analysis. The third entry is used by *Smooth Emulator* during tuning, but only if a Monte Carlo method is used. The spread, $\sigma_{\mathbf{A}}$, describes the variance of the output due to a single term in the Taylor expansion. Because this is treated as a random variable, and varied from one sampling of the coefficients to another, one must have an initial choice for it before the MCMC procedures in the tuning can proceed. One simply needs to choose this value within a few factors of two from the optimized value. Otherwise, the initial burn-in stage of the MCMC might require more time to coverge in the right neighborhood. Once the burn-in of the MCMC procedure is completed, this parameter is not used. If the analytical method is used for tuning (which is recommended) this parameter is irrelevant.

8.4 Running *Simplex Sampler*

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

```
#Simplex_LogFileName    simplexlog.txt # comment out to direct output to screen
Simplex_TrainType       1              # 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 $n = 4$ 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_PROJECT}/modelruns/`.

Now the user can run *Simplex Sampler*, which must be run from the project directory,

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

If all goes well there is no screen output. The program writes information about the training points in the `modelruns/` directory. Changing into that directory, there should now be five directories, corresponding to the training points at 5 places: `modelruns/run0`, `modelruns/run1`, `modelruns/run2`, `modelruns/run3`, and `modelruns/run4`. 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. If one had set the parameter `Simplex_TrainType` to 2, there would be 15 sub-directories in `modelruns/`. The next step will be to run the full model for the parameters in each directory. Thus for `Simplex_TrainType=1`, one would need 5 full-model runs, and for `Simplex_TrainType=2`, one would need to do 15 full-model runs. The corresponding observables will be written in the files `modelruns/runI/obs.txt`

8.5 Running the Fake Full Model

Once the training points have been generated, the user will input a Real full model based on the given structure, tailored to address their specific problem. For the tutorial, a fake model is provided. It reads the model-parameter values in each `modelruns/runI/mod.parameters.txt` file and writes the corresponding observables in `modelruns/runI/obs.txt`.

The fake model is a simple python script, which is enacted by entering the command:

```
${MY_PROJECT}% python3 ${MY_LOCAL}/fakemodel/templatemod.py
```

The output should appear as

```

${MY_PROJECT}% python3 ${MY_LOCAL}/fakemodel/templatemod.py
${MY_PROJECT}% Writing: length 10.871583624279312 1.0
${MY_PROJECT}% Writing: mass -48.906793378963606 1.0
${MY_PROJECT}% Writing: time -50.76565279455246 1.0

${MY_PROJECT}% Writing: length -27.67701646408149 1.0
${MY_PROJECT}% Writing: mass -50.35567713223607 1.0
${MY_PROJECT}% Writing: time -34.04264570717603 1.0
.
.
```

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

```

length 10.871583624279312 1.0
mass -48.906793378963606 1.0
time -50.76565279455246 1.0
```

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

8.6 Running *Smooth Emulator*

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

```

#SmoothEmulator_LogFileName smoothlog.txt
SmoothEmulator_LAMBDA 3.0
SmoothEmulator_MAXRANK 4
SmoothEmulator_NASample 8 # No. of coefficient samples
SmoothEmulator_TuneChooseMCMC true # set false if NPars<5
SmoothEmulator_UseSigmaYRreal false #
SmoothEmulator_ConstrainAO false
SmoothEmulator_CutoffA false
SmoothEmulator_ModelRunDirName modelruns
SmoothEmulator_CoefficientsDirName coefficients
SmoothEmulator_ModelParInfoDir Info
```

```

SmoothEmulator_ObservableInfoDir Info
SmoothEmulator_TrainingPts 0-4
SmoothEmulator_MCStepSize 0.01
SmoothEmulator_MCSigmaAStepSize 50.0
SmoothEmulator_NMC 10000 # Steps between samples
SmoothEmulator_UsePCA false

```

The parameters are described in detail in Sec. ?? . For the purpose of this tutorial we review a few of them. The parameter `SmoothEmulator_TrainingPts` is set above to include all five of the training points. However, if *Simplex Sampler* were run with a different choice, or if the User had added more points to be used by some other means, this parameter should be adjusted. The three parameters, `SmoothEmulator_MCStepSize`, `SmoothEmulator_MCSigmaAStepSize` and `SmoothEmulator_NMC` might be adjusted depending on the rate of Monte Carlo convergence of the tuning.

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

```

${MY_PROJECT}% ${MY_LOCAL}/bin/smoothy_tune parameters/emulator_parameters.txt

```

```

Tuning Emulator for length

```

```

success percentage=30.900000, SigmaA=170.951454, logP/Ndof=-0.604220,BestLogP/Ndof=-0.4826
success percentage=29.980000, SigmaA=842.098167, logP/Ndof=-0.731110,BestLogP/Ndof=-0.5440
success percentage=30.840000, SigmaA=184.488302, logP/Ndof=-0.651830,BestLogP/Ndof=-0.5288
success percentage=31.330000, SigmaA=151.392117, logP/Ndof=-0.784629,BestLogP/Ndof=-0.5839
success percentage=30.120000, SigmaA=120.719517, logP/Ndof=-0.588851,BestLogP/Ndof=-0.4810
success percentage=29.950000, SigmaA=178.510175, logP/Ndof=-0.671494,BestLogP/Ndof=-0.4950
success percentage=29.710000, SigmaA=281.210451, logP/Ndof=-0.750559,BestLogP/Ndof=-0.6051
success percentage=29.930000, SigmaA=171.487852, logP/Ndof=-0.599591,BestLogP/Ndof=-0.5212

```

```

Tuning Emulator for mass

```

```

success percentage=19.490000, SigmaA=53.649126, logP/Ndof=-0.625247,BestLogP/Ndof=-0.49247
success percentage=19.910000, SigmaA=84.204171, logP/Ndof=-0.656591,BestLogP/Ndof=-0.52713
success percentage=17.380000, SigmaA=46.138790, logP/Ndof=-0.780594,BestLogP/Ndof=-0.57353
success percentage=18.750000, SigmaA=92.444543, logP/Ndof=-0.531562,BestLogP/Ndof=-0.50588
success percentage=19.190000, SigmaA=60.575128, logP/Ndof=-0.540636,BestLogP/Ndof=-0.45743
success percentage=17.920000, SigmaA=75.219279, logP/Ndof=-0.674120,BestLogP/Ndof=-0.53911
success percentage=18.360000, SigmaA=40.116381, logP/Ndof=-0.656642,BestLogP/Ndof=-0.55767
success percentage=19.470000, SigmaA=74.261850, logP/Ndof=-0.675975,BestLogP/Ndof=-0.56443

```

```

Tuning Emulator for time

```

```

success percentage=19.760000, SigmaA=82.486706, logP/Ndof=-0.669089,BestLogP/Ndof=-0.48988
success percentage=19.970000, SigmaA=44.492860, logP/Ndof=-0.632434,BestLogP/Ndof=-0.49769
success percentage=19.520000, SigmaA=80.956164, logP/Ndof=-0.682640,BestLogP/Ndof=-0.54988
success percentage=19.340000, SigmaA=70.625835, logP/Ndof=-0.668105,BestLogP/Ndof=-0.51880
success percentage=19.590000, SigmaA=116.988915, logP/Ndof=-0.673993,BestLogP/Ndof=-0.5827
success percentage=19.080000, SigmaA=89.891122, logP/Ndof=-0.631130,BestLogP/Ndof=-0.50822
success percentage=19.880000, SigmaA=102.710384, logP/Ndof=-0.707850,BestLogP/Ndof=-0.5426
success percentage=19.450000, SigmaA=206.622539, logP/Ndof=-0.800493,BestLogP/Ndof=-0.4865

```

If the success percentage above is small, much less than 50%, then one may wish to decrease the Monte Carlo stepsizes, `SmoothEmulator_MCStepSize` and `SmoothEmulator_MCSigmaAStepSize` 0.01, to increase the success rate of the Metropolis steps. If the percentage is large, much greater than 50%, conversely, one may wish to increase the step size to increase the coverage of the Monte Carlo trace. The quantity `SigmaA` represents the characteristic width of the distributions of coefficients. It is allowed to vary, and from experience, the range can be quite large. If the value of `SigmaA` does not seem to have randomized, one may wish to increase the number of Monte Carlo steps, `SmoothEmulator_NMC`. The quantity `logP/Ndof` is the logarithm of the weight for a coefficient set divided by the number of degrees of freedom (number of coefficients). It should fluctuate throughout the trace. Higher values (usually less negative) indicate better fits. If the value is well below -1, and rising throughout the trace, it suggests the burn-in was insufficient and `SmoothEmulator_NMC` should be increased. The last entry in the output lines, `BestLogP/Ndof`, simply reports the highest weight encountered in the trace. Again, if that is significantly increasing throughout the trace, `SmoothEmulator_NMC` should be increased.

To represent the uncertainty, there were N_{sample} sets of coefficients generated by *Smooth Emulator*. In this case, the parameter `SmoothEmulator_NASample` was set to 8. As the Monte Carlo trace proceeds, after `SmoothEmulator_NMC` steps, the coefficients are stored, and an output line is generated as seen above. Thus there are 8 lines printed above for each observable.

The program generates Taylor coefficients for N_{sample} samples and saves them in the `coefficients` directory. Each observable has its own sub-directory with its name. Besides the coefficients, *Smooth Emulator* also produces a `meta.txt` file in the same directory. It includes essential information like the observable's maximum rank and net number of model parameters. This file is used to access the coefficients later. In this case, `smoother_tune` created the directories, `${MY_PROJECT}/coefficients/length`, `${MY_PROJECT}/coefficients/mass` and `${MY_PROJECT}/coefficients/time`. Within each of these sub-directories `smoother_tune` created files `meta.txt`, `sample0.txt`, `sample1.txt` ... `sample7.txt`.

9 Generating Emulated Observables

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 `smoother_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 `par1=20`, `par2=40`, `par3=60` and `par4=80`,

```

${MY_PROJECT}% ${MY_LOCAL}/bin/smoothy_calcobs parameters/emulator_parameters.txt
Prior Info
#      ParameterName Type      Xmin_or_Xbar  Xmax_or_SigmaX
0:      par1      uniform          0           100
1:      par2      uniform          0           100
2:      par3      uniform          0           100
3:      par4      uniform          0           100
Enter value for par1:

```

```

20
Enter value for par2:
40
Enter value for par3:
60
Enter value for par4:
80
length = 61.349 +/- 5.14857
mass = -26.4496 +/- 2.6206
time = -44.9779 +/- 2.0495

```

One can test the emulator by entering the coordinates of training point. For example, one of the training points is $\text{par1}=50$, $\text{par2}=50$, $\text{par3}=88.7298$ and $\text{par4}=40$. Running `smoothery_calcobs` for that coordinate,

```

${MY_PROJECT}% ${MY_LOCAL}/bin/smoothery_calcobs parameters/emulator_parameters.txt
Prior Info
#      ParameterName Type      Xmin_or_Xbar  Xmax_or_SigmaX
0:      par1      uniform          0           100
1:      par2      uniform          0           100
2:      par3      uniform          0           100
3:      par4      uniform          0           100
Enter value for par1:
50
Enter value for par2:
50
Enter value for par3:
88.7298
Enter value for par4:
40
length = -45.878 +/- 6.7435e-07
mass = -7.8083 +/- 8.42937e-08
time = -52.4031 +/- 1.16801e-06

```

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

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

```

#include "msu_commonutils/parametermap.h"

```

```

#include "msu_smooth/master.h"
#include "msu_commonutils/log.h"
using namespace std;

int main(int argc, char *argv[]){
    if(argc!=2){
        CLog::Info("Usage smoothy_calcobs emulator parameter filename");
        exit(1);
    }
    CparameterMap *parmap=new CparameterMap();
    parmap->ReadParsFromFile(string(argv[1]));
    CSmoothMaster master(parmap);
    master.ReadCoefficientsAllY();

    CModelParameters *modpars=new CModelParameters(master.priorinfo); // contains info about

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

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

    return 0;
}

```

This illustrates how one can write a code that

- a) Reads the parameter file
- b) Creates a *master* emulator file (called master because it includes emulators for all the observables)
- c) Creates a model-parameters object, **modpars**, that stores the coordinates of the model-parameter point
- d) Calculates the observables from the emulator