5 Tuning the Emulator

5.1 Summary

Smooth emulator finds a sample set of Taylor expansion coefficients that reproduce a set of observables at a set of training points. The process of finding those coefficients is referred to as "tuning". For a given observable, a particular sample set of coefficients gives the following emulated function:

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

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

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

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

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

where σ_A is varied to maximize the overall probability given the constraint of reproducing the training points. More discussion is provided in Sec. ??. Whereas *Smooth Emulator* does a nice job of finding an optimum value for σ_A 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. (??). The User sets the number of sets of coefficients, typically of order $N_{\mathrm{sample}} \approx 10$, 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 $N_{\textbf{sample}}$ 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 $N_{\textbf{sample}}$ sets of coefficients from the training data, then stores $N_{\textbf{sample}}$ 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, $\vec{\theta}_{\mathbf{train}}$, at each training point. These points can be generated by *Simplex Sampler*, as described in Sec. **??**, or they can be generated by hand. If the number of full-model runs performed is $N_{\mathbf{train}}$, Smooth emulator requires files for each run. Each file is named `${MODEL_RUN_DIRNAME}/runI/mod_parameters.txt`, where $0 \leq I < N_{\mathbf{train}}$, and I denotes the point in parameter space for the $I^{\mathbf{th}}$ full-model training run. The directory `${MODEL_RUN_DIRNAME}/` is typically `${MY_PROJECT}/modelruns`, but can be defined otherwise (see below). For example the file `modelruns/run0/mod_parameters.txt` might look like

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

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

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

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

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. **??**.
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 $N_{\textbf{sample}}$ 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 $N_{\textbf{sample}}$ sets of coefficients. Each set reproduces the training points, but differs away from the training points. Setting $N_{\textbf{sample}} \sim 10$ 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 $N_{\textbf{sample}}$ 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:

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

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 $\boldsymbol{0 \leq I < N_{\mathrm{sample}}}$. 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 $\vec{X}$ 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 $\boldsymbol{\Lambda}$, 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 t
```

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();`