Chapter 1

Code Usage for Malleus

1.1 Introduction

For the NCD phase of SNO, the first signal extraction used a Markov Chain Monte Carlo (MCMC) method [1], then new to the SNO project. For the Day/Night aspect of the NCD phase, we decided to use this method again, as it has many benefits. It properly integrates "nuisance parameters" (systematics), is able to handle a large number of parameters and is "embarrassingly parallel", i.e. very easy to run across multiple computers simultaneously. With so many parameters (more than 50 for the full data set with all systematics and backgrounds), this is a crucial feature. The code used for that signal analysis, with the SNO-specific parts removed, became Malleus. It was designed from the start to be flexible enough to be used for other experiments with minimal change, ideally with only re-writing configuration files. How to write these configuration files and make needed changes is described in this document.

At its core, Malleus runs the Metropolis algorithm, discussed in Chapters 2 and 3. The likelihood function evaluated is the Extended Likelihood, with the "parent" pdf that the data is compared against generated by binning MC simulation events. Some of the fit parameters are systematics, which are applied to the MC events at the time of binning, so the resultant histogram is rebuilt at each step in the Metropolis algorithm. The output is the posterior distribution for each of the parameters. This can then be fit if one chooses; a program to fit gaussians to the results is supplied with the main program.

1.2 What's in the box

In its current incarnation, Malleus compiles five executables:

- `Malleus` is the principal program. This actually runs the MCMC
- `metaConfig.exe` is a conversion program to translate from the "meta" config file format (more human readable) to the config file format used by `Malleus`. Malleus can also directly read these meta files, but the other programs (`autoFit.exe`, etc) cannot.
- `autoFit.exe` fits Gaussians (with some options) to the results of a run of `Malleus`
- `getAutoCorr.exe` computes the autocorrelation of each parameter in the MCMC results, useful for finding optimal step sizes
- `drawResults.exe` creates a set of plots showing binned data v. binned MC for the parameters its given, useful for checking reasonableness of results

Of these, only `Malleus` is actually *necessary*. We highly recommend using `metaConfig.exe` as well, as the "native" config file format used by `Malleus` is very error-prone when written by hand. The other three are post-processing utilities, included for convenience.

The code itself is C++ and compiles against CERN's Root, available at `http://root.cern.ch` [2]. If Root is installed properly, simply running `make` should be sufficient to build

the program. Unfortunately, due to speed requirements it may be necessary for the user to add systematic functions, which must be present at compile time (i.e. the code itself must be edited). This should be a minimal change, and is detailed later.

1.3 Malleus

This section details the usage of the main program. This description assumes that you are writing meta files rather than directly writing config files. In addition, it assumes a basic working knowledge of the Metropolis algorithm and the MC events → pdf process, described in Chapters 2 and 3. Since the code is still a work-in-progress, I'll strive to highlight potential pitfalls and known places where the error handling may not catch problems.

1.3.1 Objects

This term has two meanings here: objects in the sense of "Object-Oriented" (i.e. C++ classes) and in the sense of things in the program you can manipulate. Here we discuss the latter. The code has several "layers" of objects.

At the top is the MCMC itself. It keeps track of the MCMC status (step, parameter values, likelihood value), the "pure" parameters (numbers that are varied, but not associated with any other object; they are independent objects in this description), and computes the Likelihood functions not associated with another object (these are called LogLikelihoodFormulas, and are also independent objects). The MCMC also handles I/O. Options can be passed to the MCMC directly to control its behavior, called directives.

The MCMC contains a number of Pdfs, each of which serves one function: it takes a list of the current parameter values, and returns an Extended Likelihood. These are then summed and added to those computed by the MCMC. Each Pdf corresponds to and keeps track of a data set $\{\vec{x}\}$; the dimension of events $\vec{x}_i$ determines the dimension of the Pdf. Currently, 1-D and 3-D Pdfs are available. The binned pdf created from the MC data is also stored in Pdf and has the same number of dimensions as the data. Each dimension has an Axis, which is an object in its own right. In addition, the Pdf needs to know what branches are present in the MC events. If multiple MC sets are used, they all need to have those branches. This may require the creation of dummy branches.

Each Pdf contains a number of Fluxes and Systematics, and a number of Axes equal to Pdfs dimension. Each Flux keeps track of a set of MC events and its normalization. Each Systematic is a function applied to a subset of the Fluxes on an event-by-event basis. These are the "lowest level", containing no further (user-accessible) parts.

1.3.2 Parameters, Names and Keys

At each step in the MCMC, each element of the set of parameters $\vec{\alpha}$ is incremented. This is still true for the code implementation, though a step width of zero (no step) is allowed. However, $\vec{\alpha}$ can be more complicated here - it is, in some sense, just a collection of real numbers that are varied, that the user can combine in almost any way he or she sees fit.

Parameters can be created explicitly as "pure" parameters (see section 1.3.7), i.e. parameters not associated with any object. Additionally, whenever a Flux or Systematic is created, a parameter with the same name is created automatically. For Fluxes these automatic parameters have a special meaning (see sections 1.3.9 for details). These parameters can be combined using the AsymmFunc system described in section 1.3.11. Since the program keeps track of everything by its Name, each parameter must have a unique Name, including those that are automatically generated. So each Flux, Sys, and Parameter must be uniquely named.

In the meta file, everything is described by key pairs (which become unique when expanded to the Config file). Except for `new`, every command in the meta file must be of the form `Key=Value` or `Key = Value` (these spaces are ignored), where `Value` is either an integer, a floating point number (read in as a `Double_t`) or a string (which cannot have spaces in it); which one it is depends on the `Key` being used. See section 1.3.4 examples of use, and the appropriate section below for valid Keys for a particular object.

1.3.3 Program behavior

The program begins by reading in the config file and using that information to create and set up all the appropriate objects, and set up the output file and TTree. It then initializes all parameters and computes the Log Likelihood (LL) for these initial parameter values, to create the "step zero" starting point for the MCMC.

The code then runs the Metropolis algorithm. At a given step, the code:

1. Creates a proposed value for each parameter by adding to it a draw from a Gaussian of mean zero and width specified by the user (constant throughout the program), giving $\tilde{\vec{\alpha}}$
2. Computes any parameters whose values are altered by an AsymmFunc
3. Computes the LL contribution from any LogLikelihoodFormulas and checks to see if any pure Parameters are outside their minimum or maximum (if specified), and computes their constraints' contributions to the LL
4. Gives a Pdf the current list of parameter values and asks it for its contribution to the LL. It does this for each Pdf in order of appearance in the meta file (this order shouldn't matter). Each Pdf:
   (a) Checks to see if any parameters it "owns" are out of their bounds
   (b) Rebuilds the MC pdf by:
      i. Emptying the histogram
      ii. Taking an MC event from a Flux
      iii. Passing this MC event through each Sys that affects it, in the order they appear in the meta file (order may matter here), giving the new value for the event
      iv. Filling this event into the histogram (only using those values which correspond to Pdf dimensions), with a weight of `InternalUseWeight*FluxName/TimesExpected` (see 1.3.9)
      v. Repeating this for each event in the Flux
      vi. Repeating this for each Flux (in the order of appearance, though order shouldn't matter)
      vii. Checking that no bins are zero or negative, and setting any that are to a nominal value (currently 1e-10), to avoid problems with taking the logarithm
      viii. Dividing each histogram bin by that bin's volume (width, area, etc) to "normalize" it to a proper un-normalized pdf
   (c) Takes each data point $\vec{x}_i$, asks the MC pdf for its value at that point to get $p(\vec{x}_i|\tilde{\vec{\alpha}})$ and computes the LL
   (d) Computes the LL contributions from constraints on any parameters it "owns"
   (e) Returns this LL
5. Compares the computed LL with the LL from previous step to decide whether to accept the step (see Chapter 2)
6. If step is accepted, updates parameter list, if not, discards proposed parameters
7. Records current parameter list in TTree
8. Every 100th step, AutoSaves TTree
9. After all steps are taken, saves TTree to output file and exits

1.3.4 Basic meta file

The only control we have over the process in section 1.3.3 is in setting up the system. Via the mechanism of the config file, we can create parameters, set step sizes, control the behavior of the MCMC to a certain degree, set up AsymmFuncs and LogLikelihoodFormulas, command the creation of any combination of objects, etc. But then the code runs its prescribed process. The config files themselves must have unique keys for each command, accomplished via a numbering system. This turned out to be difficult to keep track of, since misnumbering caused odd behavior, so I created the much more readable meta files. Running `metaConfig.exe` converts from meta file to config file. Malleus can read these meta files directly (with the -m switch) or the config file (with the -c switch), but the other programs can only read the config files.

Every meta file consists of some number of directives to the MCMC, to control how many steps to take, where to save files, etc; the creation of at least one Pdf, and at least one Flux inside that Pdf, plus a number of Axis objects equal to the dimension of the Pdf. Most also have some Systematics as well, and a few pure Parameters. An example of a simple metafile is:

```
MCMC_ChainLength=5000
MCMC_PrintFrequency=100
OutputFilename=results_test.root

new Pdf
Name=SimpleTest
Dimension=1
MCBranch=Energy
DataFile=fakeData.root
DataTree=data

new Axis
Name=Energy
Bin=0
Bin=0.1
Bin=0.3
Bin=0.7
Bin=1

new Sys
Name=AddConstant
Init=0.1
Width=0.01
Mean=0.1
Sigma=0.05
Function=AddConst
Target=Energy
MCMCParameterValue=AddConstant

new Flux
Name=Sim
File=fakeMC.root
Tree=mc
TimesExpected=5
Min=0
Init=1
Width=0.01
```

In general, a new object is created with the `new` command. `metaConfig.exe` knows about the heirarchy of objects, so any Axis, Flux, etc. created will go in the Pdf most recently created. Between one invocation of

`new` and the next, commands apply to the "active" object. So `Name=AddConstant` above gives the Systematic the name "AddConstant". Note that commands to the MCMC directly and the creation of pure Parameters can occur anywhere and ignore heirarchy rules.

Working through the sample file one line at a time (skipping repeats):

`MCMC_ChainLength=5000` tells the MCMC to take 5000 total steps

`MCMC_PrintFrequency=100` tells the MCMC to print its status to stdout at every 100th step, useful if something goes wrong

`OutputFilename=results_test.root` sets the output file. This defaults to the current directory. This file will contain a single TTree named Tmcmc, this can be overridden (see 1.3.6)

`new Pdf` creates a new Pdf, at this stage with undefined characteristics

`Name=SimpleTest` gives the Pdf the name "SimpleTest"

`Dimension=1` sets SimpleTest to be a 1-D Pdf

`MCBranch=Energy` tells SimpleTest that its Fluxes will contain the branch Energy. If we wanted more branches (for example, if there was a branch "TrueEnergy" that was the input simulation energy, useful for altering resolutions), we just add an additional line `MCBranch=TrueEnergy`

`DataFile=fakeData.root` tells SimpleTest that its data is located in the file fakeData.root

`DataTree=data` tells SimpleTest to look for a TTree named data for its data

`Bin=0` Tells the Energy Axis the location of bin boundaries. In this case, it has 4 bins, $[0, 0.1)$, $[0.1, 0.3)$, $[0.3, 0.7)$ and $[0.7, 1]$

`Init=0.1` tells the MCMC that the parameter AddConstant takes the value 0.1 initially, before stepping starts

`Width=0.01` tells the MCMC that the parameter AddConstant should be incremented by a Gaussian draw with $\sigma = 0.01$ at each step

`Mean=0.1` and `Sigma=0.05` tell the pdf SimpleTest that the parameter AddConstant has a Gaussian constraint of mean 0.1 and $\sigma = 0.05$, which is added as a penalty term to the Likelihood computed by SimpleTest

`Function=AddConst` tells the AddConstant Sys to use the function AddConst (defined in `FunctionDefs.h`). See Section 1.3.10 for details.

`Target=Energy` sets the AddConstant Sys to alter the branch Energy

`MCMCParameterValue=AddConstant` sets the first (and in this case only) parameter of the AddConst function to be the automatically created variable AddConstant

`File=fakeMC.root` tells the Flux Sim that the MC events for it are in the file fakeMC.root

`Tree=mc` tells Sim to look for a TTree named mc for its events

`TimesExpected=5` tells Sim that it has 5 times as many MC events as it expects for data events, i.e. its simulation had "five experiments" worth of simulated events. This acts as a normalization, more on that in Section 1.3.9

`Min=0` tells the MCMC that the parameter Sim has minimum 0. If it is ever below this value, it returns Likelihood zero (i.e. it cannot take this step). Since we're using Log Likelihoods, it actually returns a very large negative value (but not negative infinity)

1.3.5 new

As mentioned, the `new` command creates a new object. Valid objects to create are: `Pdf`, `Sys`, `Flux`, `Axis`, `Parameter` and `LogLikelihoodFormula`. They must obey the heirarchy: to create a Flux, Sys, or Axis, a Pdf must already exist for it to "live in". At any given time, only the most recently created object is "active" and any commands given apply to it. MCMC Directives ignore this rule and can be anywhere in the file, but I've found it easiest to just include all of them at the beginning of the file to prevent confusion later.

1.3.6 MCMC Directives

A number of commands can be given to the MCMC to change its behavior. A few of these are necessary and showed up in the sample file. If a command isn't present, a default value is assumed, listed with that command below. A full listing of these commands and their consequences is:

`MCMC_ChainLength=integer` sets the total length of the chain. DynamicSteps are taken first, then normal steps. If this number is less than `MCMC_DynamicSteps`, it is ignored. If it isn't present, the MCMC defaults to 1000 steps.

`MCMC_DynamicSteps=integer` tells the MCMC to take steps where the step size is allowed to vary, with the goal that the chain take 23.5% steps. This was implemented for testing and is described in Section 2.5. Default is 0.

`MCMC_PrintFrequency=integer` controls how often the MCMC prints its status to the screen, setting it to print every n^{th} step. Useful for making sure the chain is still going and checking on its status as it is running. Default is 1 (i.e. every step).

`MCMC_SkipSteps=integer` originally was going to control how many "burn-in" period steps to skip when making graphs, but all of the graphing programs take that input separately and ignore this now. Defaults to 0.

`MCMC_UseAsymmetry=true/false` tells the MCMC whether to check for AsymmFuncs. It *must* be set to `true` if AsymmFuncs are being used. See section 1.3.11 for more details. Default is `false`.

`MCMC_SaveProposed=true/false` tells the MCMC whether to save the proposed values for each step as well as the accepted. These are the values $\vec{\tilde{\alpha}}$ generated by varying the parameters. If the step is rejected, these values are lost if this set to `false`, but the output files are half the size. Useful for debugging. Default is `true`. These values are saved as `ProName`, where Name is the parameter name, while the accepted values are `AccName`.

`MCMC_SaveUnvaried=true/false` tells the MCMC whether to save the value of parameters with width ≤ 0, i.e. those that aren't varied by the MCMC. Often useful for debugging. Setting this to `false` gives smaller output files. Default is `true`.

`RandomSeed=integer` sets the seed for the TRandom3 that generates the random numbers needed throughout the MCMC process. Default is 0, i.e. a random seed based on the clock time, among other things.

`OutputDirectory=string` sets the directory to which MCMC writes output files. Default is `.` (current directory).

`OutputFilename=string` sets the output filename. It is concatenated with the OutputDirectory internally, so putting the directory string as part of the file name has the same result as setting `OutputDirectory`. Default is `results.root`.

`OutputTreeName=string` sets the name of the TTree created in the output file. Default is `Tmcmc`. Note that `autoFit.exe` assumes that the tree is named `Tmcmc`.

1.3.7 Parameter

A "pure" parameter is one not associated with another object. It is simply a number that is varied when the parameters are varied by the MCMC. It can be altered with an AsymmFunc. To create one, use `new Parameter`. Required settings are `Name`, `Init` and `Width`. `Name` must be unique. If `Width` is ≤ 0, this Parameter will not be varied (though AsymmFunc can still change its value). Valid Keys are:

`Name=string` sets the name. Must be unique. If this isn't present, an error message is generated and setup fails.

`Init=double` sets the initial value. Error and setup failure if not present.

`Width=double` sets σ for the Gaussian drawn from at the incrementing step for this variable. If `Width` is ≤ 0, parameter is not incremented. Error and setup failure if not present.

`Mean=double` creates a Gaussian constraint for this parameter. Sets μ of this constraint to `Mean`. If this is present and `Sigma` is not, an error is generated and setup fails. If neither is present, no constraint. See `Sigma` for more details.

`Sigma=double` creates a Gaussian constraint for this parameter. Sets σ of this constraint to `Sigma`. This constraint gives LL contribution $-\frac{(\alpha-\mu)^2}{2\sigma^2}$. If this is present and `Mean` is not, an error is generated and setup fails. If neither is present, no constraint.

`Min=double` sets the minimum value for the parameter. If its value goes below `Min`, the returned LL is $-1e200$, effectively $-\inf$, so no step is taken. If this is not present, no minimum.

`Max=double` sets the maximum value for the parameter. If its value goes above `Max`, the returned LL is $-1e200$, effectively $-\inf$, so no step is taken. If this is not present, no maximum.

`AsymmFunc=string` creates an Asymm function for this parameter. See section 1.3.11. If this is not present, no AsymmFunc.

`AsymmPar=string` sets a parameter for the Asymm Func. See section 1.3.11. If this is present and `AsymmFunc` isn't, an error is returned and setup fails.

1.3.8 Pdf

A Pdf is a basic building block of the system. It contains a data set and the corresponding histogram for the binned probability distribution for that data. Its dimension must be specified and a number of Axis objects equal to that dimension must be created. The data must be stored in a TTree in a `.root` file, and only the branches that have the same name as an Axis will be used. All events in the data set *must* be in the range of the bins defined in the Axis. If this isn't the case, the behavior is undefined (and is likely to be a program crash, or at least very strange behavior), since this will involve requesting binned pdf values outside the range where they are defined. Any number of `MCBranches` can be defined, these describe the branches that the MC events in the various Fluxes will have. This will cause all Fluxes to have the same branches, and only those branches will be used. Each Axis name must have a corresponding MCBranch and each set of MC events must have branches with the same names as the Axis objects. Valid Keys are:

`Name=string` sets the name. Must be unique. If this isn't present, an error message is generated and setup fails.

`Dimension=integer` sets the dimension of the Pdf. A number of Axis variables equal to this `Dimension` must be created in the Pdf. Only dimensions 1 and 3 are currently available. If this isn't present, an error message is generated and setup fails.

`DataFile=string` sets the location of the file containing the data events. Can include a directory string, assumes current working directory otherwise. The file must contain the TTree named in `DataTree`. If this isn't present, an error message is generated and setup fails.

`DataTree=string` sets the name of the TTree containing the data events, in the file `DataFile`. Must contain branches with the same names as the Axis objects. If this isn't present, an error message is generated and setup fails.

`MCBranch=string` tells the Pdf to look for a branch of this name in the TTrees containing the MC events in the Flux objects. To have the Pdf look for more than one branch (necessary for 3D Pdfs, useful for any Pdf), simply call this for each branch. They will be numbered (in the order called) internally. Must be called at least a number of times equal to the `Dimension` of the Pdf.

1.3.9 Flux

A Flux object must live in a Pdf object, so a Pdf must be created (with `new`) before a Flux is created. Each Flux contains one set of MC events. When this is created, a parameter with the same `Name` as the Flux is created. During the filling of the MC binned pdf, the Flux hands events to the Pdf, and they are

filled in to the Pdf with weight `Name*InternalUseWeight/TimesExpected`. `InternalUseWeight` is an internally created MC branch that is used for reweighting procedures, see section 1.3.10 for more details. The `TimesExpected` nominally is used to keep track of how many "experiments worth" of MC you have, i.e. if the MC contains 500 times the events you expect for that particular signal in your data, a `TimesExpected` of 500 gives a nominal value of 1 to the automatic parameter. Another option is `TimesExpected` set to the number of events in your analysis window, then the nominal value of the automatic variable is the number of data events. These MC events are not restricted to being in the range specified by the Axis objects; events outside the range are ignored. This is imporant for any Systematic that changes the effective analysis window (say by adding a constant to the energy or position) - if this happens, there need to be MC events in the new region or the log likelihood will not give the correct value. Fluxes can also have a `FluxNumber` assigned, which is a distinguishing feature used by the Sys to decide which Flux to act on (so if you have a systematic that you want to act on this Flux and not others, give it a distinct `FluxNumber`). Any number of Pdfs can share the same `FluxNumber`. Valid Keys are:

`Name=string` sets the name. Must be unique. If this isn't present, an error message is generated and setup fails.

`Init=double` sets the initial value of the automatic parameter. Error and setup failure if not present.

`Width=double` sets σ for the Gaussian drawn from at the incrementing step for the automatic parameter. If `Width` is ≤ 0, parameter is not incremented. Error and setup failure if not present.

`Mean=double` creates a Gaussian constraint for the automatic parameter. Sets μ of this constraint to `Mean`. If this is present and `Sigma` is not, an error is generated and setup fails. If neither is present, no constraint. See `Sigma` for more details.

`Sigma=double` creates a Gaussian constraint for the automatic parameter. Sets σ of this constraint to `Sigma`. This constraint gives LL contribution $-\frac{(\alpha-\mu)^2}{2\sigma^2}$. If this is present and `Mean` is not, an error is generated and setup fails. If neither is present, no constraint.

`Min=double` sets the minimum value for the automatic parameter. If its value goes below `Min`, the returned LL is $-1e200$, effectively $-\inf$, so no step is taken. If this is not present, no minimum.

`Max=double` sets the maximum value for the automatic parameter. If its value goes above `Max`, the returned LL is $-1e200$, effectively $-\inf$, so no step is taken. If this is not present, no maximum.

`AsymmFunc=string` creates an Asymm function for the automatic parameter. See section 1.3.11. If this is not present, no AsymmFunc.

`AsymmPar=string` sets a parameter for the Asymm Func. See section 1.3.11. If this is present and `AsymmFunc` isn't, an error is returned and setup fails. `File` sets the location of the file containing the MC events. Can include a directory string, assumes current working directory otherwise. The file must contain the TTree named in `Tree`. If this isn't present, an error message is generated and setup fails.

`Tree=string` sets the name of the TTree containing the MC events, in the file `File`. Must contain branches with the same names as the MCBranches of the Pdf this Flux is in. If this isn't present, an error message is generated and setup fails.

`TimesExpected=double` weights each MC event when the binned pdf is filled with (automatic variable)*`InternalUseWeight/TimesExpected`. If this isn't present, an error message is generated and setup fails.

`FluxNumber=integer` sets the `FluxNumber` of the Flux. This allows for Sys to be selective in which Fluxes they act on, as one of the settings for Sys is which FluxNumbers to affect. Fluxes can share the same Flux number. Default is -1.

1.3.10 Sys

The Sys are functions that are applied to the MC events before they are filled in to the binned pdf. They can be selectively applied to some Fluxes and not others, and only apply to Fluxes in the Pdf in which the

Sys object resides. A Sys can use any number of MC branches and parameters in its evaluation, but can only alter (target) one MC branch. So each Sys is a function taking some number of real values and returning a single real value. In addition to the MC Branches listed in Pdf, there is an additional, internal branch created named `InternalUseWeight`. It is set to 1, but can be targeted with the Sys to change this value. `InternalUseWeight` is directly multiplied by any other weighting for the MC events, so that it effectively acts a relative weight for MC events. Since multiple Sys may (and usually do) apply to the same Flux (and hence to the same MC event), the Sys function returns not the new value of the branch, but the change in that value due to this Sys's action, so that each Sys can see the values as they originally appear in the Flux. The option to look at the branch values including all changes up to that point (the Sys are applied in the order they appear in the meta file) is also available, but then the order the Sys are applied in matters. To make a Sys apply to only a select subset of the Fluxes, it can be set to only act on certain FluxNumbers. When a Sys is created, a parameter of the same name is automatically created. This parameter does not have to be used in the Sys, but must be unique as it identifies the Sys to the code.

The possible functional forms that can be used by Sys (in the meta file they are selected by using `Function=name`) are defined in `FunctionDefs.h`. A simple selection is included, but will most likely need to be expanded upon to specialize to a specific task. This is done by creating a class that inherits from the class `RealFunction` (defined in `RealFunction.h` and `RealFunction.cxx`). This function needs to know how many parameters it is looking for, which must match the number listed in the configuration file. Note that a function defined here is not restricted to a single Sys entry; if there are multiple Sys that use the same function form, it is best to use a single function for them. An example definition is:

```
class MultiplyConst : public RealFunction {
 public:
  MultiplyConst() {
    nPars = 2;
    parameters = new Double_t[nPars];
  }

  Double_t Eval() {
    return (parameters[0]*parameters[1]);
  }
};
```

Only two things need to be defined: the constructor and the function `Eval`. The `nPars` term defines how many parameters the function takes. The function `Eval` actually defines the systematic function. This can call any function in `TMath` (from Root) and can use `TRandom3` (from Root), though random number draws are very slow and should be used only if necessary. Note that the way that Sys works needs this function to return the *change* in the target value, if `useMultiply=false`. In the above case, if this is the only Sys, the resulting value will be $x_{new} = x_{MC} + a_0 x_{MC}$ (where a_0 is parameters[0]). The `parameters` array has the values of the MCMCParameterValues and the MCBranchValues from the config file. The order is the order selected in the config file, with MCMCParameterValues first and MCBranchValues second (the order is hard-coded). A mis-match between `nPars` and the number of parameters in the parameters array will result in undefined behavior, and could cause a seg fault, as with any array.

Since the Sys are somewhat complicated and at the heart of the pdf generation process, we will present two stereotypical Sys. The first will implement $x_{new} = x_{MC} + a_0 x_{MC}$, i.e. $\Delta x = a_0 x_{MC}$, while the second will multiply the event's weight by a constant. For both, we will assume we have a 1-D Pdf with an Axis named "x", and our MC events contain just one branch (also named "x"). In addition, we will assume we created a parameter with the following meta file command (where the Init and Width aren't relevant to our discussion, but are included for completeness)

```
new Parameter
Name=Const
Init=1
Width=0.1
```

To implement $\Delta x = a_0 x_{MC}$, where a_0 is the Const parameter, the meta file reads

```
new Sys
Name=LinearScale
Init=0
Width=-1
Function=MultiplyConst
Target=x
MCMCParameterValue=Const
MCBranchValue=x
UseMultiply=false
UseOriginalData=true
```

Note that the `UseMultiply` and `UseOriginalData` commands are unnecessary, since they are using the default values. Also, we do not use the automatically generated LinearScale variable, so we do not bother to vary or constrain it.

To multiply the weight by a constant, the meta file reads

```
new Sys
Name=WeightAdjust
Init=0
Width=-1
Function=AddConst
Target=InternalUseWeight
MCMCParameterValue=Const
UseMultiply=true
UseOriginalData=false
```

Here again the automatically generated variable is not used. Note that the AddConst function just returns the value of the parameter, which is what we need. Also, note that the `UseMultiply` and `UseOriginalData` variables are needed, as we are multiplying the InternalUseWeight by a constant. If this were the only such multiplier, the UseOriginalData would be optional, but if multiple ones are present, for the multiplication to work correctly we need it to be `false`.

In the meta file, the Sys's function, target branch to alter, and parameters can be set. In addition, the behavior of the automatic parameter sharing the name of the Sys can be controlled, with the same controls as the Parameters of section 1.3.7. Valid Keys are:

`Name=string` sets the name. Must be unique. If this isn't present, an error message is generated and setup fails.

`Init=double` sets the initial value. Error and setup failure if not present.

`Width=double` sets σ for the Gaussian drawn from at the incrementing step for this variable. If `Width` is ≤ 0, parameter is not incremented. Error and setup failure if not present.

`Mean=double` creates a Gaussian constraint for this parameter. Sets μ of this constraint to `Mean`. If this is present and `Sigma` is not, an error is generated and setup fails. If neither is present, no constraint. See `Sigma` for more details.

`Sigma=double` creates a Gaussian constraint for this parameter. Sets σ of this constraint to `Sigma`. This constraint gives LL contribution $-\frac{(\alpha-\mu)^2}{2\sigma^2}$. If this is present and `Mean` is not, an error is generated and setup fails. If neither is present, no constraint.

`Min=double` sets the minimum value for the parameter. If its value goes below `Min`, the returned LL is $-1e200$, effectively $-\inf$, so no step is taken. If this is not present, no minimum.

`Max=double` sets the maximum value for the parameter. If its value goes above `Max`, the returned LL is $-1e200$, effectively $-\inf$, so no step is taken. If this is not present, no maximum.

`AsymmFunc=string` creates an Asymm function for this parameter. See section 1.3.11. If this is not present, no AsymmFunc.

`AsymmPar=string` sets a parameter for the Asymm Func. See section 1.3.11. If this is present and `AsymmFunc` isn't, an error is returned and setup fails.

`Function=string` selects the function from `FunctionDefs.h` to use.

`Target=string` selects which MC branch is altered by this Sys. Note that if `UseMultiply=False`, the value of this Sys is *added* to the branch, rather than replacing it. If `UseMultiply=true`, the branch value is multiplied by the Sys value. Having some Sys multiply a branch and others add to the same branch results in undefined behavior.

`UseMultiply=true/false` sets the Sys to either add its value to the targeted MC branch (`false`) or multiply the targeted branch by the Sys value (`true`).

`UseOriginalData=true/false` sets the Sys to either take the MC branch values (if applicable) as they appear in the Flux (i.e. the unaltered MC values), if `true`, or after applying all Sys applied thus far, if `false`

`AddFluxAffected=integer` tells the Sys to apply to Fluxes labeled with the flux number selected. Can be called multiple times to affect multiple FluxNumbers. If not called, default behavior is to affect all Fluxes.

`MCMCParameterValue=string` sets a parameter of the Sys's function to the selected MCMC parameter (either pure Parameter or automatically generated parameter). Calling multiple times sets multiple parameters, in order - if a function needs four parameters, four calls will identify all four parameters. Note that if both MCMCParameterValue and MCBranchValue are used (which is normally the case), the MCMCParameterValue terms are first, regardless of order in the config file.

`MCBranchValue=string` sets a parameter of the Sys's function to the selected MC branch. Calling multiple times sets multiple parameters, in order - if a function needs four parameters, four calls will identify all four parameters. Note that if both MCMCParameterValue and MCBranchValue are used (which is normally the case), the MCMCParameterValue terms are first, regardless of order in the config file.

1.3.11 AsymmFunc

The AsymmFunc system is used to alter the values of parameters in user-specified ways, rather than just the random draw used to vary them. The basic idea is that `AsymmFunc` defines a function, using the notation of Root's `TF1`, which is *added* to the parameter's value. The reason for adding rather than replacing is an artifact of how this is implemented: it uses the machinery of Sys. The `AsymmPar` then specify what parameters to use in the function. An example use is:

```
new Parameter
Name=NCFlux
Init=0
Width=-1
AsymmFunc=[0]*[1]-[2]
AsymmPar=BoronFlux
AsymmPar=PMTEfficiency
AsymmPar=NCFlux
```

This takes the Parameter NCFlux and sets its value to be

NCFlux + BoronFlux*PMTEfficiency - NCFlux = BoronFlux*PMTEfficiency

In this case, the NCFlux parameter isn't varied because its value is irrelevant - it is entirely replaced by the combination. This doesn't have to be the case, of course. The notation of `AsymmFunc` is that each number in a set of square brackets is a parameter (of `TF1`); there cannot be spaces in the function definition (due to parsing issues). Any parameter defined in the meta file is valid to use, and the function itself can use any combination of basic arithmetic operations and functions defined in TMath. Random numbers are not available here, which should not be a strong constraint as the parameters are randomly varied. The `AsymmPar` correspond to the parameters `[i]`, numbered in order of appearance starting with 0. There is limited error

checking on the parameter numbers: the system should give an error if too high of a parameter number is requested in the `AsymmFunc`, but will not if a number is missing. This can be problematic for long functions, especially when they are changed; exercise caution.

1.4 metaConfig.exe

This section can be safely skipped by most users. It details what happens during the conversion process from the human-readable meta file to the less-readable (but still human readable, just much harder to follow) config file.

The config file format used by `Malleus` is parsed and read-in by the class described by `ConfigFile.h`. This is the only code reused from [1], as it is an open source utility available at [3]. The basic premise is that the config file consists of Key-Value pairs, similar to the meta file. The major differences are that each Key must be unique, the order that the keys appear in does not matter at all, and the existence of objects is kept track of by a numbering system.

The `metaConfig.exe` system is fairly simple: it keeps a running total of each kind of object present. When the first `Pdf` is created with `new`, the `Pdf` counter is set to 0. Until `new` is called again, the system then adds `Pdf_0_` to the beginning of each command, so that

`Name = FirstPdf`

is converted into

`Pdf_0_Name=FirstPdf`

A special case here is `MCBranch` (or anything else that can be called repeatedly: `AsymmPar`, `MCMCParameterValue`, ...), which has its own numbering system, so that every time it is called it is incremented. As an example

`MCBranch= energy`
`MCBranch=x`

is converted into

`Pdf_0_MCBranch_00=energy`
`Pdf_0_MCBranch_01=x`

When `new` is called again for an object inside a `Pdf`, such as `Axis`, that counter begins incrementing. Then a command is appended with `Pdf_0_Axis_0_`, assuming this is both the first `Pdf` and the first `Axis`. `Bin` is again a special case with its own counting system, but this time the format is slightly different, as

`Bin=1`

becomes

`Pdf_0_Axis_0_Bin00=1`

All of counters under a particular object are reset when `new` is called for that object, so if `new Pdf` is called, the first call of `MCBranch=y` will give

`Pdf_1_MCBranch_00=y`

This pattern is repeated for every object with sub-objects. Those that interact directly with the MCMC, however, are kept track of separately. The MCMC directives are just copied, as they take the same form in both the config and meta files. The `Parameter` counter is never reset, calling `new Parameter` anywhere in the meta file increments it.

One word of caution is that the `metaConfig.exe` program does very little error checking. It will return errors if it encounters something it can't translate into a config file, such as creating a `Axis` without first creating a `Pdf`. It will also return an error if it encounters a command it doesn't recognize - both in the sense of commands that don't conform to the pattern key=value, and in the sense of calling something like `new Banana`, as it does not recognize an `Banana` class. Anything else that fits the valid patterns will be allowed. It is thus essential to run a test run of any config file through `Malleus` to make sure setup completes properly.

1.5 Other Programs

The other three programs included, `autoFit.exe`, `drawResults.exe` and `getAutoCorr.exe`, are simple enough that the built-in help is sufficient. This can be accessed by running either the the program with no arguments or with the argument `--help`.

Chapter 2

Markov Chain Monte Carlo

The main signal extraction presented in Chapter 3 minimizes the difference between the Monte Carlo and the data, varying proportions of the various signals and backgrounds and distortions in the MC (the systematics). This involves a large number of parameters and each step is very computationally expensive, making a typical minimizer such as Minuit [4] impractical. Instead, we turn to the Markov Chain Monte Carlo (MCMC) method. Here we describe the basic ideas of the method, the particular MCMC algorithm we have chosen, and why this method is preferable for this problem.

2.1 Definitions

The Markov Chain Monte Carlo (MCMC) method derives from the principles of Bayesian statistics. We will not go in to the fundamentals of this, which can be found in a number of references [5, 6]. But we will need a few definitions of terms that are commonly used in this field, as the following discussion will rely on this notation. In addition, we define several terms as shorthand or for convenience in the following discussion. To illustrate the meaning of these quantities, we look at a hypothetical problem: we have a model of the weather in Boston, and wish to test its validity.

$p(x)$ is the probability distribution function (pdf) for x. This requires context - if x is Temperature, then $p(x)$ for Boston will be different from $p(x)$ for Orlando, which will be very different than $p(x)$ for a quark-gluon plasma.

$p(x|y)$ is a conditional probability of x, given y (i.e. assuming y is true). Continuing our example, $p(x|\text{summer})$ will have much more weight near 80°F than $p(x|\text{winter})$.

$\{\vec{x}\}$ is a set of data points. Individual points will be denoted $\vec{x}_i$. Our example could be a set of measurements of temperature and wind speed at various times.

$\{\vec{y}\}$ is a set of simulated events from a Monte Carlo based on our model. Individual points will be denoted $\vec{y}_i$.

$\vec{\alpha}$ is a set of model parameters. Again, individual points will be denoted α_i. Our example could be the season and the time of day, so that we are asking our model to only look at summer afternoons.

2.2 Markov Chain Monte Carlo

The method of the Markov Chain Monte Carlo is a combination of two parts: a "Markov Chain", a random walk with the next step only dependent on the current step; and a "Monte Carlo", a simulation method using repeated random sampling. It requires a known relation between the data distribution and a set of model parameters, giving $p(\vec{x}|\vec{\alpha})$ and a measured set of data points $\{\vec{x}\}$; it returns information about the values of $\vec{\alpha}$, i.e. about the probability distribution $p(\vec{\alpha}|\{\vec{x}\})$ (called the posterior distribution). The method randomly

walks through the space of $\vec{\alpha}$ in a particular way (which defines the algorithm but is always a Markov Chain) and returns a random sample (the Monte Carlo aspect) from $p(\vec{\alpha}|\{\vec{x}\})$, usually called a "chain".

This random sample is akin to the output of a "normal" fitter, most of which assume that this distribution is Gaussian to report some value $\alpha_i \pm \sigma_{\alpha_i}$ for each model parameter. To convert from $p(\vec{\alpha}|\{\vec{x}\})$ to $\alpha_i \pm \sigma_{\alpha_i}$, we need merely assume that $p(\vec{\alpha}|\{\vec{x}\})$ is Gaussian near its maximum and, for each parameter individually, either fit with a Gaussian or numerically compute the mean and standard deviation of the random sample. But with the MCMC methods, we have the flexibility of looking directly at the probability distribution if we wish.

The method relies on having a way to define the probability that a data set was drawn from a distribution, since we will be varying our distribution as we vary $\vec{\alpha}$. This is most naturally done with the Likelihood function, described in the next section. In the context of the overall analysis, the question arises of how to describe $p(\vec{x}|\vec{\alpha})$ at all, given the complexity of the experiment forces us to rely on Monte Carlo simulations. We will delay answering this question until Chapter 3; for the rest of the chapter we will assume we have this available to us.

There are many MCMC algorithms (see [5, 7] for an overview), differing mainly in how the random walk through parameter space is implemented. The simplest, and first to be discussed, is the Metropolis algorithm [8]. This is the one we will use. Many factors went in to this decision. The most proiminent are that the more advanced algorithms often require the analytic derivative of $p(\vec{x}|\vec{\alpha})$, which we do not have available, or they require more evaluations of $p(\vec{x}|\vec{\alpha})$, which we will find is very computationally expensive to evaluate. In addition, the Metropolis algorithm was used is the previous SNO analysis presented in [9], so the SNO collaboration had experience with the algorithm and had already approved its use.

2.3 Extended Log Likelihood

In general terms, we have an experiment that produces a set of data points $\{\vec{x}_i\}$, with each point consisting of a number of values. Assuming that these data points represent statistically independent events (interactions in our detector), we can say that these are random draws from some probability distrubtion $p(\vec{x}|\vec{\alpha})$, where $\vec{\alpha}$ is a set of parameters describing this distrubtion's shape. Since it is a probability distribution, we have $\int_{\text{all } \vec{x}} p(\vec{x}|\vec{\alpha}) = 1$ for any $\vec{\alpha}$. We don't know this distrubtion a priori (else we wouldn't need to do the experiment), but we assume that we have a good enough understanding of the system that our ignorance can be characterized by the set of parameters $\vec{\alpha}$. Our goal is now to find the $\vec{\alpha}$ that best matches our data.

As per usual, the Likelihood is defined as the product of the probabilites of drawing each of the points.

$$\mathcal{L}(\vec{\alpha}) = \prod_i^n p(\vec{x}_i|\vec{\alpha}) \tag{2.1}$$

This gives a measure of the "probability" that the data set was drawn from $p(\vec{x}|\vec{\alpha})$, in the sense that the better that $p(\vec{x}|\vec{\alpha})$ agrees with the data, the larger $\mathcal{L}$.

Unfortunately, this is too simple. First, this assumes that we are drawing from a normalized probability density function (PDF), when in reality the number of events in our data set is very important. Second, many of the elements of $\vec{\alpha}$ are constrained by external measurements. The former is dealt with by using the Extended Likelihood, which penalizes differences between the number of events expected and the number of data events, and the latter by adding terms to the likelihood.

The Extended Likelihood is a standard prescription. We now have a "PDF" $N \cdot p(\vec{x}|\vec{\alpha})$, which will have N events in it - integrating across the distribution gives $N \cdot 1$. To incorporate this, we multipy the Likelihood in equation 2.1 by a Poisson distribution for the number of data events, with mean equal to the number of events N in the "PDF" (which is now no longer normalized to one). We call the number of data events n and we will drop the quotes around PDF from now on, with the knowledge that we are now talking about $Np(\vec{x}|\vec{\alpha})$ instead of $p(\vec{x}|\vec{\alpha})$. This gives

$$\text{EL}(\vec{\alpha}) = \frac{N^n e^{-N}}{n!}\mathcal{L}(\vec{\alpha}) = \frac{N^n e^{-N}}{n!}\prod_i^n p(\vec{x}_i|\vec{\alpha}) \tag{2.2}$$

We will later take the logarithm of this for computational reasons. Quickly evaluating that logarithm gives

$$\log(\text{EL}) = n\log(N) - N - \log(n!) + \sum_i^n \log(p(\vec{x}_i|\vec{\alpha})) = \sum_i^n \log(Np(\vec{x}_i|\vec{\alpha})) - N - \log(n!) \tag{2.3}$$

Note that n is a constant (since the set of data points is fixed), while N is not (since this is a parameter of our PDF, which we will later take to be part of $\vec{\alpha}$).

The constraint terms are external measurements for our parameters, giving independent information about the values the parameter can take. We simply multiply our $\mathcal{L}$ by the (normalized) pdf that corresponds to the measurement. This is stating that we have a prior distribution for that parameter, so that our external information tells us what we think the probability distribution for that parameter should be. In our analysis, these take one of three forms: a Gaussian, a "two sided Gaussian", and a covariance matrix.

The simplest and most common case is the Gaussian. If a parameter α_j has a value $\bar{\alpha}_j \pm \sigma$, then we multiply $\mathcal{L}$ by a gaussian distribution

$$p(\alpha_j) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\alpha_j - \bar{\alpha}_j)^2}{2\sigma^2}\right) \tag{2.4}$$

The "two-sided Gaussian" is the case of asymmetric uncertainties, i.e. a measured value of $\bar{\alpha}^{+\sigma_+}_{-\sigma_-}$. This corresponds to a Gaussian with one value of σ above the mean and a different one below, so we multiply $\mathcal{L}$ by

$$p(\alpha_j) = \frac{\sqrt{2}}{\sqrt{\pi}(\sigma_+ + \sigma_-)} \begin{cases} \exp\left(-\frac{(\alpha_j - \bar{\alpha}_j)^2}{2\sigma_-^2}\right) & \text{if } \alpha_j \leq \bar{\alpha}_j \\ \exp\left(-\frac{(\alpha_j - \bar{\alpha}_j)^2}{2\sigma_+^2}\right) & \text{if } \alpha_j > \bar{\alpha}_j \end{cases} \tag{2.5}$$

A covariance matrix is somewhat more complicated, as it links the values of several parameters. It corresponds to a multi-dimensional Gaussian. Given an $m \times m$ covariance matrix $\mathbf{\Sigma}$ linking m parameters (label them $\vec{\beta}$)

$$p(\vec{\beta}) = \frac{1}{(2\pi)^{m/2}|\mathbf{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\vec{\beta} - \bar{\vec{\beta}})^T \mathbf{\Sigma}^{-1} (\vec{\beta} - \bar{\vec{\beta}})\right) \tag{2.6}$$

For ease of computation, and computational accuracy on a finite-precision machine, we take the logarithm of the product of the extended likelihood and the constraints. Since $\log(x)$ is a monotonically increasing function of x, maximizing one is equivalent to maximizing the other. We drop all terms that are constants and hence do not contribute to the maximization process to end up with the Extended Log Likelihood:

$$\text{ELL}(\vec{\alpha}) = \log(\text{EL}) + \sum_{constraints} \log(p(\vec{\alpha})) = \sum_{i=0}^{n} \log(Np(\vec{x}_i|\vec{y})) - N + \sum_{constraints} \log(p(\vec{\alpha})) \tag{2.7}$$

When the constraint terms are expanded, in general the normalization coefficients, being constant, are dropped.

In sum, we now have a way of measuring the probability that our data set $\{\vec{x}\}$ was drawn from the distribution characterized by $\vec{\alpha}$, taking in to account the known constraints on $\vec{\alpha}$. How this ELL changes as we change $\vec{\alpha}$ will tell us how we want to direct our random walk through parameter space, as codified in the Metropolis algorithm.

2.4 Metropolis Algorithm

The Metropolis algorithm consists of a loop which walks across the parameter space of $\vec{\alpha}$. At the end of the i^{th} step, it will have the parameter values $\vec{\alpha}_i$. The $(i+1)^{th}$ step starts by varying each of the parameters by adding a random draw from a Gaussian with $\mu = 0$ and $\sigma = \sigma_{\alpha_j}$, i.e. each parameter is varied around its current position with a width specific to that parameter (determined before the process starts, see Section 2.5). This creates a "proposed" step $\vec{\alpha}^{prop}_{i+1}$. The Extended Likelihood (including all constraints, so $\exp(\text{EL}(\vec{\alpha}))$ is then compared between the previous and the current step by taking the ratio $p(\text{step}) = \dfrac{EL(\vec{\alpha}^{prop}_{i+1})}{EL(\vec{\alpha}_i)}$. A random number is drawn from the uniform distribution between 0 and 1, call it

rand$(0,1)$. If rand$(0,1) < p(\text{step})$ then we keep the proposed value and set $\vec{\alpha}_{i+1} = \vec{\alpha}_{i+1}^{prop}$, if not we don't take the step and keep our old one, so $\vec{\alpha}_{i+1} = \vec{\alpha}_i$. This latter point is important - the repetition in our data set of "better points" is a large part of what gives rise to the useful behavior of this algorithm. In more algorithmic form this is:

$$\vec{\alpha}_{i+1}^{prop} = \vec{\alpha}_i + \vec{g}(\vec{\sigma}_{\vec{\alpha}})$$
$$p(\text{step}) = \frac{EL(\vec{\alpha}_{i+1}^{prop})}{EL(\vec{\alpha}_i)}$$
$$\textbf{if } \text{rand}(0,1) < p(\text{step})$$
$$\quad \textbf{then } \vec{\alpha}_{i+1} = \vec{\alpha}_{i+1}^{prop}$$
$$\quad \textbf{else } \vec{\alpha}_{i+1} = \vec{\alpha}_i$$

The output of the algorithm is the set $\{\vec{\alpha}_i\}$. The first few steps depend on the initial starting position $\vec{\alpha}_0$, so we discard those as the "burn-in" period (discussed in Section 2.5). The rest of the points are a random draw from the posterior distribution of the parameters $p(\vec{\alpha}|\{\vec{x}_i\})$ - the probability distribution for our parameters given the data points. This is exactly what any fitting algorithm (or experiment, for that matter) sets out to find. Since we have a collection of random draws from this distribution, we bin them to create an approximate PDF, or we assume that they will be Gaussian (or two-sided Gaussian) near the maximum and do an unbinned fit, to find out what values of parameters best correspond to our data. An example fit, from the final analysis, is shown in Figure 2.1. Since this is directly the probability distribution, its mean is the parameter value and its standard deviation (σ) is the parameter uncertainty. With many parameters, if we do this for a single parameter (so we only fit or histogram one element of the vector), we get the distribution integrating across all other parameters. If we wish to find the covaraince between two parameters, we merely histogram them against each other, fit them as a 2D fit, or directly compute the covariance of the two sets of values.

2.5 Step Size Finding

In the MCMC process, at each step each parameter is varied by adding a random draw from a Gaussian, with a different width for each parameter. These widths don't affect the theoretical behavior of the chain, in the sense that any choice of widths will eventually produce a random sampling of $p(\vec{\alpha}|\{\vec{x}\})$. They do, however, have a very large impact on the practical behavior of the MCMC.

This can be illustrated by thinking about a case with one parameter, for example one with the Likelihood versus parameter value shown in Figure 2.2. Suppose we start our chain at the point marked with an arrow. The dimension marked σ is a reasonably good width for our Gaussian. Since the Likelihood is steep there, we are much more likely to take a step "uphill" and will reach the maximum in a few steps. More importantly, the region containing most of the probability is only approximately ten σ's wide. Once a parameter value near the maximum is reached, the algorithm should have a good probability of stepping back "downhill" in one direction or the other, and thus exploring the space around the maximum, as a step of size σ will not change the likelihood value too dramatically. If the width were a tenth of σ, however, we would have two problems: first, it would take many steps to get to the maximum (this corresponds to a longer burn-in period) and, once at the maximum, it would take a very large number of steps to explore the majority of the probability. If the width were ten times σ, the opposite problem would occur: it would be very likely that the step would take us away from the maximum completely, and with a very low likelihood away from the maximum, that step would be rejected by the algorithm. We seek to balance between these two cases.

Looking at Figure 2.2, we can immediately see that the starting location of our chain plays a role. Thankfully, it can be proven that this role is transient, that is that regardless of the starting point the chain will settle in to a proper sampling $p(\vec{\alpha}|\{\vec{x}\})$ [7]. We need to discard this initial transient, called the burn-in period, as it is not part of the sample we want. The characteristic behavior of this burn-in period is a rapidly changing log-likelihood, as the chain wanders through low-probability regions where it is very probable for it to take steps "uphill". Once the chain reaches a high probability region, the stable sampling behavior will set in; this is called a stationary chain or stationarity. This gives rise to graphs of log-likelihood versus step such as Figure 2.3. The end of the burn-in period is fairly obvious on the graph, occuring around step 200,

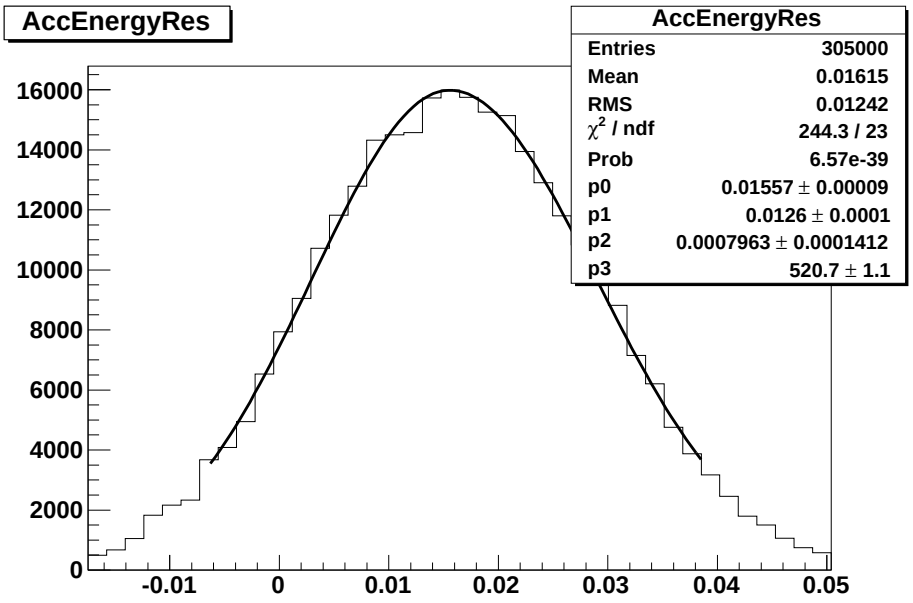

Figure 2.1: An example posterior distribution, with an asymmetric Gaussian fit. This is the energy resolution systematic from the final fit.

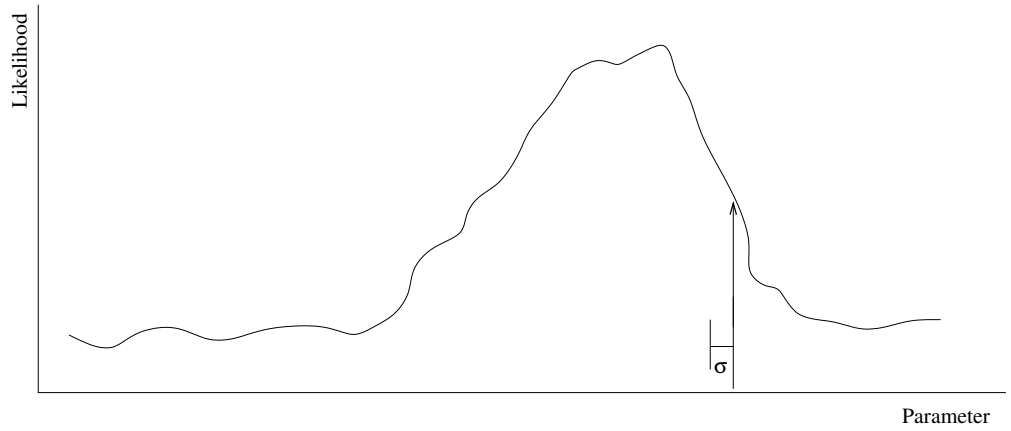

Figure 2.2: An example of a possible Likelihood versus parameter value graph, in the case of a system with a single parameter. See text for details.

when the changes become the size of the long-term fluctuations. We customarily add 50% or 100% to the size of the burn-in period to be safe, as occasionally a parameter will not have settled in to stationarity at that point, if it has a very small impact on the likelihood. Ideally, we should check each parameter individually, but this is not practical when we are dealing with hundreds of runs each containing dozens of parameters. Instead, we choose a few sample runs and check to make sure the safety margin is sufficient. We can, of course, force the burn-in period to be shorter by moving the initial point closer to the maximum likelihood point (as with any fitter).

Before discussing how to optimize these step sizes, we need a goodness measurement for our step sizes. We seek to avoid two problems: too large a step, which results in few steps taken, and too small a step, which results in the space not being sampled properly. The former is simple to measure, we just measure how often the algorithm takes a step. Denote that "% steps". The rule of thumb is that it should be roughly 23.4% for a system with multiple parameters [5, 10], though the precision of this number belies the range of acceptable values - any % step between 20% and 30% is quite good, and something between 15% and 35% is acceptable. The latter problem is less obvious, but is well measured by the autocorrelation. This measures how many steps the algorithm takes before values are no longer correlated with earlier values, in colloquial terms how long before the chain "forgets" where it was before. The smaller this is, the better we are sampling the space, so it measures how independent our samples of $p(\vec{\alpha}|\{\vec{x}\})$ are. The autocorrelation is not an indepedent measure from % steps, as when very few steps are taken, the chain "remembers" its location for a long time. In fact, [5, 7] point out that the autocorrelation is the correct measure of what we desire in a chain, while the % step is just a useful rule of thumb. We note, however, that letting % step get too small, even if it gives a better autocorrelation, can unacceptably increase the burn-in period. The autocorrelation is defined as

$$\rho(h) = \frac{\sum_t [(x_t - \bar{x})(x_{t+h} - \bar{x})]}{\sqrt{\sum_t (x_t - \bar{x})^2 \cdot \sum_t (x_{t+h} - \bar{x})^2}} \tag{2.8}$$

where x_i is an element in a sequence and h is called the "delay". $\rho(h)$ is equal to the traditional definition of correlation between two sequences for the sequence x_i and x_{i+h}. Note that this is only a useful measure once the burn-in period has been removed, as we wish to measure the stationary sampling behavior of the chain. The autocorrelation should decay exponentially, with a fluctuating background level due to the finite number of points available. A sample autocorrelation is shown in Figure 2.4, which includes an exponential fit of the form

$$\rho(h) = A \exp(-\frac{h}{\tau})$$

The exponential's decay-time parameter τ gives us a single, real number measure of the autocorrelation. The smaller this τ, the faster the chain forgets its previous position, giving "more independent" random draws from $p(\vec{\alpha}|\{\vec{x}\})$.

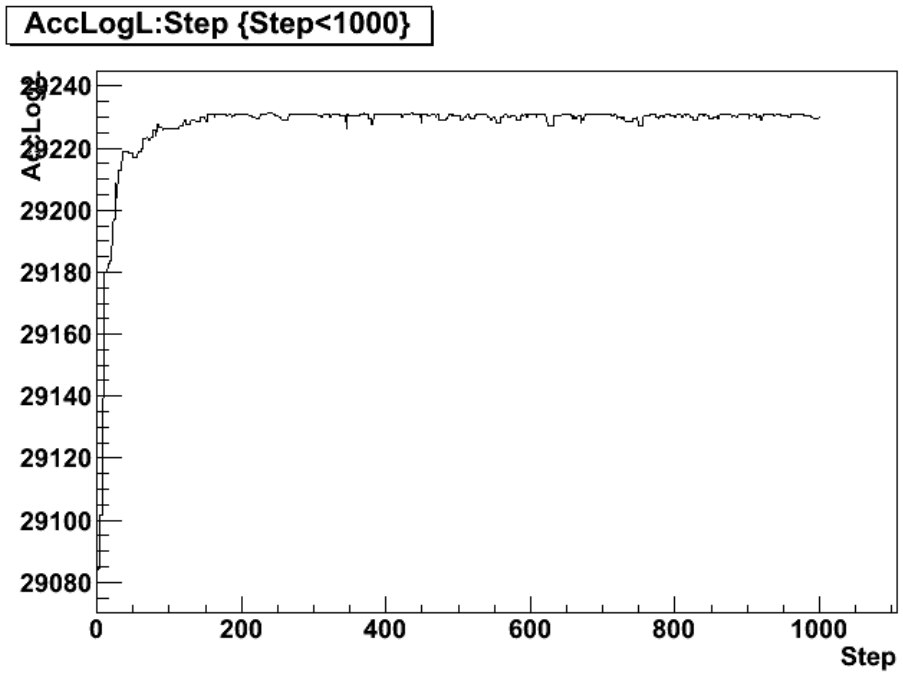

Figure 2.3: Sample log likelihood v. step plot showing burn-in

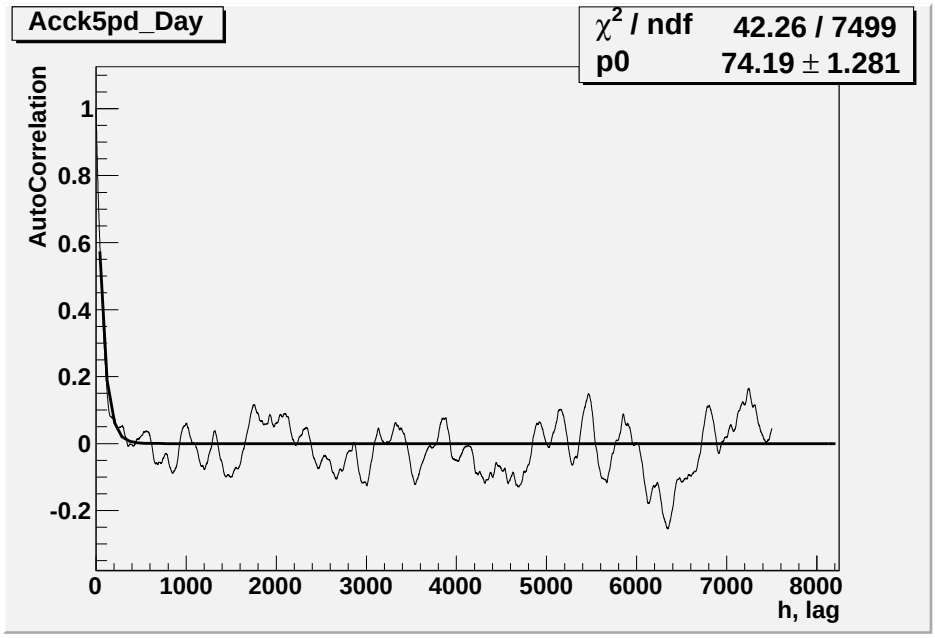

Figure 2.4: Sample autocorrelation with exponential fit

We now seek to minimize τ for all parameters in our chain simultaneously while keeping the % step in an acceptable range. A way of automating this process is still an active area of research, called Adaptive MCMC (see, for example, [10]). Having found no accepted method for finding the σ_{α_j}'s, through trial and error we settled on a methodology that, while a bit slow and labor intensive, produces good results. This method attempts to make best use of our parallel computing resources by running many chains with different step sizes in parallel at each step. We will refer to each instance of MCMC running in parallel as a "run". Each run has a fixed step size for each parameter.

We start by running a large sweep over possible step sizes. Since we are assuming to start with very little knowledge of the correct step size, we will look over several orders of magnitude and sample logarithmically, so that we get approximately the same number of samples in each decade. First we select a range to sample over for each parameter. In the case of a parameter with an external constraint, this is straightfoward - the step size will be less than the width of the constraint, but could be much smaller if the data more sharply constrains it. We will denote the constraint width as δ. If we know nothing else about the system, we take a range $[10^{-4}\delta, \delta]$. We restrict this range if we have more information, such as a similar system of parameters to compare against. In the case of a parameter with no constraint, we need to guess. If we have an idea of the ranges of values that we expect the parameter to take, we use these in place of δ, possibly extending the range to $[10^{-6}\delta, 10\,\delta]$. Otherwise, we simply run as broad of a sweep as we can, starting with a range of $[10^{-8}, 10^{2}]$ when we run 100 runs in parallel (to keep about 10 runs per decade). We may find we are still outside the range we want, in which case we repeat with a different range. We then randomly sample within this range for each parameter, to generate a number of runs with different step sizes. We randomly choose instead of picking a progression as the large number of parameters (over 50 in the main analysis) would require a unacceptably large number of runs. We then run these in parallel, compute the autocorrelation for each parameter and % step for each run, and generate a plot of τ versus step size and % step versus step size for each parameter. The burn-in period can be a problem here, as it will vary for various step sizes. We attempt to control for this by starting the chains as close to the maximum as we can. Since we do this step size finding process on simulated data to avoid bias and blindness problems, we know the actual maximum and can start arbitrarily close to it. In addition, those runs with too small a step size to find the maximum will return very bad autocorrelations, which we need to keep in mind when looking at results.

A sample τ plot is shown in Figure 2.5. The key characteristics to note are that τ tends to decrease as the step size increases. Figure 2.6 shows a sample % step versus step size plot. Here we see that as step size increases, % step decreases. This is the fundamental trade-off in the step size finding process: increasing the step size of a parameter decreases that parameter's autocorrelation, but decreases the % step, which increases the autocorrelation of all parameters. We have found that the "corner" in the τ graph, where τ stops decreasing as quickly (about 0.4 in Figure 2.5, though we note that the τ v. step size behavior is actually power-law like) is approximately the best value. We then find this corner for each parameter and record it; we will denote this value ς. If the corner is not present, we have not looked over a sufficiently large range of step sizes so we repeat this step with altered ranges.

With the ς's in hand, we know approximately the correct values. Unfortunately, the τ's are correlated, both through the % step change and through the MCMC process itself. Thus we must search again, though on a more restricted range. We again create runs with logarithmically randomly selected step sizes for each parameter, this time from the range $\frac{1}{3}\varsigma, 3\varsigma$ (this is, of course, arbitrary). We again run these in parallel and compute the autocorrelations. Instead of graphing the results, we search though them for those with desirable characteristics. We search for two types of runs: those with the smallest τ's (in particular ones with all τ's less than a particular value, which we lower until only a few runs qualify) and those with the most uniform τ values. Ideally, we'll find a run with all τ's similar and small, in the case of the final experimental run this was a τ of approximately 200. If we do, and this has a reasonable % step, we have found our step sizes and we stop. If not, we look at the best runs and see what needs to change.

If the runs are close to what we need, we adjust by hand and test. In the case of one or two parameters with too large τ's, we increase the step sizes of those problem parameters and decrease the step sizes of a few parameters with much lower τ's to compensate (unless the % step is too large, in which case we only increase). If all of the τ's are similar but too large, we look at the % step. If it is too large, we increase all step sizes, if too small we reduce all step sizes. We must be careful here - if all of the τ's are much too large, this can be the result of the autocorrelation not fitting well to an exponential, for example Figure 2.7. This usually arises from a run that hasn't actually passed through the burn-in period, though it can also result

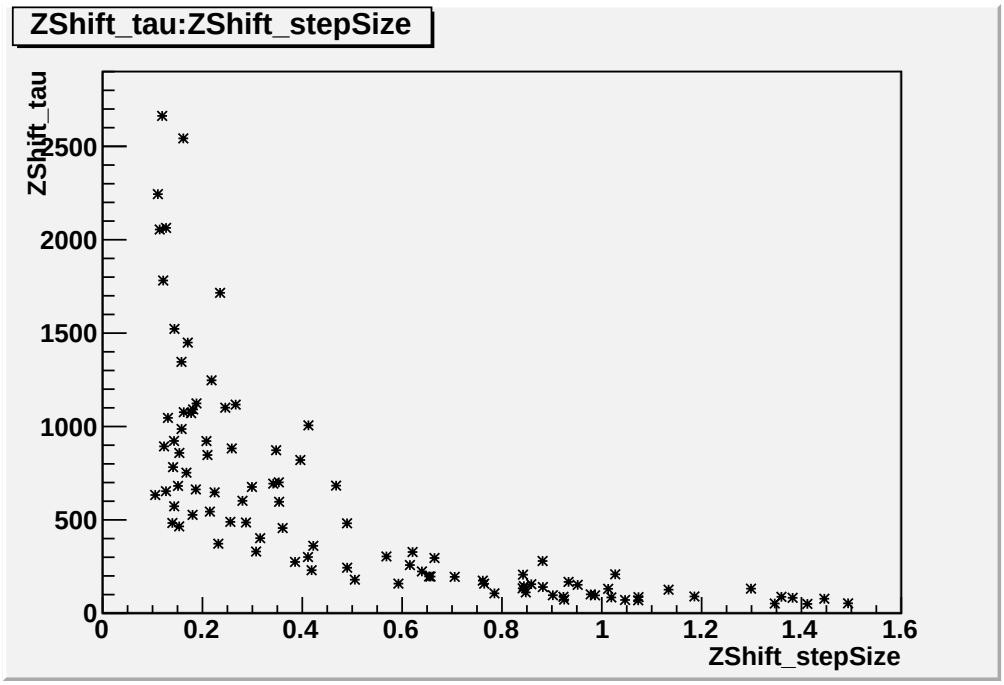

Figure 2.5: Sample autocorrelation τ v. step size

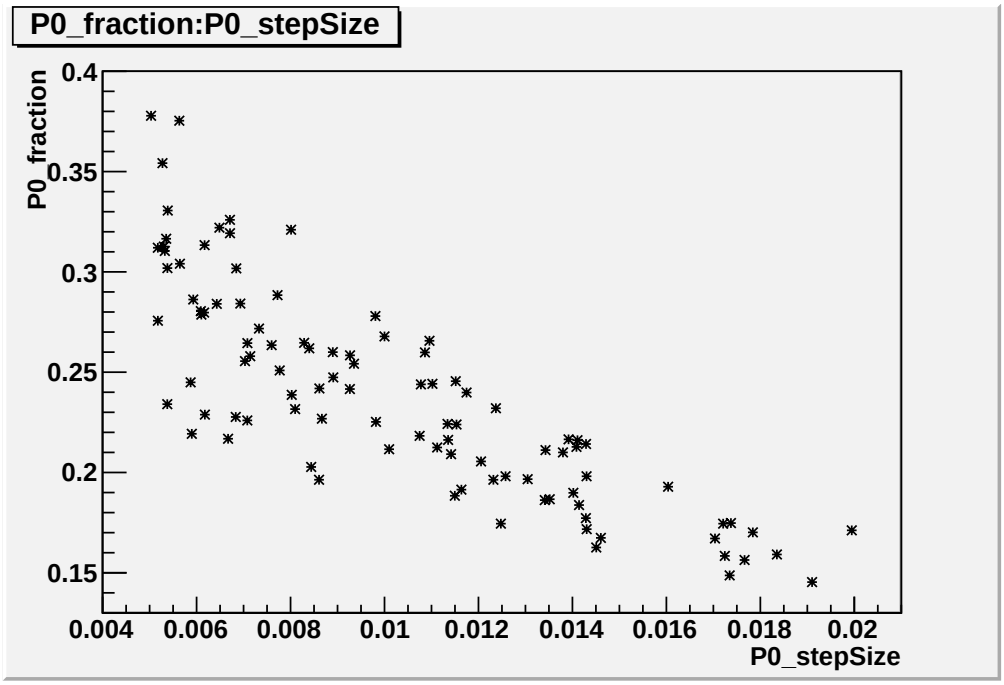

Figure 2.6: Sample % step v. step size

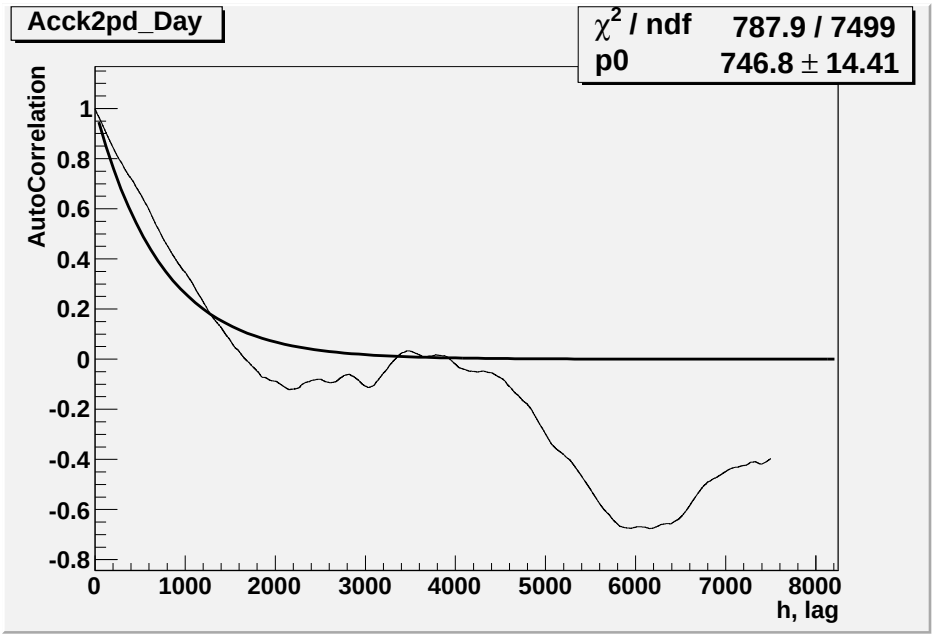

Figure 2.7: Sample autocorrelation plot where exponential fit failed

from a run with a τ so large that we simply did not run long enough to properly measure it. We reject those runs, as they tell us nothing. Either way, this gives a new set of values, $\tilde{\zeta}$. We now again randomly sample, this time of the linear range $[\frac{1}{2}\tilde{\zeta}, \frac{3}{2}\tilde{\zeta}]$, but include an additional run with all parameter step sizes at $\tilde{\zeta}$. We then repeat the search described in this paragraph, adjusting as necessary and iterating. This usually only takes one or two iterations, but occasionally can become problematic.

In an attempt to automate and streamline this process, we implemented the Adaptive MCMC described in [10]. This is a variant on the Metropolis algorithm which adjusts the step sizes at each step, in an ever decreasing way. This adjusts the MCMC algorithm to

$$\vec{\alpha}_{i+1}^{prop} = \vec{\alpha}_i + \vec{g}(\vec{\sigma}_{\vec{\alpha},i})$$
$$p(\text{step}) = \frac{EL(\alpha, \vec{i+1}^{prop})}{EL(\vec{\alpha}_i)}$$
if rand$(0,1) < p$(step)
then $\vec{\alpha}_{i+1} = \vec{\alpha}_{i+1}^{prop}$
else $\vec{\alpha}_{i+1} = \vec{\alpha}_i$
$$\vec{\sigma}_{\vec{\alpha},i+1} = \vec{\sigma}_{\vec{\alpha},i} + \vec{\sigma}_{\vec{\alpha},0}(p(\text{step}) - 0.234)i^{-\beta}$$

where β is a real number controlling how quickly the adjustments reduce in size, usually set somewhere between 0.5 and 1. This is a very simple change: we now have a variable step size, which we adjust at every step, decreasing if the probability to take a step was smaller than 23.4% and increasing if it was larger. This serves to force the algorithm to take % step = 23.4%. This algorithm, as presented in the paper, has an unacceptable drawback: it changes all the step sizes together. This means that their relative proportions never change. This was the case for the example in the paper, but is not the case for us. We do not know beforehand what the ratios should be, since some parameters affect the likelihood more than others. To correct this, we adjust this algorithm to only alter one parameter at a time. The algorithm behaves

identically, save that the last line now reads

$$(\vec{\sigma}_{\vec{\alpha},i+1})_j = (\vec{\sigma}_{\vec{\alpha},i})_j + (\vec{\sigma}_{\vec{\alpha},0})_j (p(\text{step}) - 0.234) i^{-\beta}$$

and j is incremented at each step, modulo the number of parameters, so it walks through the parameter list from top to bottom repeatedly.

When we initially did this, we found that the first parameter was being altered too much. The size of the changes in $\vec{\sigma}_{\vec{\alpha},i}$ is falling like $i^{-\beta}$, where we used $\beta = 0.66$, which is 1 for the first step, and approximately 0.08 by the end of the first pass through our roughly 50 parameter long list. To correct for this, we instead keep the size fixed for the first pass through the list: we have the first m steps (where m is the number of parameters) replace $i^{-\beta}$ with $m^{-\beta}$.

This altered algorithm does amazingly well at causing the chain to settle in to a set of step sizes where % step is 23.4%. Unfortunately, there are many such sets of step sizes - this is a single (albeit complicated) constraint in a high-dimensional space of possible step sizes. This adaptive algorithm did not produce useful results, as it always found a region where some of the autocorrelations were very small and some were very large. We wound up discarding it after many trials and reverting to the original Metropolis algorithm with the step size finding procedure outlined above.

2.6 Benefits and drawbacks

The most obvious strengths to this method are computational. In particular, it is an "embarrasingly parallel" algorithm - to run it across multiple computers, we run an instance of the algorithm on each machine. They do not need to communicate. We then simply take each machine's output for $\{\vec{\alpha}\}$ (which will be different, as this is an inherently random process) and concatenate them in to a larger set. This is in stark contrast to a deterministic Minuit-style minimizer, which is extremely difficult to parallelize. The only catch to this is the burn-in period. Each computer will individually go through this process, so the first few events of each computer's output chain must be removed before concatenation. Even with that being true, this easily allows much longer chains to be run. For instance, in this analysis we had access to a cluster of approximately 1500 nodes, called Tier2, of which we could reasonably expect to use 100 or so at a given time. In the final analysis, we could run approximately 7000 algorithm steps in a 24 hour period. With a burn-in period of 2000 steps, we were able to use a chain of 305000 events in a 24-hour run (not all 100 jobs ran due to computer cluster problems), more than sufficient for good results.

In addition, the MCMC algorithm deals reasonably well with large numbers of parameters. The additional cost of an additional parameter is very small, one random draw from a Gaussian per step. The real additional costs come from two places: adding an additional parameter usually implies adding complexity to $p(\vec{x}|\vec{\alpha})$, which can be expensive, and more parameters generally require a longer burn-in period and chain length. The longer burn-in period is from adding a dimension to our parameter space, which slows the random walk process. The chain length increases because of the step size problem: as more parameters are added, the probability of taking a step goes down (assuming the additional parameters have a reasonably strong influence on the ELL), requiring smaller step sizes.

Another, less obvious benefit is that the MCMC deals well with systems where the likelihood is very complicated. If the likelihood is not smooth near the maximum, as is the case for our SigEx, a typical Minuit-type minimizer can easily get caught in a local maximum. In addition, since these fitters assume Gaussian-like behavior near the minimum (corresponding to our maximum), they can get incorrect uncertainties. The MCMC, however, simply maps out the likelihood space and is immune to these problems. They will, instead, show up as non-smoothness is $p(\vec{\alpha}|\{\vec{x}\})$, which is appropriate. How to interpret this is a separate issue. We found all of our parameters to have posterior distributions sufficiently close to a two-sided Gaussian near the maximum that we were comfortable fitting that function to them and report a value $\alpha_j \, ^{+\sigma_+}_{-\sigma_-}$ for each.

From a statistical standpoint, a major benefit is that we get a random sample of $p(\vec{\alpha}|\{\vec{x}\})$. We can now compute any quantity we like, such as the correlation between any pair of variables, by simply computing that quantity on those two elements of $\{\vec{\alpha}\}$. In addition, when we histogram a single parameter's value, we integrate across all other parameters. This is in a rather literal sense: when we do this, we get

$$p(\alpha_j|\{\vec{x}\}) = \int \cdots \int p(\vec{\alpha}|\{\vec{x}\}) \mathrm{d}\alpha_1 \ldots \mathrm{d}\alpha_{j-1} \mathrm{d}\alpha_{j+1} \ldots \mathrm{d}\alpha_m \tag{2.9}$$

This is the correct way to treat nuissance parameters such as systematics, as it correctly folds their contributions to the likelihood in to the width of the distribution of the parameter, including any correlations. This then avoids complications associated with correlated parameters, though if we want to give the point of maximum likelihood rather than the most probable value for each parameter, we need to fit a multi-dimensional function to the joint distributions of those parameters we are concerned about.

The MCMC is not without drawbacks. The most glaring is the issue of finding step sizes. This adds greatly to the complexity of running an MCMC, though we hope that researchers will one day perfect an adaptive MCMC that does not have this problem. Another problem related to its complexity is that, as a random process, it tends to be slower than a deterministic fitter on simpler systems. In addition, it is not always clear when the burn-in period has passed, as the algorithm does occasionally fall in to local minima. It will wander out of them eventually, but that can be much longer than is reasonable to wait. This is unusual, thankfully, and can be tested by either starting many parallel chains with different initial conditions or by running one very long chain. The last drawback we will mention is that since this algorithm is less well understood than some other fitters, interpreting the results is more difficult. It is impractical to report the full $p(\vec{\alpha}|\{\vec{x}\})$, except possibly for a few parameters. In addition, the expectation is that a single number plus an uncertainty will be reported. This is fine when $p(\alpha_j|\{\vec{x}\})$ is well approximated by a Gaussian or two-sided Gaussian near the maximum, but that is not always the case.

Chapter 3

Signal Extraction

Thus far, we have looked at all the pieces that make up the analysis presented in this thesis, the three phase Day/Night signal extraction. Now we put all of these pieces together to actually perform the analysis. We look at the underlying methodology, the computer code used to implement it, and the tests done to ensure it was working as expected.

We begin by summarizing the method: The analysis starts with a set of Monte Carlos of the SNO detector, simulating the principle fluxes (CC, ES, $ES_{\mu\tau}$ and NC) and backgrounds. These are summed together, varying the relative amounts of each and applying a set of distortions representing possible inaccuracies and uncertainties in the simulation (the systematics). The parameters describing the relative amounts and distortions are collectively refered to as $\vec{\alpha}$. This summed, distorted MC is compared to the data, giving a log likelihood. We then vary this $\vec{\alpha}$ via a Markov Chain Monte Carlo method (the Metropolis algorithm) to extract the probability distributions for values of our parameters given our data - this is the posterior distribution $p(\vec{\alpha}|\{\vec{x}\})$ described in Chapter 2.

To understand how this process is carried out, we first explain how $N\,p(\vec{x}|\vec{\alpha})$ is calculated. We then explain how the results of the two external analyses LETA and PSA are included. We then explain the implementation itself and how our computer code is structured. Finally, we show the results of our tests to confirm the code is working as expected.

3.1 PDF

Throughout our discussion of the log likelihood, we left $N\,p(\vec{x}|\vec{\alpha})$ abstract. To actually build a functional MCMC, we must specify how it is evaluated. There are many ways to do this in principle, though the complexity of any modern experiment leaves us with basically one choice: Monte Carlo. A detailed MC of our experiment gives us a set of MC simulated events $\{\vec{y}\}$. Each of these events will consist of a set of values containing those in the data, $\{\vec{x}\}$, plus additional information from the MC not available in the data, such as the "true" values that went in to the generation of the event (as opposed to the measured values). As described in Chapter **??**, the analysis only uses three data variables, E, ρ^3 and $\cos(\theta_{sun})$, while the Monte Carlo uses E, E_{true}, x, y, z, x_{true}, y_{true}, z_{true}, $\cos(\theta_{sun})$, and E_ν.[1]

To turn this set of points in to a PDF, we histogram $\{\vec{y}\}$ and interpret the value in each bin as $p(\vec{x}|\vec{\alpha})$ for any point in that bin. This potentially introduces a systematic uncertainty - we are effectively integrating the "true" $p(\vec{x}|\vec{\alpha})$ in each bin and only looking at the average. If the underlying $p(\vec{x}|\vec{\alpha})$ varies quickly, we may wash out features that we are interested in and lose discriminating power. This can be tested by looking at the one dimensional projections of the MC in each of the principle variables (E, ρ^3 and $\cos(\theta_{sun})$), reducing the bin size significantly and looking for any regions of rapid change. The one dimension projections are used only to increase statistics and allow finer bins. This was done for the NCD phase analysis presented in [9], though not discussed there. Additionally, we repeated this simple test and saw no such regions that would be averaged out by the binning process.

[1] ρ^3 is then computed from x, y and z

Flux	MC events	Data events	`TimesExpected`
CC	6198.589	5628150	907.9728
NC	266.051	244751	919.9367
ES	532.611	970822	1822.762
$ES_{\mu\tau}$	82.7599	1508525	18227.725
ex	20.60133	119417	5796.528
d2opd	8.2987	244751	29492.692
ncdpd	5.7265	70175	12254.431
k2pd	9.36112	112710	12040.226
k5pd	11.557	118464	10250.411
atmos	24.66224	244751	9924.119

Table 3.1: The total number of MC events in the analysis window, expected number of data events and `TimesExpected` for each flux and background. `TimesExpected` can be computed by (expected data events)/(MC events). Note that the expected data events are for the PMTs, the amounts in Table **??** are for the NCDs.

We define a set of bins in each of the data values (a 3-D histogram here) and fill with the MC events. As explained in Section **??**, we restrict to a particular range in each value, so we only create histogram bins over that range. Many of our MC events are outside these ranges and thus do not appear in the histogram at this stage, but we retain them as the systematics may alter their values enough to move them inside our cuts. In this analysis, we used the following bins: 10 equal bins in ρ^3 from 0 to 0.77025, 25 equal bins in $\cos(\theta_{sun})$ from -1 to 1 and a set of bins in E consisting of 0.5 MeV spacing between 6 and 13 Mev, a larger bin from 13 to 20 MeV (13 total bins). Since the Pdf is three dimensional, this it has $10 \cdot 25 \cdot 13 = 3250$ bins. These are the same bins used in the NCD phase analysis [9].

Several different physical processes, our primary fluxes, contribute to our signal. Each of these has a separate set of MC events. We also expect to have several backgrounds producing events, again each with a separate set of MC events. These backgrounds are treated identically to the fluxes. We take each set of MC events separately, apply the relevant subset of the systematics to each event in that set, and fill the resulting "new" event values in to the histogram. Each event is filled with a weight determined both by the weight systematics and a conversion factor from the number of MC events to the expected number of data events. In this manner, we are able to vary the relative proportions of each flux in the resulting histogram, as well as the total number of events N. This makes N dependent on both the intensities of each flux as well as the systematics.

This leads us to our construction method for $N\,p(\vec{x}|\vec{\alpha})$. We have m sets of MC, one for each flux (or background), and a set of parameters $\vec{\alpha}$. This $\vec{\alpha}$ contains values for our systematic parameters and parameters defining the size of each flux. The latter took several related forms over the course of testing the code, but for sake of exposition we'll use the final form: each flux has a parameter with nominal value 1 that represents the "scale", i.e. how much it deviates from expectation (with expectation normalized to 1). This necessitates a fixed conversion factor from MC events to data events representing how many "experiments worth" of MC are in the set, called `TimesExpected`, defined as the number of events expected for this flux (in the data) divided by the number of MC events in the analysis window. The systematics can change the number of events, but this ratio assumes systematics at their nominal values. Since not all systematics are applied to each flux, each flux is assigned a `FluxNumber` and each systematic knows which `FluxNumber`'s it acts on. The process is then to go through the fluxes one at a time, taking each individual event, computing the value changes for the relevant systematics and filling the histogram with the resulting event values, with weight given by

$$W_{final} = \frac{\text{Scale}}{\text{TimesExpected}} \prod_k W_{sys,k} \tag{3.1}$$

where the survival probability from Section **??** is treated as a systematic. The number of MC events, expected number of data events and `TimesExpected` for each flux and background are shown in Table 3.1.

The resulting histogram is *almost* $N\,p(\vec{x}|\vec{\alpha})$. If the bin volumes in our histogram are all equal, then it is and we stop. However, if they are unequal, we must compensate for that. The problem arises in that

we are interpreting this as an (unnormalized) PDF. A simple example to illustrate is the case of a constant probability over some region, say $[0, 1]$. A sequence of 1000 random draws from this distribution (as our MC) will be uniformly distributed across $[0, 1]$, as we expect. Suppose our histogram has two bins. If they are $[0, 0.5)$ and $[0.5, 1]$, then there is no problem - each has (on average) 500 events. If the two bins are $[0, 0.3)$ and $[0.3, 1]$, however, this doesn't work - one has 300 events and the other 700. If we then look at these as our $Np(x)$, we see that this will incorrectly over-weight the larger bin. To correct for this, we divide each bin by its width. This results in each bin having a "size" (it is no longer simply counts) of 1000.0, again equal. This naturally raises the question of whether we should then re-scale the histogram to get back to some semblance of counts. Interestingly, this actually does that for us - we have $\int (Np(x))\,\mathrm{d}x = N$, in this case 1000. Of course, the next question is: does not dividing by the bin width in the uniform case cause a problem? Since we are using the log likelihood, the answer is no. If we have a constant scaling factor γ applied to all bins, this is equivalent to having $N\gamma p(\vec{x}|\vec{\alpha})$. Ignoring constraint terms (which are unaffected), using this in our ELL gives

$$\mathrm{ELL}(\vec{\alpha})_\gamma = \sum_i^n \log(N\gamma p(\vec{x}_i|\vec{\alpha})) = \sum_i^n (\log(Np(\vec{x}_i|\vec{\alpha})) + \log(\gamma)) = \mathrm{ELL}(\vec{\alpha}) + n\log(\gamma) \tag{3.2}$$

Since γ is fixed during the maximization process (it is a property of the binning, which doesn't change), it has no effect and we can neglect it.

3.2 Implementation

To actually run the signal extraction, the abstract method we have been describing must be made concrete in the form of a computer program. We wanted this program to be written in C++ using ROOT, the current standard in particle physics, as C++ gives reasonably fast, portable code and ROOT allows direct interaction with ROOT's powerful analysis utilities. We wanted it to be object oriented to take advantage of C++'s object-oriented nature and object-oriented code's flexibility and understandability. We wanted it to be flexible, so that it could be used for another experiment with minor or no changes. Finally, we wanted as much as possible to be controlled by a configuration file that exists outside of the code, so that we could run various tests on different configurations of data, systematics and MC without having to change the underlying code.

In the following discussion, we will attempt to distinguish between the C++ computer code (the code), the compiled executable that runs (the program), the configuration file that directs the program as to what to do (the config file or the configuration file) and the files containing the data events (the data file) and the MC events (the MC file). Often the difference between something in the code and something in the program is ambiguous, as the program is generated from the code. In those cases, it is simplest to assume that the thing in question is part of both, or rather that it is part of the abstract conception underlying both rather than these two concrete expressions of this conception. Additionally, the expression "structure" here does not mean a traditional C/C++ `struct`, rather it refers to any sort of logical organization within the code. Anything appearing in `typewriter` font refers to a component of the code or program, for example to distinguish between the `Pdf` class and the mathematical Pdf $N\,p(\vec{x}|\vec{\alpha})$.

3.2.1 Overview

A run of the program consisted of several stages. First, the config file was read and checked for errors (usually typos). Next, the information in the config file was used to set up the necessary internal structures, telling the program how many fluxes of what type, how many systematics and what their action should be, the initial state and step sizes for each parameter, likelihood contributions from any constrained parameters, etc. Then the MCMC chain ran, walking the parameters across configuration space and recording the appropriate parameter values. Finally, the results were output to files and the program performed clean up actions (closing files, clearing memory, etc).

The setup phase is the most complicated. A hierarchy of structures is created. At the highest level, the MCMC itself runs. It contains $\vec{\alpha}$ as a list of parameters, with an initial starting point, step size, current value and possible constraint for each extracted from the config file. It also extracts from the config file how

many steps the chain should take and how it should record its output, i.e. to what file, which parameters to write and how often to print status messages to the screen. To run the MCMC, it varies the current values of the parameters, evaluates ELL($\vec{\alpha}$), decides whether to keep or reject the step and records the result. Computing ELL($\vec{\alpha}$) is not trivial. There are two contributions: from the data and from the constraints. With a few exceptions noted in the class descriptions below, the constraints are dealt with by the MCMC via the mechanism of the `LogLikelihoodFormula` or parameter constraints. The former is a function defined in the config file that is explicitly added to the log likelihood. The latter adds directly to the log likelihood as well, but is a separate, simpler function that only deals with Gaussian constraints, which are by far the most common kind. The data portion is handled by the `Pdf`'s, one for each data set. For this analysis we have two three dimensional `Pdf`'s, one for the day data and one for the night.

Each `Pdf` has a simple task: it takes a the current $\vec{\alpha}$ as input and returns the corresponding log likelihood. To this end, it contains the data set $\{\vec{x}\}$ and a number of `Flux`'s and `Sys`. Each `Flux` corresponds to a flux as we have been referring to it so far: it is a repository of the MC events $\{\vec{y}\}$ for a single signal or background. Each `Sys` takes a single MC event $\vec{y}_i$ as input and returns how the vector has changed, the $\Delta\vec{y}_i$ from Chapter **??**. The Pdf thus takes one flux at a time, applies the relevant systematics (not all systematics are applied to all fluxes, see Section **??**) to it one event at a time, then fills that event in to a histogram with the appropriate weight, all of which are calculated by the `Sys`. Once all the fluxes have been processed, the histogram is renormalized to create the binned Pdf. The log likelihood is then computed by finding the value of our Pdf corresponding to each data event and summing, then adding the extended log likelihood component at the end to give the ELL($\vec{\alpha}$) described in Section 2.3.

3.2.2 Classes

In object-oriented programming, the majority of the code and actual computation is handled through classes. The code has ten classes, three of which are "helper" classes that aren't directly involved in the signal extraction, each of which handles a particular aspect of the process. To explain how the code as a whole works, we must first describe how each class works. Thankfully, the nature of object-oriented programming allows us to ignore most of the internal details of a class when it interacts with another class, only the inputs and outputs matter. In this explanation, we will strive to separate the descriptions in to a short explanation of how the "outside world" sees the class, then a longer explanation of what happens inside the class. We will start from the most basic classes and build up to the most complicated, as the topmost class (the `MCMC` class) contains within it instances of almost all the other classes.

ConfigFile

`ConfigFile` is the first helper class, and the only code not written expressly for this project. It is an open-source config file reading utility, written by Richard J. Wagner at the University of Michigan [3]. This is the only code we inherited from and have in common with [9]; that analysis downloaded this class from an online repository. This rather simple class reads in a file that consists of a list of lines of the form `key = value` and creates an internal structure allows one to either traverse the list of keys or reports the value associated with a particular key. It is very similar to the `map` class in the C++ standard library. We will refer to this library by its customary acronym STL for brevity.

Tools

`Tools` is the next helper class. As the title suggests, it contains a set of tools that were useful across most of the other classes. The first of these utilities is `SearchStringVector`, which looked through an STL vector (effectively a list) of strings for a particular string. This was used a great deal during set up, as the various program components were kept track of by a string (their "name"). The next is `ParInfoToString`, which converted internal descriptions of components to the external, more human-readable ones used in the config files. The third is
`DoublesAreCloseEnough`, which compares to doubles to see if they only differ in the last few bits - this allows checking if two doubles are "equal" within machine rounding. The final tool is `VectorScramble`, which randomizes the order of entries in an STL vector. This was used mostly for debugging.

Errors

The final helper class is `Errors`. It is a very simple class that contains an STL vector of strings. This is a global list, so that any time any piece of code adds to it, all of the program can see it. This was used for error handling - any time a piece of code encountered an error, for example a malformed configuration file entry or a non-existant file, it reported this to `Errors`. This allowed the program to process things as best as possible instead of immediately exiting, allowing multiple problems to be seen at once. At the end of the setup phase or when an error that could not be overcome was encountered, the `Errors` class printed all of the accumulated error messages to the screen and exited the program cleanly, rather than crashing. This was extremely helpful any time the config file was changed, as it often caught all the typos in one run.

RealFunction

`RealFunction` is a simple class that takes an arbitrary number of double (real number) inputs and returns a single double, i.e. a code implementation of a simple multivariable function $f(x, y, z, ...)$. Another class can set the values of each input with `SetParameter` and ask for the computed value by calling `Eval`. It is actually a virtual parent class, with each daughter class being specified to perform one computation. For example, one daughter class takes two inputs and returns their product, and another takes four inputs (a, b, c and x) and returns $a + bx + cx^2$. So what function `RealFunction` performs is decided at set up time by choosing one of the available daughter functions, which have the property that the code can't distinguish between them and `RealFunction` in terms of use.

Internally, `RealFunction` contains an array of doubles, one element for each input. The `Eval` function is different in each daughter class, but it is almost always just the real function being performed. Unfortunately, that means that if a function not already available is needed, a new daughter class must be written and the code recompiled (these daughter classes are very simple and are all defined in `FunctionDefs.h`. This naturally raises the question of why this was done this way, rather than writing a general purpose class that can take any arbitrary function without needing to specify beforehand. The answer is simple: speed. This general purpose function exists in ROOT as `TF1`, but is much, much too slow to be useful. This implementation is over ten times faster, and this function gets called billions of times during a typical run, meaning that speed is critical. In fact, evaluation of RealFunction is more than 30% of the total running time of the program.

Flux

The `Flux` class keeps track of the MC events for a single flux or background. It is the first "named" class - it has a unique string value that identifies it, so each instance must have a name that is unique in the program (not shared with any other named structure). This allows for it to be looked up and called in other classes by name rather than some more abstract way, which is necessary as part of the setup process. During setup, it reads the MC events from a config file specified file. It also reads from the config file the `TimesExpected` (described in Section 3.1) and a quantity called `FluxNumber`; the former is a scaling factor that determines the nominal number of data events for this flux, the latter is used by `MCMC` to decide whether a given `Sys` should act on this flux. During running, this simple class returns an individual MC event $\vec{y}_i$ when called. It can return events in any order.

Internally, the MC events are stored in memory as a large array, for speed. They are returned as a vector of doubles of length equal to the length of $\vec{y}_i$ plus one, with the extra element being the weight W, set to 1 and altered by the `Sys`. `Flux` has member functions to return its name, number of MC events, `TimesExpected` and `FluxType`

Sys

The `Sys` class applies a function to a set of values. It is another named class, again its name must be unique within the program. At its heart, an instance of Sys contains either a `RealFunction` or a `TF1` that returns a single double given a set of parameters. The `Sys` keeps track of which parameters out of $\vec{\alpha}$ it needs, as well as any parts of $\vec{y}_i$ it needs, and hands these to the underlying function. It then records the results in a separate vector with the same size as $\vec{y}_i$, which it returns. This allows for all `Sys` to have access to $\vec{y}_i$ as it

comes from the Flux. Each Sys also contains a list of FluxNumber's that it affects, and returns either true or false when asked if it affects a particular FluxNumber. Most instances are within a Pdf, but the class also serves double duty in that it is used to implement both the LogLikelihoodFunction's and the AsymmFunc's, described in the MCMC description.

As mentioned, Sys contains either a RealFunction or a TF1. These are nearly identical in function, save that RealFunction has a very short list of functions it can perform but is fast, while TF1 can perform an arbitrary function but is slow. During setup, the Sys is given a list of all the parameters in $\vec{\alpha}$ (called the mcmcPars) and all of the components of the MC events (called the branchPars or branches). The config file tells the Sys what function to use and what values it needs in what order, out of these two lists. Since this is a bit error prone, many of these are hard-coded into the Sys class and automatically selected by the Sys's name, though this can be overridden. Sys keeps track of which values it needs to pass to the underlying function, in the sense that it stores these values as class members. Each Sys only alters one component of $\vec{y}_i$, called the "target", just as each systematic does. This is necessary, both to follow the logical structure of the analysis and because the underlying function only provides a single output. The config file (or the automatic selection) sets whether the computation is done using the $\vec{y}_i$ from Flux or $\vec{y}_i + \Delta\vec{y}_i$, i.e. the "new" value incorporating all Sys applied thus far. It also sets whether the output value should be added to the target element (most systematics) or multiply the target element (the weight altering systematics). Finally, it reads from the config file (or the automatic selection) a list of FluxNumber's, which it stores. During a run, a Sys takes the list of parameters $\vec{\alpha}$ and updates its internal values to correspond to the new list. A member function takes a FluxNumber as input and returns either true or false, corresponding to whether it acts on that FluxNumber or not. This happens once per Flux. Finally, the Sys is given an event $\vec{y}_i$ and the changes made thus far to its value $\Delta\vec{y}_i$ and updates those changes.

Pdf

This actually encompasses three classes: PdfParent, Pdf1D and Pdf3D. Pdf1D and Pdf3D implement a 1D and a 3D histogram/pdf, respectively, while PdfParent handles tasks that are common to the two. We will ignore the division of labor and dimensionality differences and speak of a Pdf class. From the viewpoint of outside entities, it only has two tasks: it reads from the config file, data file and MC file the information it needs (and sets itself up) and it returns its contribution to ELL($\vec{\alpha}$) when given $\vec{\alpha}$.

Internally, this class coordinates the activities of the other classes (except MCMC). It stores the data events $\{\vec{x}\}$ in the same way that Flux stores the MC events. Also, it contains the histogram that is to be filled with the MC data to create $N\,p(\vec{x}|\vec{\alpha})$, a TH1D for Pdf1D and a TH3D for Pdf3D. During setup, it reads from the config file which fluxes it should use when building its Pdf and which systematics (including things treated as systematics, such as P_{ee}), creates instances of Flux and Sys as appropriate, and gives them the information they need to construct themselves. In particular, each element in the MC event vector $\vec{y}_i$ has a name, given in the config file, which must be consistent across all MC sets. The config file specifies what file the Pdf's data is in, which it reads in to an internal array, with the element names also specified in the config file as for the MC events. During a run, the Pdf is given a new value for $\vec{\alpha}$, which is passes along to the Sys. The histogram is then emptied. Then the first Flux is asked for its FluxNumber and each Sys is asked if it acts on this flux, creating a list of Sys that act on this Flux. The Flux is then asked for its elements one at a time, with each acted upon by the appropriate Sys to create $\Delta\vec{y}_i$. The histogram is then filled with $\vec{y}_i + \Delta\vec{y}_i$, with weight given by equation 3.1. The Scale term is special, in that it is a parameter with the same name as the Flux that is automatically created by MCMC when the Flux is created. This is then repeated for each Flux and the histogram is rescaled as explained in Section 3.1. Then the Pdf takes each data event, finds the corresponding bin in the histogram, and adds the logarithm of the histogram's entry to log likelihood (which is reset to zero when this process starts, so only this Pdf and this step's values are used). The extended log likelihood is computed by subtracting the total number of events. This value is returned.

MCMC

The MCMC is the highest level class. There is only one instance per program. In fact, other than some input processing and a timer, the main program consists of having MCMC read the config file then calling MCMC.Run(). From the outside point of view, this class is a black box that takes a config file as input, then performs the analysis (including reading the files and writing the output).

As most of the work is actually done by `Pdf` and the classes inside it, the `MCMC` class mostly acts to coordinate. During setup, it reads from the config file those parameters directly governing the MCMC chain behavior, such as the number of steps in the chain, how often to print to the screen (every n^{th} step it prints the current parameter values) and which extra information about the parameters to save (such as the proposed values $\vec{\alpha}_i^{prop}$ that are rejected). It reads the name of the output file and creates both that file and the `TTree` (a ROOT data storage class) that will be written to the file. It reads the number and dimension of the `Pdf`'s and creates the appropriate `Pdf`'s, then passes the config file to each to allow it to construct itself. It then reads out of the config file all of the parameters that make up $\vec{\alpha}$, most of which are explicitly parameters, but one is automatically created and shares a name with each `Flux` and `Sys`. For each of these parameters, it reads in the initial value, maximum and minimum value (if defined), step size and (if applicable and Gaussian) the mean and width of the parameter's constraint. Finally, it reads from the config file the setup information for the `LogLikelihoodFunction`'s and the `AsymmFunc`'s. These are special instances of the `Sys` class that do not act on the MC events. Instead, the `LogLikelihoodFunction` uses the `Sys` machinery to create a function that takes some number of the parameters as an input and outputs a number that is directly added to the log likelihood at each step in the MCMC. This is used to compute values for non-Gaussian constraints, which can be arbitrary functions, and for direct contributions to the likelihood such as LETA and PSA. The `AsymmFunc`'s are used to alter the values of parameters based on other parameters. This allows a parameter such as the energy scale to be computed - it is actually the sum of two other parameters. It is also used to compute the values for the day and night versions of a parameter from $\bar{\alpha}$ and A_α, as the name might suggest. Once this setup is complete, the `MCMC` unsurprisingly run the Metropolis algorithm. At each step, it varies the values of the parameters according to their step sizes (which can be zero) and adjusts them according to any `AsymmFunc` to create the $\vec{\alpha}_i^{prop}$. If any parameter is above its max or below its min, the step is rejected. Otherwise, the log likelihood is reset to zero and the contribution from each constraint, each `LogLikelihoodFunction` and each `Pdf` is computed and added. Then the Metropolis algorithm decides whether to keep or reject the step. If it is kept, the proposed parameters replace the old parameters, if not the proposed parameters are rejected. The values of the parameters are then added to the `TTree`. When the whole algorithm has finished (i.e. it has taken the prescribed number of steps), the `TTree` is written to the output file, stray memory is cleaned up and the program exits.

Details about how to interact with the various aspects of the program, in particular how to write a config file to use the program, are given in Appendix 1.

Bibliography

[1] Blair Jamieson. SNO NCD phase signal extraction on unblinded data with integration over systematic nuisance parameters by Markov-chain Monte Carlo. `http://manhattan.sno.laurentian.ca/sno/ananoteb.nsf/URL/MANN-7NCU9S`, 2008.

[2] CERN. Root homepage. `http://root.cern.ch`.

[3] Richard J. Wagner. http://www-personal.umich.edu/ wagnerr/ConfigFile.html.

[4] F. James. MINUIT reference manual. `http://wwwasdoc.web.cern.ch/wwwasdoc/minuit/minmain.html`.

[5] Phil Gregory. *Bayesian Logical Data Analysis for the Physical Sciences*. Cambridge University Press, 2005.

[6] E. T. Jaynes and G. Larry Brelthorst. *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.

[7] Radford M. Neal. Probabilistic inference using markov chain monte carlo methods. Technical Report CRG-TR-93-1, University of Toronto, 1993.

[8] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *Journal Of Chemical Physics*, 21:1087–1092, June 1953.

[9] B. Aharmim et al. Independent Measurement of the Total Active $B8$ Solar Neutrino Flux Using an Array of $He3$ Proportional Counters at the Sudbury Neutrino Observatory. *Phys. Rev. Lett.*, 101(11):111301, Sep 2008.

[10] Yves F. Atchad and Jeffrey S. Rosenthal. On adaptive markov chain monte carlo algorithms. *Bernoulli*, 11(5):pp. 815–828, 2005.