RapidFit
A beginners Guide, Talk 1 of ?

Robert Currie, Edinburgh University

Introduction

- How To Setup RapidFit. **(Properly!)**.
- What is a PDF in RapidFit?
- My First PDF & XML
- My First Projection
- My First ΔLL Scan
- My First Toy Study
- Analysing My Output(s)

Next Week and Future Talks...

- Expanding Upon DataSet(s)
- More on ΔLL Scans
- My First ΔLL Contour
- More on Projections
- More on Toy Studies
- More on XML Options
- More Runtime Arguments
- Running on ECDF/The Grid
- Complex studies with RapidFit
- Commonly used RapidFit C++ objects
- Some features of the utils codebase
- ...Anything else? What are people most eager to have a tutorial on?

Introduction

All of the code will be available soon. (my work on $J/\psi\ \phi$ willing)

All of the XML I present will be made available in svn HEAD
(They are not there at this time, apologies)

The more feedback I get the more I can improve future talks/documentation.

What do I need to Get RapidFit Setup?

- CERN computing account
  (preferably with lxplus/AFS privileges)
- Access to RapidFit repository
  (speak to Dean Lambert CERN)
- UNIX skills (ssh, make, gcc, etc)
  (We have NO plans to support windows. EVER!)
- This Tutorial

(alternatively speak to myself and I will advise you on what to do)

What Does RapidFit Need?

- ROOT
  Almost everything you do in PPE relies on ROOT.
  It's a love-hate relationship, try to get past the fact it's ugly.

- bash
  RapidFit and a lot solutions I can provide are bash based.

- (Otional) GSL libraries
  This is required for few special analyses, but is available for everyone to use.

- UNIX
  We rely on common POSIX and STL over ROOT solutions.

- Future dependencies/developments include: CUDA / OpenCL
  Ask Dianne

How to Setup RapidFit

- Firstly Get the RapidFit source code:

  ```
  svn co svn+ssh://cern-username@svn.cern.ch/reps/lhcbedinburgh/Code/Fitting/RapidFit/trunk
  ```

- Configure ROOT:
  - on lxplus: SetupProject ganga
  - compile it yourself with '–all' and 'source /some/path/thisroot.sh'
  - install from your distro's Repo i.e. Ubuntu (ask Google)
  - compile using homebrew and XCode on OSX (ask Google)
  - We rely on many ROOT 'optional' tools. When in doubt: Don't build yourself!

- Optional make sure gsl-config is in your $PATH

- Build RapidFit with 'make' or 'make all'

Building RapidFit

- RapidFit uses GNU make.
  This simply generates a lot of gcc commands to build source.
  This is configured by the 'Makefile' in the top of the trunk.

- The options to make you should know are:
  - make or 'make all'
    This builds everything
  - make utils
    This builds the tools in the utils folder
  - make bin/fitting
    This builds RapidFit
  - make gsl
    This builds RapidFit with optional GSL addons.
  - make clean
    Clear the old build files after the code has changed
    Use after 'svn up'

Why use Compiled Code?

There are several extremely good reasons as to why we use compiled code:

- It's faster
- The compiler knows better C++ than me!
- The compiler knows better C++ than you!
- It's really, really fast
- It forces you to think about what you're doing
- Stops problems due to really bad code that can be in CINT

ALWAYS: rebuild after changing your code.

BEFORE: you claim you've found a bug can you repeat it after a make clean?

XML & PDF

In order to run your fit you need to have an XML to configure RapidFit.

The XML consists of:

- ParameterSet
  What am I trying to measure?
- ToFit(s)
  What am I trying to use to make a measurement?
- FitFunction
  How am I defining how well my ParameterSet matches the ToFit(s)
- Minimiser
  What I am using to calculate the best ParameterSet for my ToFit(s)

And a simple XML, one can fit on a single overhead!

```
<RapidFit>
<ParameterSet>
<PhysicsParameter>
<Name>sigma</Name>
<Minimum>0.</Minimum>
<Maximum>5.</Maximum>
</PhysicsParameter>
</ParameterSet>
<Minimiser>Minuit</Minimiser>
<FitFunction>NegativeLogLikelihood</FitFunction>
<ToFit>
<PDF>
<Name>SimpleGauss</Name>
</PDF>
<DataSet>
<Source>Foam</Source>
<NumberEvents>1000</NumberEvents>
<PhaseSpaceBoundary>
<Observable>
<Name>x</Name>
<Minimum>-10.</Minimum>
<Maximum>10.</Maximum>
</Observable>
</PhaseSpaceBoundary>
</DataSet>
</ToFit>
</RapidFit>
```

A simple PDF

OK so I have a simple fit in mind:

1: PhysicsParameter: sigma
1: Observable: x
1: PDF: 'SimpleGauss'
1: DataSet: Foam 1,000 events

How do I construct SimpleGauss to allow me to do this?

Constructing SimpleGauss

XML requirements:

SimpleGauss has to be constructed to tell RapidFit what Observables it expects to be provided. (x)
This is done by populating: SimpleGauss::allObservables;

It also has to tell RapidFit what PhysicsParameters it wants to measure.(sigma).
This is done by populating: SimpleGauss::allParameters;

Constructing SimpleGauss

C++ Requirements:

SimpleGauss must 'advertise' itself to RapidFit as a PDF:
use:
PDF_CREATOR(SimpleGauss);

SimpleGauss must provide a method to Evaluate the Probability of finding a DataPoint:
SimpleGauss::Evaluate(DataPoint*);

SimpleGauss should (not must) provide a method to Normalise the Probability of finding a DataPoint in a whole PhaseSpace.

SimpleGauss::Normalise(PhaseSpaceBoundary*); or
SimpleGauss::Normalise(DataPoint*, PhaseSpaceBoundary*)

Constructing SimpleGauss PDF

```
#include "SimpleGauss.h"
#include <string>
#include <vector>
#include <cmath>
using namespace::std;
PDF_CREATOR( SimpleGauss );
SimpleGauss::SimpleGauss( PDFConfigurator* config ) :
  xName( "x" ), sigmaName( "sigma" )
{ this->MakePrototypes(); }
void SimpleGauss::MakePrototypes()
{
  allObservables.push_back( string(xName) );
  allParameters = ParameterSet( vector<string>( 1, string(sigmaName)) );
}
SimpleGauss::~SimpleGauss() {}
double SimpleGauss::Evaluate( DataPoint* input )
{
  double sigmaVal = allParameters.GetPhysicsParameter( sigmaName )->GetValue();
  double xVal = input->GetObservable( xName )->GetValue();
  double numerator = exp(-(xVal*xVal)/(2.*sigmaVal*sigmaVal));
  return numerator;
}
double SimpleGauss::Normalisation( PhaseSpaceBoundary* range )
{
  double sigmaVal = allParameters.GetPhysicsParameter( sigmaName )->GetValue();
  double denominator =(sigmaVal*sqrt(2.*Mathematics::Pi()));
  return denominator;
}
```

Running the Fit

In order to run RapidFit you need to know the options to pass to it.

RapidFit as a binary when you have built the code is:
'./bin/fitting'

A standard fit of one XML is run by using:

'fitting -f someXML.xml'

Some basic help and information exists:
'fitting –help' and 'fitting –about'

More about the various Runtime options next week.

Aside- Where is My Data coming from?

In this example, and as with all Toy Studies and simple tests of the PDF the Data is generated using the PDF itself.

This is because I have used:

```
<DataSet>
<Source>Foam</Source>
...
</DataSet>
```

When this source of Data is used the ParameterSet in the XML is taken to be the truth or CV to be used in constructing the DataSet.

The only function in the PDF used for generating the DataSet is PDF::Evaluate(DataPoint*);

More on this when it becomes important!

How do I process Output?

I have written a nice tool which does 90%+ of what you want a tool to do in post-processing from the RapidFit Results.

It is called 'RapidPlot'. You can pass it the output from toys, ΔLL scans and contours as well as the results from FC analyses.

The tool makes use of a highly modular code-base trying best to follow the 'rule of 7' (7 functions per class each with 7 lines each doing a specific job)

Plots are almost publication ready (you may need to make tweaks for referee's but they're almost good enough to publish out of the box).

My First Projection

Projections allow us to get a handle on the question of:
“How well does my Fit Describe this Observable?”

In order to run a projection in RapidFit you need to add a new XML segment to the file:

```
<Output>
<Projection>x</Projection>
</Output>
```

This tells RapidFit to project 'x' for all ToFit segments in my XML.

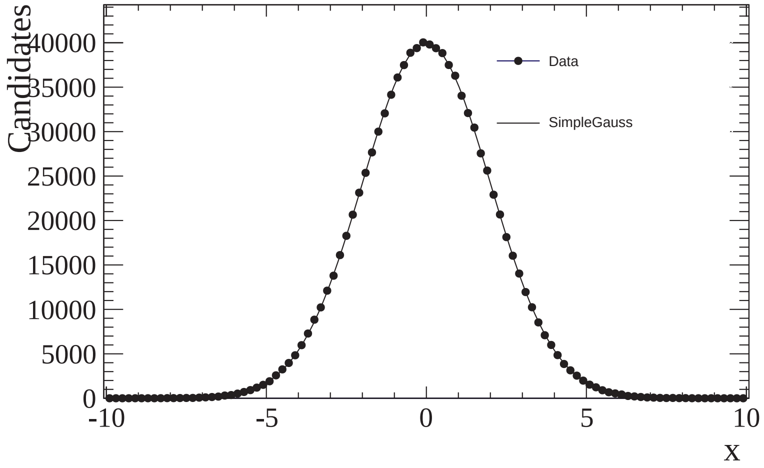
Projections allow you to determine how well you describe the distribution of a given Observable.

You can alternately use:

```
<Output>
<ComponentProjection>x</ComponentProjection>
</Output>
```

(I will give more examples on this next week I'm covering enough for now.)

My First Projection - Output



My First ΔLL Scan

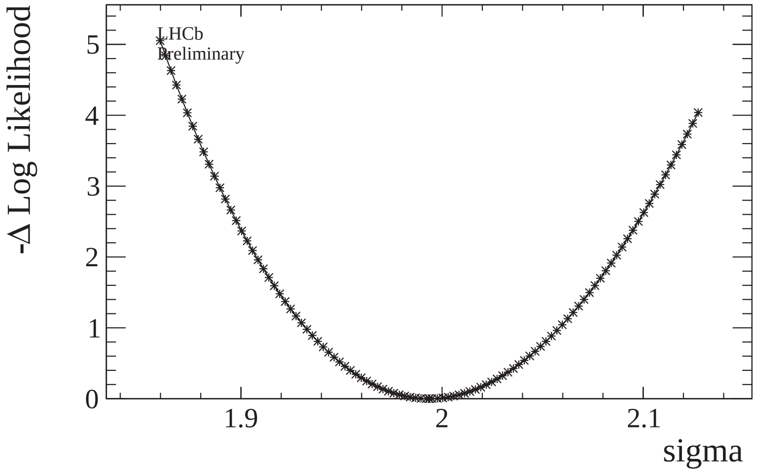
Running ΔLL Scan(s) allow you to determine:
“How well behaved is a given parameter in my fits?”

In order to perform a scan you need to add another section of XML to your XML file.

```
<Output>
<Scan>
<Name>sigma</Name>
<Sigma>3</Sigma>
<Points>100</Points>
</Scan>
</Output>
```

This code is NOT run by default when you run a fit, you need to request it to be run with '–doLLscan' when you launch RapidFit.

My First ΔLL Scan Output



My First Toy Study

Toy Studies allow you to address the question:
"How Reproducible is my Result?"

To do this you need to repeat your fit with many toy Datasets.

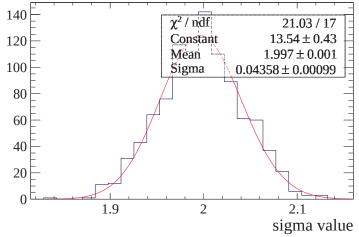
You can instruct RapidFit to do this in 2 ways: XML:

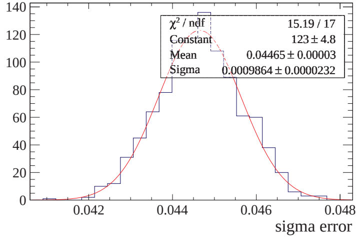
```
<Repeats>100</Repeats>
```

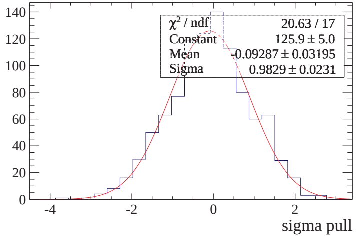
or:
pass '-repeats 100'

to repeat 100 toys.
You generally want between 1,000 toys and 100,000 toys depending on how accurate you want to be about your statements.

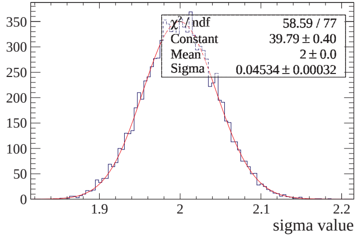
My First Toy Study Output (1k studies)

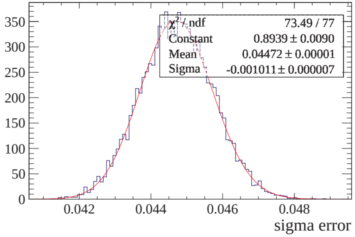


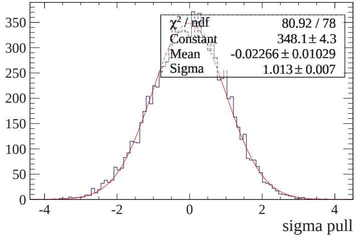




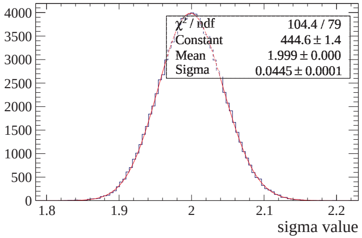
My First Toy Study Output (10k studies)

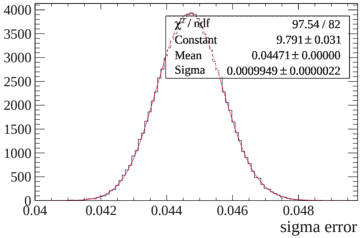


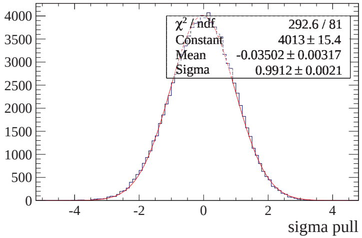




My First Toy Study Output (100k studies)







Summary

I've gone through the minimal required tools to run:

- Simple Foam fit
- ΔLL scan
- Toy Study
- The most minimal PDF
- The most minimal XML

A Minimal introduction to RapidFit.

I will upload example PDFs and XML and email around where to find these in svn. I will be adding to this and can give a talk about once a week for the next month or so on useful RapidFit tools for everyone.

RapidFit Utils

I want to advertise the 'tools' in the utils directory.

Don't re-invent the wheel when it comes to working 'with' ROOT.
I have many tools written to quickly/efficiently get information in/out of ROOT objects and into STL template objects.

This includes many template functions.

The code has been separated into source files building a small 'library' of common tasks which have been 'automated'.

Why care about things in the Utils Directory?

There exist **many** pre-written functions to do things like:

Read in a branch of a TTree into a vector. **2 lines of code!**
Read in a branch with a cut applied. **2 lines of code!**
Names of TBranches in a TTree. **1 line of code!**
Opening a TTree in a file regardless of name. **1 line of code!**
List all Histograms in a file. **1 line of code!**
Opening a File and checking it opened safely. **1 line of code!**
Make a Histogram from a vector safely. **1 line of code!**
Add a new TBranch to a TTree correctly. **1 line of code!**