RapidFit

A developers Guide, Talk 3 of ?

Robert Currie, Edinburgh University

Introduction

- Useful development tools in RapidFit
- Useful debugging tools in RapidFit
- Copying complex PDF objects
- Advanced Fitting
- RapidFit or RapidRun?

Useful Development Tools

There are some powerful commonly used functions coded up in various places in RapidFit:

- ClassLookUp.h
  This class provides powerful tools for copying objects through interfaces
- StatisticsFunctions.h
  This class provides methods for processing vectors of lots of numbers efficiently
- StringProcessing.h
  This class provides methods for efficient (T)string manipulation
- EdStyle.h
  Very closely based on the LHCb style this provides static methods for outputting nice looking plots

Useful Debugging Tools

DebugClass.h
This class provides a lot of static methods for debugging your code in RapidFit.
(Technically this is a C++ sentinel with a lot of static methods bundled in but this is an aside...)

This class has a very powerful method of:
DebugClass::DebugThisClass(string myClassName);
This method allows you to construct code and output in a similar way to the LHCb framework which isn't executed by default by RapidFit but can be turned on/off without recompiling.

Another set of methods are:

- Dump2File(fileName, vector)
- Dump2TTree(fileName, vector, TTreeName, TBranchName)
- Dump2File(fileName, vector<vector>)

These allow you to quickly and easily dump the contents of a vector(<vector>) to a file from somewhere in RapidFit, useful for numerical debugging.

Useful Debugging Tools Cont.

Usage:

```
if( DebugClass::DebugThisClass( "myCustomClass" ) )
{
  cout << "Some value here is: " << someValue << endl;
}
```

Now this code will NOT run by default unless you execute RapidFit like:

```
./bin/fitting -f myAmazingFitXML.xml --DebugClass myCustomClass
./bin/fitting -f myAmazingFitXML.xml --Debug
./bin/fitting -f myAmazingFitXML.xml --DebugClass SomeOtherClass:myCustomClass
```

Note: running with - -Debug will cause ALL of the debugging in RapiFit to be turned on, this is EXTREMELY verbose!

Copying Objects in RapidFit

Most classes in RapidFit have private pointers to objects. This allows for the scope of an object to be more easily seen (in this authors opinion).

As a result of this most objects require custom copy constructors to be written. Many methods of these already exist, and it is chosen that C++ objects should be copied through their copy-constructor with most copy by assignment operators being protected or private methods which are often not implemented.

Complex objects are passed through their interfaces and there exist abstract copy methods to perform a deep-copy the objects through their interface, these exist in ClassLookUp.h.

Many objects can be copied through their copy constructor with some exceptions such as IDataSet objects which should be instantiated once and passed by reference only due to their obviously large memory footprint.

Copying complex PDF objects

RapidFit allows you to run multi-threaded highly parallelized fits using your PDFs. This is a good thing.

However, if you insist on using the more advanced features of C++ be aware your class MUST be thread-safe and have a fully working copy constructor.

A LOT of work has been done in the framework which allows you to avoid the latter of these by calling:
`this->SetCopyConstructorSafe( false );`
within your PDF constructor.

This causes RapidFit to create a new instance of your PDF rather than copying it. (This potentially slower and may introduce unknown bugs, it is safe, but shouldn't be used unless necesarry)

Advanced Fitting

RapiFit allows for a lot of flexibility from the command line which can overload the options configured within the XML.
Remember:
Commandline ALWAYS takes priority in configuring things

You can run a fit with multiple XML files, this has several advantages/disadvantages.

```
./bin/fitting --files 2 myXMLfile1.xml myXMLFile2.xml
```

This will pick up multiple ToFit segments from multiple files using the FitFunction and Minimiser as configured in the first XML file.
When an object is defined in XML multiple times i.e. the same PhysicsParameter exists in 2 files, the first definition takes priority.

XML Hacking

There are 2 VERY powerful tools at your disposal to change any XML object in the existing XML file(s).

- -DebugClass XMLConfigReader_TAG
and
- -OverrideXML /RapidFit/something/myTag newValue.

The first option "- -DebugClass XMLConfigReader_TAG" is a special case of the DebugClass as there is an intentional exit statement built into the code. This will dump a lot of internal info about the XML file as it has been read into RapidFit including the path value and numbers of children for each XMLTag in RapidFit.

The second option "- -OverrideXML /RapidFit/something/myTag newValue" overrides the value of an XMLTag at the path: "/RapidFit/something/myTag" with the "newValue".

A word on OverrideXML

Override is only possible due to RapidFit's (delicate) simple XML parser.
It would be nice to port RapidFit to use a proper XML parser however this would remove this powerful feature and make commenting out single lines of XML more difficult.

In RapidFit we can ignore a whole line of XML by pre-pending it with a '#' symbol. THIS IS A RAPIDFIT-ISM.
The XML standard DOES NOT do this.

However when developing a fit we have found this is quick and simple and easier than wrapping each single line in:

"<!-- ... -->".

fitting vs RapidRun

As RapidFit can be compiled as a ROOT library (libRapidRun.so) RapidFit can be loaded by ROOT as part of a script.

This allows for potentially high level analyses to be relatively easily constructed e.g:

```
# We want ROOT
import ROOT
# Import RapidFit
ROOT.gSystem.Load("lib/libRapidRun")

# RapidFit arguments
args = ROOT.TList()
# Construct a DataSet and Save it
args.Add( ROOT.TObjString( "RapidFit" ) ) args.Add( ROOT.TObjString( "-f" ) )
args.Add( ROOT.TObjString( "myXML.xml" ) ) args.Add( ROOT.TObjString( "--saveOneDataSet" ) )
args.Add( ROOT.TObjString( "myTestFile.root" ) )

# Modify this DataSet
someCustomFunction( "myTestFile.root" )

# Perform a fit to the Modified DataSet
args.Add( ROOT.TObjString( "RapidFit" ) ) args.Add( ROOT.TObjString( "-f" ) )
args.Add( ROOT.TObjString( "myXML.xml" ) ) args.Add( ROOT.TObjString( "--OverrideXML" ) )
args.Add( ROOT.TObjString( "/RapidFit/ToFit0/DataSet/Source" ) ) args.Add( ROOT.TObjString( "File" ) )
```

Bonus! Some hints for spotting and crushing bugs

My Code leaks memory	try running with: valgrind - -tool=massif ./bin/fitting -f someXML.xml and use massif-vizualiser to interporate your output
My Code is taking too long to run	try running with: valgrind - -tool=callgrind ./bin/fitting -f someXML.xml and use kcachegrind to interporate your output
My Code crashes and I'm certain it's not my fault	Try reading the stacktrace which ROOT generates and make sure it's not in your
I suspect there is a bug in my code	Compile it with 'make clang' and 'make debug' to see more output from the compilers about your code quality

If you build with 'make valgrind' and run with 'valgrind - -tool=callgrind - -instr-atstart=no' valgrind will only benchmark projections and the code that is run during the fit, not the whole RapidFit framework.

Conclusions

I've highlighted some of the pre-written tools and method within RapidFit to people. Please consider using them before re-inventing the wheel where possible.

I've pointed you to some debugging tools in the codebase. They're a little more powerful than the simple "cout" and "exit", but their use is up to you.

I've shown some powerful tools in OverrideXML, DebugClass and the pyROOT wrapping of RapidFit. Have fun fitting!