**EMPHATIC MAGNET ANALYSIS CODE NOTES**

Leo Bellantoni

24-May-24

## Summary

This is the documentation for the code developed to create plots for and construct a stitched magnetic field map for the first “China” or “Phase I” magnet of the EMPHATIC experiment.

As for the deeds of this magnet, are they not written in the DocDB of EMPHATIC, particularly entries 1608v1, 1768 and 1643?

[Summary 1](#_Toc167433242)

[The L++ framework 2](#_Toc167433243)

[The magnet analysis code 3](#_Toc167433244)

[About histograms and GUIs 4](#_Toc167433245)

[About ra<T> 5](#_Toc167433246)

[About FieldMap 7](#_Toc167433247)

[A step-by-step analysis 9](#_Toc167433248)

[Appendix A: The L++ man page: 12](#_Toc167433249)

[Appendix B: A guided tour through findOffs.h: 16](#_Toc167433250)

## The L++ framework

The Helmholtz decomposition code was developed on a MacBook Pro running OS X 12.7.2. The preferred development environment there is Xcode (v14.2 14C18), the Apple IDE. Using Xcode I’ve developed a library of various bits of code that proved useful to me at one time or another. The original intent of this construction was for I, Leo, to learn C++. Hence the name for the framework, L++. The man page for L++ is in appendix A.

In porting L++ to Linux, some modifications had to be made. The biggest single change is the replacement of the Xcode build system with a GNU make Makefile. As a general rule, dependencies in the sources actually used to do the calculation itself are listed. Dependencies for the L++ sources also used are not listed in the Makefile, although the Lpp: and LppD: targets build the L++ sources which are required here. The 2 high level targets in the Makefile are MAIN and DEBUG. The former, which is the default, builds the code in optimized format and the latter, in a form that could be used by gdb or ddt. The #include file for lapack could not be found for some TBD reason on this still relatively new Alma 9 system, so a temporary copy of this widely distributed file has been used. The #include statements had to be changed because Xcode’s header search list isn’t quite there in the Makefile.

The original L++ project, which should be runnable on any Mac with Xcode and gnuplot installed, is in doc/EMPHATIC\_Magnet\_analysis.tgz, along with this document.

Unfortunately, the use of gnuplot via a pipe created with the libc function popen() has a problem in that as gnuplot talks to x11, it (or x11) gets confused about which command goes to which plot. Because emphaticgpvm nodes take quite a while to render a plot on many remote clients with x11, this can really screwup the plots. At this time, there is a 60 second timeout to allow each plot to be rendered before proceeding to the next plot. This may be changed by altering the value of plotWaitTime, which is in seconds, found in GUI/plotbase.h. It is probably too high.

If you can’t stand waiting, you may edit the code in main.cpp to output a comma separated values (.csv) file and skip the displayed plots. See below for a description of the L++ GUI and histograms classes.

## The magnet analysis code

The key files used in the calculation and checking of the Helmholtz decomposition are:

* Inputs.h – this file takes the place of e.g. *art*’s fcl files. It contains parameters that you might want to change for each run of the code. In Xcode, the build system is quite fast for a project of this size, and it is just as easy to use C++’s strong type checking by including a bunch of const declarations.
* main.cpp – the code that will be made and executed.
* Data\_in/EMPHATIC\_\*.csv – these 3 files are the data from the Fermilab AP-STD’s measurement of the field for the CMSM magnet. There are a few minor edits to the upstream (smallApertureFringe) and downstream (largeApertureFringe) files, as described in EMPHATIC DocDB 1643. A few obviously erroneous data points have been cleaned up, as described there. These 3 files are the entirety of the measurement data we have.
* Data\_in/CylinderGrid\_Section\*.txt – these 3 files are data from the Fermilab AP-STD’s measurement of the field for the SABR magnet. Other data taken for this magnet are in the other\_data subdirectory. The CylinderGrid\* files are of the form useful for analysis here. But not very useful, as they only span from the upstream aperture of the magnet to 2 cm from the downstream aperture thereof, and we need the fringe fields. Other issues with the data here are documented in EMPHATIC DocDB 1847. At this time, we anticipate creating our own magnetic field maps for the SABR magnet.
* FieldMap and interpolate.cpp – these are the central class of the analysis, and will be described below.
* Probe\_N\_magnet\_info/\* - these are information about the Hall probe used to measure the magnetic field. They are not directly needed, but are kept for reference.
* integrands.h - this is not to be confused with the file of the same name in the Helmholtz decomposition package. Both are code to implement certain numeric integrals, but the integrated quanties are quite different.
* CodeGuts/xxx.h, where xxx ∈ {showXYscan, divHisto, showBvsZ, sliceMag, SliceDiv, sliceVec, sliceBz, BdlHisto, plotOverlapDiff} – these use gnuplot to plot the differences between the Helmholtz decomposition. As of this writing, gnuplot is not available in the containers that sandbox SL7 on the emphaticgpvm nodes. However, it is available in Alma 9, so when the SL7 → Alma 9 transition is complete, that should not be an issue. The file sliceVec.h is slightly different from the version used in the Helmholtz decomposition code. Much of the contents of the directory CodeGuts is code that logically could be put into main.cpp as these are methods that are only executed once. However, their inclusion in main.cpp would make that code bulky and unwieldy.
* histograms/\*.h and GUI/\*.h – these are the classes for histograms, i.e. objects with bins data, and GUI objects, which can be displayed on the screen. These are described, in part, below.
* ra/ra<T> - understanding this template class is important to the user. Again, detailed below.

### About histograms and GUIs

Important histogram classes for this work are Lhist1, Lhist2 and Ldata. These correspond to the data for 1 and 2 dimensional histograms and both vector and scatter plots. Some simple statistics may be calculated from the data in a histogram object using methods in the corresponding class. The Lbins class defines the binning; it is combined with either Lhistbin objects to make Lhist1 or Lhist2 objects. The Ldata class is basically a std::vector of objects of the class Ldata::Point. An Ldata::Point is basically an (*x*, *y*) location and a pair (*dx*, *dy*). For a scatter plot, (*dx*, *dy*) are the error bars, and *dy* is used for any fits. For a vector (field) plot, (*dx*, *dy*) are the 2 components of the drawn vector.

Throughout the code, the convention is that Lbins objects have names that start with b, histograms have names that start with h, and GUI objects have name that start with p. For example, in divHisto.h, the lines  
  
 Lbins bDivCenter(nBinsHistoCenter, loHistoEdgeCenter,   
 hiHistoEdgeCenter);

Lhist1 hDivCentral (bDivCenter,  
 "{/Symbol \321\267}{/Palatino:Bold B} [T/m] -   
 central field map");  
  
define the bins for the histogram which will be filled with the distribution of ∇⋅*B* for the central field map. The peculiar string is the label for the histogram, which is used by the GUI object that will plot it. The GUI objects are generally constructed from their corresponding histograms. So,  
  
 Lfit1 pDivCentral(hDivCentral);

pDivCentral.logY = true;

pDivCentral.show();  
  
creates a GUI object, of the class Lfit1, out of the histogram hDivCentral, sets it’s vertical scale to be logarithmic, and asks it to show itself on the screen.

Given the large lag time for the show() methods of the GUI classes in the emphaticgpvm environment, you may want to just write the data out as a comma separated file.  
  
 hDivDecomp.CSVout(“SomeFileName”);  
  
should create a csv file with the obvious name. I’m pretty sure that should work. Don’t have “.csv” at the end of the filename.

Alternately, you might just want to convert to root. In that case, open a root file at the start of main(/\*…\*/) and go into a different subdirectory for each of the files in CodeGuts.

### About ra<T>

In C++ there is a template class called valarray. It is perhaps best thought of as a container, but is typically characterized as part of the numerics library. It corresponds to the fast, fixed-size arrays which might be used in linear algebra or finite element analysis on fixed grids. In addition to rapid indexing, valarray provides for slicing. If you have an *n*×*n* array, and you happen to just want to look at the *i*th column or *j*th row of that array, then you want a slice of the array.

ra<T> is a wrapper around std::valarray. It contains:

* The valarray itself.
* Sizes for up to 4 dimensions and constructors with size arguments. So ra<double> lookSee(3,4); will create a valarray named lookSee’s with space for 12 doubles, and those elements can be accessed with lookSee(i,j).
* Access methods, with optional bounds checking. In order to implement fast access without bounds checking, the symbol raBOUND must be undefined at the top of ra.h.
* Copy and assignment constructors.
* Setter/getter methods to change the offsets for the indexing. The default offset is 1, i.e. the first element of the ra<double> lookSee(3,4) is lookSee(1,1) – but with these methods, any set of up to 4 offsets is allowed.
* The operators operator+=(T), operator-=(T), operator+=(T) and operator+=(T), are provided. Operators to perform math operations on entire ra<T> instances are *not* provided.
* Methods to fill the valarray from a traditional C array or a C++ valarray are provided. If raBOUNDS is defined, these methods will check that the valarray in the ra<T> is large enough to fit the data being copied in
* Methods to find slices of the ra<T>. Here is an example of their use:

ra<double> lookSee(3,4);

ra<double> takeAslice(3);

lookSee.fillFromC(≪some 12 element C style array≫, 12);

const gslice knife = lookSee.sliceRA(3,lookSee.ALL);

takeAslice = lookSee[knife];

The ra<T> template does have 2 efficiency limitations. Firstly, move semantics have not been implemented as of yet, so operator= is a deep copy, and expensive for a large ra<T>. If you wish to avoid that problem, you will need a pointer to your ra<T>. That is not an issue for the problem at hand. Secondly, the creation of a new ra<T> requires dynamic memory allocation, and an algorithm which creates and destroys a lot of small ra<T>s will bog down. That is an issue for the problem at hand, as 1, 2 and 3 dimensional arrays of 3 elements each are heavily used in the quadratic interpolation code.

Accordingly, RA3.h is a bare bones C style version of an ra<double>(3), an ra<double>(3,3) and an ra<double>(3,3,3). Dot and cross operators are provided for the ra<double>(3) case. The quadratic interpolation in poly3int and the evaluation of the vector integrands in integrands.h rely upon RA3.h.

### About FieldMap

The field map itself is implemented as an ra<T> and a collection of methods to manipulate and study it. The ra<T> is actually a pointer to the ra, ra<AtPoint>\* where AtPoint is a structure containing an (*x*, *y*, *z*) location, the 3 components of the magnetic field at that location, and a flag to indicate distinguish a point where the magnetic field is not known from a point where the field is known to be zero.

This class is implemented using the constructor/getter pattern, in which the constructor accumulates all the important information and most of the other methods in the class permit access to that information.

The map can be accessed with either iterators or indices. An example of accessing someMap via iterators is

for (FieldMap::FieldMapItr itr=someMap.pBegin;  
   itr<someMap.pEnd;  
 ++itr) {  
 if (itr->valid) {  
 ≪do something with the AtPoint given by \*itr≫  
 }  
}

Indices have 0 offset. An example of accessing someMap via indices is

int nXbin = someMap.getNxBins();  
int nYbin = someMap.getNyBins();  
int nZbin = someMap.getNzBins();  
for (int iZ = 0; iZ <nZbin; ++iZ) {  
 for (int iZ = 0; iZ <nZbin; ++iZ) {  
 for (int iZ = 0; iZ <nZbin; ++iZ) {  
 FieldMapInd indy = {iX,iY,iZ};  
 if (someMap(indy).valid) {  
 ≪do something with the AtPoint given by  
 indy≫  
 }  
 }  
 }  
}

Methods exist to convert indices to iterators and back, and to find either the location or the magnetic field corresponding to a specific FieldMapInd.

In addition to the ra<AtPoint>\* which contains the field map data, the FieldMap class also has an ra<double> to give the offset of the map’s origin relative to the magnet’s origin. That is, the map came to us from AP-STD with an origin which is not the magnet coordinate system used by EMPHATIC, and the relationship between these needs to be encorporated in the FieldMap. The offset is set to zero in the constructor; this is correct for maps which are produced by this code. Only maps directly from AP-STD need a nonzero offset, and the determination of that offset is a complicated affair. The map mfCMSMMapStitched\_5mm\_longZ.txt is produced by the FieldMap class in a different code package, and it does have the correct offset incorporated for it; in main.cpp the offset is set to zero explicitly, to emphasize this.

The method FieldMap::insideMap will determine if a specified point in the magnet coordinate system is within the spatial extent of the map.

Slices of the map corresponding to all the map points along a given line in *x*, *y* or *z* are obtained with the sameYZ(FieldMapInd), sameXZ(FieldMapInd) and sameXY(FieldMapInd) methods.

There are methods to easily provide info for constructing bins for histograms.

There is are methods to compute the gradient and the divergence of a field at a given FieldMapInd.

The quadratic interpolation method is sufficiently complicated that the code to implement it is located in a separate file, interpolate.cpp. The full definition of the class is thus in 3 files.

## A step-by-step analysis

Here we will go through the whole process of making all the plots available to examine the field map data, and then to stitch together a single unified map.

The first step is to set up your working environment, and at this time, there is no simple, clear and concise way to do that. At this time, the emphaticgpvm nodes are running Alma9, which does support the tools you need, except for the main EMPHATIC software from github. Also, you may run an SL7 container. That will let you get your hands on the source code for this, and all the other EMPHATIC analyses.

Probably your best bet is to get the code from the MagneticField/MapMaking area of emphaticsoft using a container, and then get out of the container and just use the native Alma 9 environment for all that follows.

Here is how to check that you have all the required pieces:

1. ls -l  
   You should see 3 subdirectories, one named code, one named doc and one named other\_data. This document better be in the doc area! Also there is the .tgz file mentioned above in the doc area.
2. gnuplot --persist  
   You will need is gnuplot. After gnuplot is started, try test palette to make sure that plots will appear on your screen and then exit.
3. g++ -std=gnu++17  
   The C++17 option will be needed too. This command should fail by telling you that you have no input files. If it fails by telling you that -std=gnu++17 is an unrecognized command option, that is bad. Should that happen, you might need to source /cvmfs/larsoft.opensciencegrid.org/spack-packages/setup-env.sh, followed by spack load gcc.
4. (Possibly) you will want to make sure that your terminal is using UTF-8 encoding; at the check stage there are some lines sent to stdout which assume that.

Now you should be good to go.

1. cd code
2. ≪editor of your choice≫ Inputs.h  
   In Inputs.h, make sure dirName is the name of the directory where you are working. Also, set whichMagnetType = CMSM, as that is the magnet for which we have good data.
3. For our first set of plots, set showXYscan and showDivHist to true.
4. make; ./main.exe  
   Wait way too long to get 6 plots akin to Figures 1, 2 and 3 of EMPHATIC DocDB 1643. The binning parameters for the distributions of ∇⋅*B* are set in Inputs.h in the lines immediately following where showDivPlots is set. You will see some messages which can be ignored, which start “line 0:”; these can be ignored. Also at various times it proves to be impossible to rm some files in the /tmp area. Ignore that, too. Statistics regarding the distributions of ∇⋅*B* are sent to the screen as well.
5. WhichMap const showThis = central;  
   The enum named WhichMap is defined in CodeGuts/Funcs.h. The variable showThis is used for the next few plotting options, which work on only one of the 3 maps at a time. So,
6. Reset showXYscan and showDivHist to false and set showBxyzVsZ to true. Again, make; ./main.exe. This will make many plots, slowly, that are akin to Figure 4 of DocDB 1643. After each plot, hit the return key to wait around for another plot. When bored, hit some other key and then the return key.
7. The procedure to make plots akin to Figures 5 through 7, plus some other spiff plots, and optionally with the up and downstream maps, using showSlice\* and showIntegrals should now be evident.
8. The checkUp, checkCn and checkDn options (and later, the checkFull) tests are not strict. They consist of looking at each point in the data map, calling the quadratic interpolation at that point, and verifying that the two results are the same. You should get histograms that look an awful lot like a Dirac δ function.
9. fitUpstream = fitDownstream = plotDiffUp = plotDiffDown = true;  
   Now we do the fitting process to align the upstream and downstream fields to the central field. Physical metrology is needed of course to locate the central field to the magnet. The fitting code is not obvious to comprehend and is described in Appendix B, along with the code to make the plots describing how well maps overlap. *Do not fit any new data until you understand what happens in that code*!
10. makeSingleMap = true;  
    And again of course make; ./main.exe. Setting this true will force calls to the fitting process and will create a file, of name given by either CMSMoutMapOnDesktop or SABRoutMapOnDesktop, as set in Inputs.h. That file will have interpolated values on a grid with a spacing as defined by halfSpanX, HalfSpanY, lowestZout, highestZout, stepX, stepY and stepZ. Those are set based on the span of the actual map and allowing for quadratic extrapolation of ½ the grid spacing in the *x* and *y* directions. Extrapolation beyond that is unwise. The step size is currently set to 2 mm, and at this writing that seems to be a good level of detail for the tracking.The output file has a header incorporated as its first 9 lines, and that header has to be edited by hand to have the correct creation day and date, and also probably to have good descriptive commentary. The number of nodes will be the number of lines in the file, minus 9 lines for the header.
11. plotIntegral = true;  
    This will create Figures 8 – 10 of DOcDB 1643 and some of the numbers there. One must have the stitched map on hand of course.
12. plotStitchedBvxZ = true; or checkFull = true;  
    These provide the equivalent of Figure 4 or the interpolation checks for the stitched map. Again, one must have the stitched map on hand.

## Appendix A: The L++ man page:

L++(1) 1

**NAME**

**L++** – Leo Bellantoni's Private Library

**SYNOPSIS**

**(Build** **and** **run** **in** **Xcode)**

**DESCRIPTION**

**L++** is an Xcode bundle using the FreeBSD Unix functionality of

Darwin. It is written in C++ but does not use any Apple OS X

API, so as to facilitate porting to Linux or other \*nix

platforms. It consists of:

**•** main.cpp: The entry point code. Typically, one copies the

**L++** bundle, modifies the main.cpp file to create and

manipulate the objects of the **L++** library, and then builds

and runs the copy in Xcode. The default version of

main.cpp contains examples of usage of the **L++** classes and

some notes about good-vs-bad C++ usage.

**•** Links to the OS X Accelerate framework, which contains

LAPACK and BLAS. **L++** also links to libLHAPDF.dylib for

parton density functions and libxml2.tbd for xml

operations.

**•** An 'Inbox' of various old routines, some still in FORTRAN,

which have not been integrated, or for that matter, needed,

in **L++**

**•** Documentation. Some, anyway.

**•** **L++** itself

The folder "Brilliant" contains main.cpp, three header files, a

comma separated values file, and a file named stdout.rtf.

FittingFuncs.h is a collection of functions to which you might

fit some data; and of course you can write others.

integrationFuncs.h are examples of functions which might be

integrated in either the OS X Accelerate framework or with

VEGAS, which is implemented here as well.

Inputs.h is a place to enter the options needed for any

particular run of **L++** , such as an input file name or particular

setting of a basic parameter. Precompile lines such as #define

which are placed here will be available in scope of int main().

The Xcode build system usually makes the compilation of these

constants a matter of only a few seconds.

EMPHATIC\_B\_Field.csv is a comma separated file with some data

that is plotted in the baseline version of **L++** and stdout.rtf

is the stdout output of that same version, when run.

Within **L++** itself (as seen in Xcode as subfolders of the **L++**

folder) there are the following:

C-- Code to remedy shortcomings in C++, of which there are

a surprising number, given how feature-laden the

language is.

general

Header files which are used through out the package.

problems

Two classes for handling exceptions, warnings, or

informational messages. The complete list of error

messages is in a separate header file.

io A class and 6 header files to simplify text, file or

keyboard driven I/O.

when? A class for handling times, and a class for handling

dates.

ra A template to implement fast-access fixed size

multidimension arrays, with slicing capabilities using

raALL, the single member of the oneval class. Also,

classes that implement fairly high-level complex

matrices and vectors and operations such as inversion,

determinants, eigen-stuff, etc. Be aware that these

classes and templates carry their dimensionality inside

of them, and that requires that creation and

destruction call memory allocation and deallocation

system function calls. There is some overhead there.

functions

Some header files with simple but handy functions.

Also, code to handle the ubiquitous sum and difference

in quadrature operations that prevent the now

oft-forgotten round-off issues of these tasks, using

algorithms developed decades ago. permuter is a class

to handle index permutations; polyQint and polyMint are

forms of polynomial interpolation using LaGrange's

method. Version Q should be a little faster and

provides a not-very-useful error estimate; Version M

will work for up to 4 dimensions. Finally, here are

Bessel functions of the 1st and 2nd kind, plain and

modified.

functionals

plotter1d and plotter2d are classes which make quick

plots of functions of 1 or 2 doubles. A quick and

pretty-good numeric integration in one dimension is

implemented as a class template in integrator.h. This

integrates a function defined in the definition of a

class with a double operator(...) method (see

functional.h), so the integrand can be parameterized

with other data members in that class. The OS X

Accelerate framework provides what are probably better

algorithms, albeit without the ability to parametrize

easily; main.cpp has an example example of their use.

For multidimensional integrals, you would probably want

to use VEGAS, LePage's famed numerical integrator.

solver.h is a root finder for a function of a single

variable. Again, the function is to be provided as the

definition of a double operator(...) method. Fitting

is done in the fitter class, which has simplex,

gradient, and minimum of quadratic algorithms.

Finally, while fast Fourier transforms are ubiquitous,

slow ones are handy to confirm the FFT code you just

wrote, and also to handle cases where the dataset to be

transformed is not an integral power of 2 in length.

Hence, the SFT class.

random Classes to create pseudo-random number sequences. The

underlying generator, in ranunl.cpp is fairly simple

and not very bad.

lies Statistical calculations, some of which are specific to

particle physics. The stats class has functions that

commonly appear in statistics, including setting

uncertainties. Also here is a nice log-likelyhood

minimization quantity. fullim, poiprob and BLUE are

answers to common simple small-stats particle physics

problems; meansig, meancorrel and poisum are answers to

other common simple stats problems.

particles

Classes or code specifically related to particle

physics, including a 4-vector class, a port of the

RAMBO algorithm into C++, and a class that provides a

lightweight interface to the LHAPDF package.

histograms

Here are classes to implement ntuples, histograms,

profiles and datasets for fitting or plotting. The .cpp

files are in a subdirectory. Actually the Ltuple class

is really suboptimal, even in comparison to the rest of

the code here. The classes in this folder contain and

manipulate the data. For plotting, you need . . .

GUI These are classes which form a C++ interface to a pipe

into gnuplot for drawing those histograms & such on the

screen.

The default bundle contains breakpoints for less-well tested

parts of the code, as well as for catching exceptions as the

pitcher is in his windup.

**COPYRIGHT**

Most of the code here was taken from other packages or from

textbooks and modified for my particular personal use.

**Copyrights** **for** **general** **or** **widespread** **usage** **or** **distribution** **do**

**not** **exist!**

**FILES**

See above.

**BUGS**

Naaahh!

Max OS X - Darwin 17 Jan 2024 Max OS X - Darwin

## Appendix B: A guided tour through findOffs.h and plotOverlapDiff.h:

findOffs.h:

Start by looking at the last method given in the file, findOffsets(FieldMap\* cnMap, FieldMap\* fgMap). The arguments are bare pointers to a central and a “fringe” field map, i.e. either an upstream or a down stream map.

The first step is to loop, using FieldMap::FieldMapItrs, through the central map and making a vector, testItrs, of points in the map that overlap the fringe field map. It’s important to realize that once this list is made, it does not change, even if the size of the offset to the fringe map gets to be large enough to change some data points in the central map to be outside of the scope of the fringe map.

To set up the fit, an object of the class FringeChi2 is constructed. That class’ definition is the other method in this file. FringeChi2 is derived from the L++ class functional, which only means that it must have a double operator()(ra<double>\* X); that operator will return the χ2 value used to fit the offset of the fringe field relative to the central field.

FringeChi2::operator() start by putting X into the offset of the fringe field, and then loops over testItrs. For each of these data points in the central map, the fringe field map’s quadratically interpolated value, in Bfringe is found, or else we get !canInterp. Assuming we can interpolate, a χ2 value is found from the difference between the measurement of the central field and the interpolation of the fringe field. The uncertainty for the χ2 is found by multiplying the gradient of the field by an error for the mechanical positioning of the probe and adding this in quadrature to an uncertainty from the probe’s reading themselves. This uncertainty is computed separately for all 3 axes. The assumed probe and mechanical uncertainties are set in Inputs.h, as the CMSM(SABR)probe(mechanical)Err parameters. The value of fitVerbosity in Inputs.h has been set to 2 so that each call to this function will print out the offset being tried by the minimizer and also – critically – the number of contributions to the χ2, which is the number of points where the central map point can be interpolated in the fringe field map. If this number changes, you have a problem with changes in the number of points where the two maps overlap. *It is necessary to have a good initial value from mechanical observations on the measurement stand for the offset and to confirm that the overlap region contains the specified number of points and that no map points are close to the boundary.*

plotOverlapDiff.h:

The L++ template integrator.h will integrate any object with a double operator(double) signature, whether a simple function or an instance of a class with that method declared. Here, we do a 2 dimensional integral by creating an object of the class integralOverXY which has that operator, and that operator creates and calls an object of the class integrandOverY. The operator() of integrandOverY creates and calls an object of the class integrandOverX. The integrandOverX object has a double operator(double) signature which computes the 3 components of the difference between the interpolated values of the two fields and adds them in quadrature.

So, the integrandOverY performs a numeric integration over *x*, obtaining the ra<double> integral1D. This ra has 2 entries; the value of the numeric integral and the estimated uncertainty thereupon. The latter is discarded and the former is returned as a value for integralOverXY::operator(). That method is doing and integral over *y*. Again, the estimated uncertainty is discarded and the value of the integral is returned to line 44 of plotOverlap::plotOverlapDiff, where it too is discarded.

Why? Because – ha ha! The integralOverXY class contains a meansig object, which provides a mean, standard deviation, error on mean, maximum and minimum of a set of values. This meansig object is passed in a truly deplorable manner to the integrandOverX object which increments the meansig object at every point where the difference between the two field is evaluated. Then the statistics in the meansig object are put into the plots created and then shown by plotOverlap::plotOverLapDiff.

Feel free to rewrite this crappy code.