**HELMHOLTZ DECOMPOSITION CODE NOTES**

Leo Bellantoni

25-Jan-24

## Summary

This is the documentation for the code developed to do the Helmholtz decomposition and construct the magnetic field map for the first “China” or “Phase I” magnet of the EMPHATIC experiment.

As for the deeds of this magnet and the Helmholtz decomposition, are they not written in the DocDB of EMPHATIC, particularly entries 1608v1, 1768, 1643 and 1835?

[Summary 1](#_Toc157090278)

[The L++ framework 2](#_Toc157090279)

[The Helmholtz decomposition code 2](#_Toc157090280)

[About ra<T> 4](#_Toc157090281)

[About FieldMap 5](#_Toc157090282)

[About integrands.h and VEGAS 7](#_Toc157090283)

[A step-by-step decomposition 9](#_Toc157090284)

[Appendix: The L++ man page: 12](#_Toc157090285)

## 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, 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 the appendix.

In porting L++ to Linux, some modifications had to be made. The hidden files used by OS X’s GUI were deleted; the #include statements had to be changed because Xcode’s header search list isn’t quite there in the Makefile (although it could be built). Because the header files for CBLAS and LAPACKE were too hard to find, the eigensolution code in Lmat was commented out; as a result, the L++ fitter code to find a quadratic minimum to a χ2 function to determine had to be commented out; as a result of that, the fitting of a curve to one dimensional histograms or profiles in Lfit1 had to be commented out. No fitting is needed here anyway.

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

## The Helmholtz decomposition 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.
* mfMapMeasured\_5mm.txt – this is the merged, 5 mm grid file which incorporates the measured field from AP-STD.
* Probe\_calibration\_Mar2023.csv – this file contains the corrections needed to implement the calibration of the Hall probe provided by the manufacturer. At this time, the only constructor written for the FieldMap class requires that an instance of the ProbeCalib class be passed to it. Actually though, because mfMapMeasured\_5mm.txt already the calibration applied to it, the FieldMap constructor ignores the ProbeCalib anyway.
* ToDoFile.csv – this is a list of the (*x*, *y* , *z*) points[[1]](#footnote-1) at which a Helmholtz decomposition is to be done.
* Results.csv – what you want!
* ToDoAndDoneLists – this class handles the ToDoFile.csv and Results.csv files, maintains an internal std::deque of events for which a Helmholtz decomposition is to be done, and has a method for the construction of a new ToDoFile.csv file.
* ProbeCalib – this class of course holds the calibration for the probe, as described above.
* FieldMap and interpolate.cpp – these are the central class of the analysis, and will be described below.
* integrands.h and VEGAS - these are the functions being integrated and the implementation of the integration algorithm. They too, will be described below.
* ra<T> - of all of the parts to L++, understanding this template class is most important to the user. Again, detailed below.

These files are either in the top level directory if they are inputs or outputs to the code and in the CodeGuts directory if they are part of the Helmholtz or field map algorithms themselves. The L++ template ra<T> is in the ra directory and the files for VEGAS are in the functionals directory.

### 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).
* Sccess 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, 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 mfMeasured\_5mm.txt actually is produced by an earlier version of the FieldMap class, 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 a method to compute 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.

### About integrands.h and VEGAS

The VEGAS algorithm is implemented in L++ as the 4 files VEGAS.h, VEGAS.cpp, VEGAS\_core.h and VEGAS\_util.cpp. The implementation was taken from the Gnu Scientific Library and extensively checked both against certain analytic results, the original documentation by G. P. Lepage and of course in the cross checks for this analysis, as given in EMPHATIC DocDB 1835.

Each call to VEGAS\_integrate goes through a number of internal iterations at estimating the value of the integral, and the consistency of the resulting estimates, which is expressed as a χ2 number is a guide as to whether or not the algorithm has converged. The code also provides a running estimate of the error in the integral and that two is a useful guide as to how well the algorithm has converged. The L++ implementation is set up so that repeat calls to VEGAS\_integrate will pick up where the last call left off.

VEGAS integrates a thing called a VEGAS\_function, which consists of a function, the dimensionality of the integral, and a parameter set.

In L++ there is not a real thread pool which maintains a collection of threads which can be started and stopped. Rather, there are the templates C--/Threadpuddle.h and C--/ThreadHandler.h which use the C++ std::async mechanism to create threads and std::future<T>.get() to obtain the result. This requires that the function to be evaluated in the created thread has to have one and only one argument.

With this information, it is straightforward to understand integrands.h.

integrationParams is the structure to pass parameters to VEGAS; these include the definition of the integration surface, the location where the field is to be evaluated and such as that. Arguments is the structure used to pass function evaluation arguments to std::async.

vectorIntegrandFunc is the calculation of the vector quantity to be integrated. Because VEGAS integrates only real valued functions, one of the parameters passed to vectorIntegrandFunc is component, which specifies if the *x*, *y* or *z* component of the vector integrand is to be returned.

doIntegration is the interface to VEGAS. This is where the integration convergence criteria are examined and a repeat call to VEGAS\_integrate made if necessary.

fullHelmholtz is the function which is passed to std::async by ThreadPuddle. It constructs 9 integrationParameters corresponding to the 3 components of the 3 parts of the cylindrical integration surface. Then 9 VEGAS\_functions are created corresponding to the vectorIntegrandFunc and one of the 9 parameter sets. The doIntegration function is called, again 9 times and then the final result is assembled. The ra<double>(9)which is returned contains the (*x*, *y*, *z*) point of evaluation, the 3 components of the result of the Helmholtz decomposition and then the 3 components of the magnetic field with the ad hoc correction included.

## A step-by-step decomposition

Here we will go through the whole process of creating a Helmhotz decomposition map, albeit of small size, and in the process, comprehending (and perhaps editing) what is in main.cpp.

After setting up the EMPHATIC software,

1. gnuplot --persist  
   You will need is gnuplot, which should be set up automatically on the emphaticgpvm cluster. After gnuplot is started, try test palette to make sure that plots will appear on your screen and then exit.
2. 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.
3. (Optionally) setup forge\_tools  
   This is where you get ddt from. It’s a handy tool. Make it your friend.
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.
5. cd ≪Helmholtz directory≫  
   The directory where this code is going to live hasn’t been determined as of this writing. But you can find it! It has a doc subdirectory, and the code directory; cd into the latter.
6. ≪editor of your choice≫ Inputs.h main.cpp  
   We won’t actually change main.cpp, but we will look at it.
7. In Inputs.h, make sure dirName is the name of the directory where you are working. Also, set today = makeFreshList, as that will be the first step in the process.
8. The only input to the creation of a fresh list is the stepsize of the grid in the list. This is set by stepX, stepY and stepZ. They are set to 2 mm steps in all 3 directions, but for brevity in this exercise, try stepX = stepY = 0.010 and stepZ = 0.050.
9. make  
   Or make DEBUG if you want to run the code under gdb or ddt. I you do that, there will be a warning message at ThreadPuddle.h:29 which can be ignored.
10. ./main.exe  
    The code at line 96 of main.cpp will run quickly. You will get a message saying that we do have bounds checking in ra<T>, one giving the dimensions of the cylinder used as the surface of integration and then a notice that you have created a fresh list of points. It is in FreshList.csv which is of course human readable. At this point, there is nothing in the ToDoFile.csv so just
11. cp FreshList.csv ToDoList.csv.
12. Edit Inputs.h again. You probably want to change stepX, stepY and stepZ back to their original values, as you are done with them for now anyway. Set today = crunchDecomposition.
13. And now, consider the interesting parameter numberWorkThreads. This is the number of individual Helmholtz decomposition evaluations happening at one time. The correct value depends on which machine you are running on; for emphaticgpvm02, which has 4 cores, setting this value to 4 will permit the kernel to assign all 4 cores to this process. If you set this parameter to e.g. 3, only 3 cores max can be used – top will reveal a %CPU under 300%. Of course, requesting 4 threads does not mean that the kernel will give you 4 cores. The comments in Inputs.h provide extensive information about how to set this parameter on an Apple M1 chip; for other machines, you will need to work out for yourself what is the best number of threads to request.
14. The other parameters for doing the numerical integration, which all begin with VEGAS\_, are probably best left alone.
15. make; ./main.exe  
    The code at lines 134 – 150 of main.cpp create an std::deque of “chores” which consist of a function call paired to an argument list. As each chore is added, they are added to the collection of std::async objects and execution of them starts; the maximum number of such executing threads is given by numberWorkThreads. As threads finish their evaluations and output their results, they are replaced by new threads assigned to new chores. The output of all of this goes to Results.csv.If you wish, you may examine Results.csv using Excel.
16. Edit Inputs.h again. Set today = checkResults. This will make some plots from the computed fields, and will also create files in the format used by EMPHATIC reconstruction software. The relevant parameter in Inputs.h is scaleOrigBy. The original 5 mm spaced grid, for the given radius and span in *z*, contains 7840 points. The distributions for this grid will be scaled by this parameter to match the increased area of plots from the map just made. You might also want to change checkFileName, which is the name of the results csv file that is to be checked. The 2 entries in outputMapOnDesktop are the places where
17. Examine lines 248 & onwards in main.cpp. This is where the plots are defined and filled. To change the plots, you will need to understand the following L++ classes:
    1. Lbins defines the bins of an axis; arguments to the constructor are number of bins, low edge, high edge. The current binning is good, if not optimal, for a 2 mm spaced grid with the cylindrical dimensions given by Inputs.h.
    2. Lhist1 and Lhist2 are histogram classes. The first argument or two are the bins for the *x* and perhaps *y* axis; the last argument, which is optional, is a string that will be the title on the plot. There are pdf files with information about how gnuplot handles special characters in the Documentation directory, should you want that.
    3. Lfit1 and Lzcol are 1 and 2 degree plots constructed from Lhist1 and Lhist2 histograms.
18. Again, make; ./main.exe  
    Unfortunately, the use of gnuplot via a pipe created with the libc function popen() does not appear to be too stable on the emphaticgpvm cluster, and the version of gnuplot which is installed is not the most recent one. Accordingly, the plots may not make a lot of sense; it seems like labels get put on the wrong plots, for example. But the distributions of the ∇⋅B histograms are output as csv files in the working directory, as are the Helmholtz-decomposed map files, with and without the ad hoc correction, as txt files, in the same format as used by EMPHATIC reconstruction software.
19. You are done!

## Appendix: 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

1. EMPHATIC DocDB 1643 explains the coordinate system used for magnet work. Units here are Tesla and meters unless otherwise specified. [↑](#footnote-ref-1)