cMag: A *C* Version of the CLAS12 magnetic field package

D. Heddle
Christopher Newport University
david.heddle@cnu.edu

May 23, 2020

Abstract

The standard CLAS12 magnetic field that reads and interpolates the binary field maps for the solenoid and torus was written in JAVA. The package described here reproduces the same functionality in *C*. That's *C*, not *C++*, [1],[2] but of course it can used in a *C++* program. The most important feature is that it reads the same fieldmap files as the JAVA version. The code has been tested on OSX 10.15.4, ubuntu linux 20.04, and one other operating system. [666]

[1]The reason should be obvious. *C* is the most beautiful programming language ever created while, remakably, *C++* is the most hideous. This is not a matter of opinion.

[2]Pointer arithmetic, fine-grained control over memory (what could go wrong?), a preprocessor that allows you to hide critical code in inpenetrable macros, and a type-unsafe compiler that looks at your line of code that equates an integer pointer to an array of strings and says: *"Cool, that works for me! I'm sure you know what you are doing."* I mean, how can you not love it!

[666]That would be Windows 10.

Contents

1 Introduction

The magnetic field package used by *ced* and by the CLAS12 reconstruction was written in JAVA. The binary field map files used by the magnetic field package were written in JAVA[3]. However, the CLAS12 simulation, GEMC, is written in *C++* and reads ascii field map files. In spite of great effort and testing, there is always a nagging suspicion that the simulation and reconstruction are using slightly different fields. This package, *cMag*, was commisioned to solve that proble, so that GEMC could read the binary maps. However, *cMag* goes beyond simply reading the maps, it also provides the same trilinear interpolation access to the fields that the JAVA package uses. This may be of use to other *C* and *C++* CLAS12 developments.

2 Where do I get it?

2.1 The Code

Like everything else that isn't available on *Amazon*, the *cMag* distribution is available on github at:

`https://github.com/heddle/cmag`.

2.2 The Field Maps

An exception to the rule stated above, the field maps are not available on either *Amazon* or github. The field map files are not part of the `cMag` distribution [4]. They can be downloaded from here:

`https://clasweb.jlab.org/clas12offline/magfield/`.

3 File Format

Provided mostly for completeness, and demonstrating the raw power of LaTeX, the fieldmap file format document has been inserted starting on the next page. If that doesn't work, the document is also included in the `docs` directory of the *cMag* distribution. [5]

[3] That's relevant, because JAVA specified that data be stored in big endian ordering on all platforms independent of architecture, while most of the machines we use in CLAS are little endian.

[4] This is because some people are overly sensitive about having gigabytes of field map data stored in every CLAS12-related repository.

[5] As, self-rerentially, this document is, referring to the location where it is stored at the location where it is stored.

Magnetic Field Binary File Format
Version 3
April 24, 2018

David Heddle
Christopher Newport University

This describes the binary format used by *ced* and also the general *magfield* package.

The binary file format contains a header of twenty 32-bit words. (The 80 bytes for this header are in the noise when it comes file size.) The header format is:

(int) 0xced (decimal: 3309) magic number—to check for byte swapping
(int) Grid Coordinate System (0 = cylindrical, 1 = Cartesian)
(int) Field Coordinate System (0 = cylindrical, 1 = Cartesian)
(int) Length units (0 = cm, 1 = m)
(int) Angular units (0 = decimal degrees, 1 = radians)
(int) Field units (0 = kG, 1 = G, 2 = T)
(float) q_1 min (min value of slowest varying coordinate)
(float) q_1 max (max value of slowest varying coordinate)
(int) N_{q1} number of points (equally spaced) in q_1 direction including ends
(float) q_2 min (min value of medium varying coordinate)
(float) q_2 max (max value of medium varying coordinate)
(int) N_{q2} number of points (equally spaced) in q_2 direction including ends
(float) q_3 min (min value of fastest varying coordinate)
(float) q_3 max (max value of fastest varying coordinate)
(int) N_{q3} number of points (equally spaced) in q_3 direction including ends
~~Reserved 1~~ High word of creation date (unix time)
~~Reserved 2~~ Low word of creation date (unix time)
Reserved 3
Reserved 4
Reserved 5

The magic number, which should have the hex value ced (i.e. 0xced), is important. The CLAS magnetic field maps are produced by JAVA code which (sensibly) enforces the use of network ordering (big endian) independent of architecture. However the machines we use in CLAS tend to be little endian. If the code reading the maps is also in JAVA, it doesn't matter. If the code reading the maps is in *C* or *C++*, byte swapping will likely be required.

As you see, there used to be five reserved 32-bit slots in the header. Two of them have been requisitioned to store the creation date of the field map file, which is a 64-bit (long) quantity. To get the creation date, the long has to be reassembled from its two pieces and then, using some sort of language supplied time function, converted into a meaningful string. The details are left as an exercise.

The only ambiguity is the meaning of the triplet $\{q_1, q_2, q_3\}$ For cylindrical coordinates, the triplet means $\{\phi, r, z\}$. It seems most natural that for Cartesian coordinates the triplet maps to: $\{x, y, z\}$. Thus, for a Cartesian field map, *x* would be the outer, slowest-varying grid component.

The total number of field points will be: $N = N_1 \times N_2 \times N_3$ (we will store floats, not doubles)). Each point requires three four-byte quantities. The total size of the binary file will be $80 + 3 \times 4 \times N$.

Noting that the number of points always includes the endpoints, the step size in direction *i* is $(q_{imax} - q_{imin})/(N_i - 1)$

In version 3, two of the reserved words have been allocated to store the creation date in unix time. The remaining reserved fields are available to be used in some manner to be defined later.

The field follows the header, in repeating triplets:

B1
B2
B3

The first three entries correspond to the field components for the first grid point, the next three for the second grid point, etc. The ordering, for consistency, should be:

$\{B_x, B_y, B_z\}$ if the field is Cartesian
$\{B_\phi, B_r, B_z\}$ if the field is Cylindrical

Example

For the binary version of the original torus map (before we encoded creation date) we have for the header:

0xced
0 (grid is cylindrical)
1 (field is Cartesian)
0 (units: cm)
0 (units: decimal degrees)
0 (units: kG)
0.0 (ϕmin)
30.0 (ϕmax, degrees)
121 (N_ϕ)
0.0 (r_{min})
500.0 (r_{max}, cm)
251 (N_r)
100.0 (z_{min}, cm)
600.0 (z_{max}, cm)
251 (N_z)
0 (Reserved 1)
0 (Reserved 2)
0 (Reserved 3)
0 (Reserved 4)
0 (Reserved 5)

Thus, the three step sizes are:

$$\Delta\phi = (30\text{-}0)/(121\text{-}1) = 0.25°$$
$$\Delta r = (500\text{-}0)/(251\text{-}1) = 2 \text{ cm}$$
$$\Delta z = (600\text{-}100)/(251\text{-}1) = 2 \text{ cm}$$

Recalling the header is 80 bytes, the total size of the binary is (had better be):

$$80 + 3\times4\times121\times251\times251 = 91{,}477{,}532 \text{ bytes.}$$

4 Building

After cloning the *cMag* repository, simply work your way down to the `src` folder where you will find a `Makefile`. Now, I have not written a makefile since CLAS was a 6 GeV toddler, but I do seem to recall that they are always very portable and never cause any grief. So I am comfortable that simply typing

```
$make
```

will work on any platform.

If it worked, you should now have toplevel `bin` and `lib` directories. Inside of `bin` should be an executable, `cMagTest`. Inside of `lib` should be the static library `libcMag.a`. Use that library, and the include files in the `includes` directory, to add the cMag functionality to your program.

4.1 Unit Testing

Assuming the build worked (and why shouldn't it?) the first thing you should do is run `bin/cMagTest` and see if it produces happy output (it does unit testing.) It will produce a *lot* of output which you may or may not find interesting. What you really care about is that `bin/cMagTest` terminates with the console print:

```
Program ran sucessfully.
```

If one of the unit test fails it will say, well, something else, depending on which test failed first.

5 Usage

Assuming the build worked, and the testing was successful, you are ready to use the package. We will not discuss how to link it; you surely know how. We will discuss how to *use* it after it has been successfuly linked. Here we describe only the"public" functions, i.e. the ones you will likely use. Of course *C*, being a highly democratic language, does not hide anything, so there are many more functions available, the functions that the"public" functions call upon. These functions are accessible it you seek to cause mischief.

We will begin with the first step, the initialization, which is the step that will most often go wrong. If you make it through the initialization, everything else should be smooth sailing.

5.1 Initialization