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

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

May 26, 2020

Abstract

The standard CLAS12 magnetic field package 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 field map 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, remarkably, *C++* is the most hideous. This is not a matter of opinion.

[2]Pointer arithmetic, fine-grained and absolute control over memory (what could go wrong?), a preprocessor that allows you to hide critical code in impenetrable 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 commissioned to solve that problem, 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/.

In Appendix B of this document you will find a description of the format of the field map files.

3 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 top-level `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.

3.1 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.)

But wait just a moment. Running `cMagTest` is the *first* thing you should do, which every programmer knows is the second thing you should do. The *zeroth* thing you should do, obviously first, i.e., before the first thing, is to make sure you have bonafide CLAS12 magnetic fields. As mentioned earlier, they can be downloaded from here:

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

[3] That's relevant, because JAVA sensibly decreed that data be stored in network format (which is big endian byte 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.

Of the magnetic fields you will find there, the two that `cMagTest` requires to run its tests are:

```
Symm_solenoid_r601_phi1_z1201_13June2018.dat

Symm_torus_r2501_phi16_z251_24Apr2018.dat
```

While the field map data directory is not hardwired into `cMagTest` (more about that anon) these two fields it tests itself upon are.

Let's suppose your username is `yomama` and you have downloaded the magnetic fields (including but not limited to the two maps mentioned above) to the directory `/Users/yomama/data/fieldmaps`. You pass that information to `cMagTest` as the one and only command line argument it processes. That is, you type:

```
$cMagTest /Users/yomama/data/fieldmaps
```

If you do not provide a directory as a command line argument, `cMagTest` will try one and only one place: `$(HOME)/magfield`. So you can put the field map files there and dispense with the command line argument.

While running, `cMagTest` will produce a *lot* of output which you may or may not find interesting. What you really care about is that `cMagTest` terminates[5] with the console print:

```
Program ran successfully.
```

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

4 Usage

Assuming the build worked, and the testing was successful, you are ready to use the package. We will not discuss how to link `/lib/libMag.a`; you surely know how. We will discuss how to *use* it after it has been successfully linked. Here we describe only the "public" functions, i.e. the ones you will likely use. [6] The complete API is provided in Appendix A.

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.

4.1 Initialization

Initialization involves successfully converting the location of the field map files (their paths) into `MagneticFieldPtr` objects, presumably one for the CLAS12 torus, and one for the CLAS12 solenoid. Once you have the valid pointers you have everything. In particular you can then ask for the field at any location.

Below we will assume that you are initializing one torus field and one solenoid field. You do not have to initialize both; if you just need one or the other then initialize just one or the other. [7].

This would be a typical initialization code snippet:

```
MagneticFieldPtr torus = initializeTorus(torusPath);
MagneticFieldPtr solenoid = initializeSolenoid(solenoidPath);

if (torus == NULL}  {
    //do something to handle a failure
}
if (solenoid == NULL}  {
    //do something to handle a failure
```

[5] Depending on the OS, it may be the last line of output or the penultimate line, the latter being the case when the OS obligingly prints: `Process finished with exit code 0`.

[6] Of course *C*, being a highly democratic and progressive language, does not hide anything, so there is really no elitist distinction between "public" and "private". In *C* such "binary" adjectives are discouraged. In short, there are many more functions available, the functions that the"public" functions call upon. These functions are accessible if you seek to cause mischief.

[7] In fact, you could initialize two tori and three solenoids. And you do not have to initialize *any* fields, but in that case we would have to wonder why you bothered to link `/libMag.a`.

```
}
```

where `torusPath` and `solenoidPath` are strings, each containing the full path to the maps you want to load. Or maybe not. It is permissible to pass `NULL` as the path argument. More about that is a second.

How do you know it it worked? Well, there should be some error prints if an initialization failed. But the programmatic test is whether the returned points are `NULL`.

Don't even ask what happens if you give `initializeTorus` a solenoid map, and `initializeSolenoid` a torus map. [8]

Another indication that it worked is that `cMag` will print out a summary of each field that was initialized. You should look for those summaries. For example, here is a summary of the solenoid: [9]

```
=====================================
SOLENOID: [/Users/heddle/magfield/Symm_solenoid_r601_phi1_z1201_13June2018.dat]
Created: Wed Jun 13 11:28:25 2018

Symmetric: true
scale factor: 1.00
phi min:    0.0  max:  360.0  Np:    1  delta:    inf
rho min:    0.0  max:  300.0  Np:  601  delta:    0.5
  z min: -300.0  max:  300.0  Np: 1201  delta:    0.5
num field values: 721801
grid cs: cylindrical
field cs: cylindrical
length unit: cylindrical
angular unit: degrees
field unit: kG
max field at index: 102625
max field magnitude: 65.832903  kG
max field vector(0.00000  , -7.56064 , 65.39731 ), magnitude:     65.83290
max field location (phi, rho, z) = (0.00  , 42.50 , -30.00)
avg field magnitude: 3.082540   kG
```

4.1.1 Environment Variables

So, what's this about passing `NULL` for a path to the initialization functions? In that case the initialization will reluctantly turn to environment variables: `initializeTorus` will first try a path obtained from the environment variable `COAT_MAGFIELD_TORUSMAP`. If that fails, it will try `TORUSMAP`. If that fails, it will give up the ghost, as far as initializing the torus is concerned. Similarly `initializeSolenoid` will first try the environment variable `COAT_MAGFIELD_SOLENOIDMAP`. If that fails, it will try `SOLENOIDMAP`.

4.2 Settings

How much control does the user have over what's happening under the hood? Not much. One global (i.e., it applies to all fields) option that is available is the *algorithm* (for obtaining field values) setting. The user can set it to `INTERPOLATION` or `NEAREST_NEIGHBOR`. The default is `INTERPOLATION`.

We don't think there is ever a need to switch it to `NEAREST_NEIGHBOR`, but should you want to, just call:

`setAlgorithm(NEAREST_NEIGHBOR).`

After you get bored with that, set it back via:

`setAlgorithm(INTERPOLATION).`

[8] Okay, since you didn't ask, I'll tell you. It's really bad. If you mismatch the calls, the secure CLAS password that we have used since the previous millennium for everything critical will be changed to `äçäÐğÆ` and nothing will work again. Ever. Okay really, nothing will happen except clarity will be sacrificed. The functions `initializeTorus` and `initializeSolenoid` are just wrappers to a single function that reads a field map. So all you will have achieved is obfuscation, which may have been your intent.

[9] The delta of ∞ for the ϕ grid of the solenoid field is a feature, not a bug.

As for field-by-field settting, each magnetic field has a `scale`, which defaults to 1. And each magnetic field has "misplacement" shifts `shiftX`, `shiftY`, and `shiftZ`, each of which defaults to 0 (units are cm). Thus you may want to do something immediate such as:

```
torusField->scale = -1;
```

Since that is often the case. [10]

4.3 Obtaining Field Values

Here we are: the meat and potatoes section. Everything has built with nary a glitch, all the unit tests have passed, and the field map files are downloaded, and the library `libCMag.a` is linked in. [11]

4.4 Miscellany

In case you'd like to know how the formatted creation date is obtained from the high and low words in the header, it's like this:

```
static char *getCreationDate(FieldMapHeaderPtr headerPtr) {
     int high = headerPtr->cdHigh;
     int low = headerPtr->cdLow;

//the divide by 1000 below is because the JAVA creation time
//(which was used in creating the maps) is in nS.

    long dlow = low & 0x00000000ffffffffL;
    time_t utime = (((long) high << 32) | (dlow & 0xffffffffL)) / 1000;
return ctime(&utime);
```

[10] We agonized over whether to make the default torus scaling -1, and finally chose the option we believe is most consistent with the *C* zeitgeist.

[11] Again, we will not comment on the link process, which for complex codes (not `cMag` which is embarrassingly simple, but for whatever is attempting to link `libCMag.a`, –which is likely to be complex beyond our ability to comprehend) generally leads to much weeping and gnashing of teeth. But just one note: `libCMag.a` does depend on the ubiquitous *C* math library, `libm.a`. No doubt your code already links that with a dash of `-lm`, but for full disclosure we are putting the dependency down on paper.

A Programmer's API

B Field Map 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. [12]

[12] 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):

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

END OF DOCUMENT