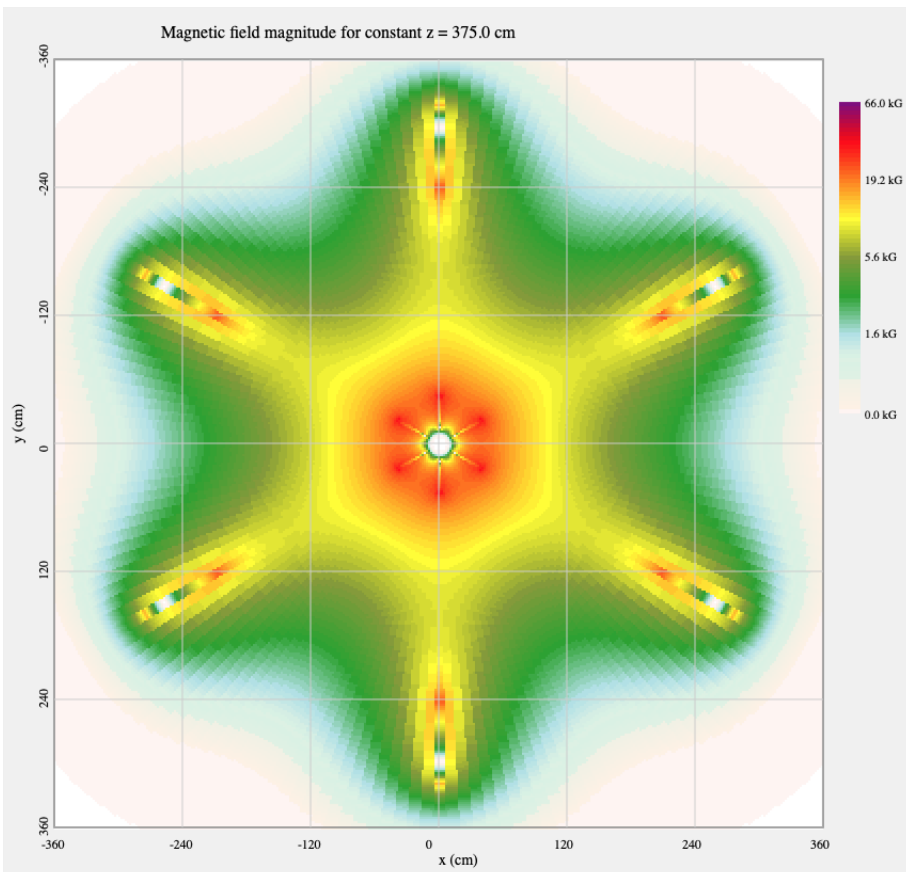
cMag: A *C* Version of the CLAS12 magnetic field package

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

June 1, 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]



A `cMag` plot of the torus field at a fixed value of z = 375 cm.

[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 tri-linear 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/`.

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

```
Symm_solenoid_r601_phi1_z1201_13June2018.dat
Symm_torus_r2501_phi16_z251_24Apr2018.dat
Full_torus_r251_phi181_z251_03March2020.dat
```

[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.

While the field map data directory is not hardwired into `cMagTest` (more about that anon) these three fields it tests itself upon are. They are, at the time of this writing, the most recent maps of the solenoid, the torus with assumed 12-fold symmetry, and the full torus with no assumed symmetry.

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
}
```

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`.

[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`.

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
numColors 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)`.

As for field-by-field setttting, 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]

---

[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 **äçäÐ ğ Æ** 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.

[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.

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]

There are two methods for obtaining the field values once have successfully read the maps. They are:

```
/**
 * Obtain the value of the field by tri-linear interpolation or nearest neighbor,
 * depending on settings.
 * @param fieldValuePtr should be a valid pointer to a FieldValue. Upon
 * return it will hold the value of the field in kG, in Cartesian components
 * Bx, By, BZ, regardless of the field coordinate system of the map.
 * @param x the x coordinate in cm.
 * @param y the y coordinate in cm.
 * @param z the z coordinate in cm.
 * @param fieldPtr a pointer to the field map.
 */
void getFieldValue(FieldValuePtr fieldValuePtr,
                   double x,
                   double y,
                   double z,
                   MagneticFieldPtr fieldPtr)

/**
 * Obtain the combined value of two fields. The field
 * is obtained by tri-linear interpolation or nearest neighbor, depending on settings.
 * @param fieldValuePtr should be a valid pointer to a FieldValue. Upon
 * return it will hold the value of the field, in kG, in Cartesian components
 * Bx, By, BZ, regardless of the field coordinate system of the maps,
 * obtained from all the field maps that it is given in the variable length
 * argument list. For example, if torus and solenoid point to fields,
 * then one can obtain the combined field at (x, y, z) by calling
 * getCompositeFieldValue(fieldVal, x, y, z, torus, solenoid).
 * @param x the x coordinate in cm.
 * @param y the y coordinate in cm.
 * @param z the z coordinate in cm.
 * @param field1 the first field.
 * @param field2 the second field.
 */
void getCompositeFieldValue(FieldValuePtr fieldValuePtr,
                            double x,
                            double y,
                            double z,
                            MagneticFieldPtr field1,
                            MagneticFieldPtr field2)
```

The first method is for obtaining the value of a single map, the second for combining the values of two maps. In all cases the length units are cm and the field values are kG. The result is placed in the `FieldValuePtr` structure pointed to by `fieldValuePtr`. That structure has three float values, texttttb1, texttttb2, and texttttb3 corresponding to texttttB_x, texttttB_y, and texttttB_z. [12]

So before calling these functions you probably want to do something like:

```
FieldValuePtr fieldValuePtr = (FieldValuePtr) malloc (sizeof(FieldValue));
```

[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.

[12] There is a reason why we didn't name them texttttbx, texttttby, and texttttbz. It's because sometimes in the innards of the code they correspond to cylindrical components.

and after the call you access the components by:

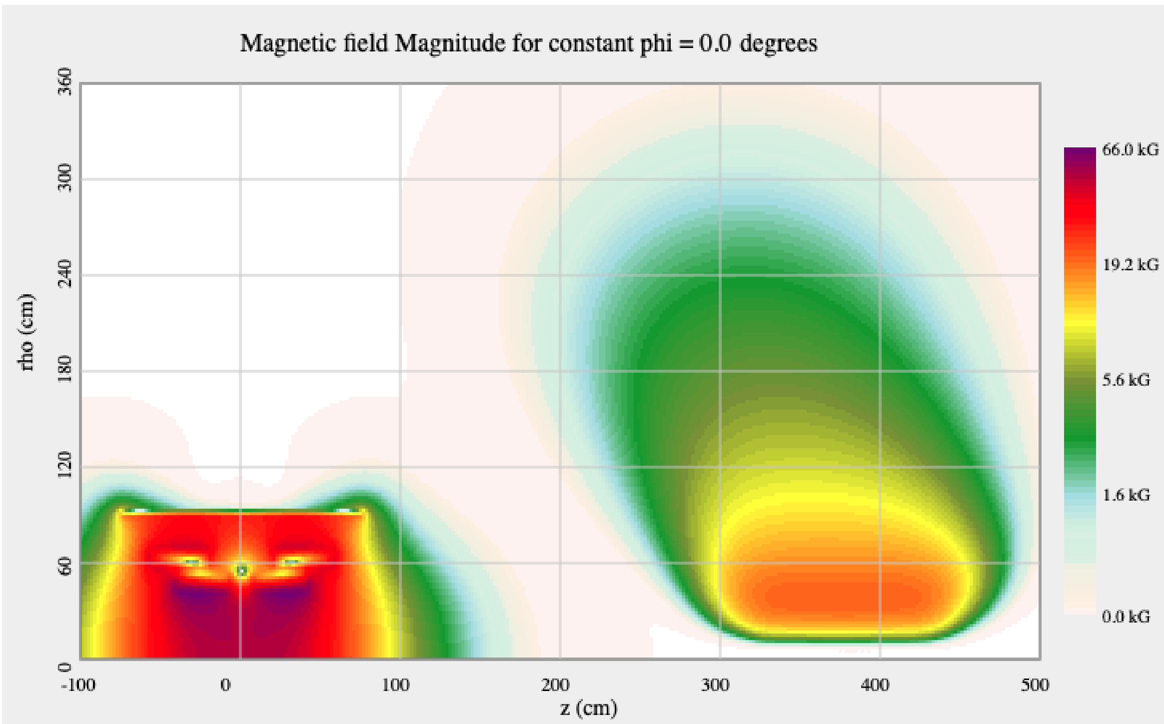
```
float bx = fieldValuePtr->b1;
float by = fieldValuePtr->b2;
float bz = fieldValuePtr->b3;
```

4.4 Miscellany

4.4.1 Seeing is Believing

I don't know about you, but I don't believe anything works unless I see it. So cMag comes with the ability to make some SVG images of the field. [13] Seeing that the images look reasonable is the best unit test. Although given the plots only show magnitude and not components, the components could be mixed up from a bad rotation or have the wrong signs. I truly hate when that happens.

Here is the canonical slice through the midplane of sector 1:



A cMag plot of the torus and solenoid in the midplane of sector 1..

Here are the current available methods for creating images:

```
/**
 * Create an SVG image of the fields at a fixed value of z.
 * @param path the path to the svg file.
 * @param z the fixed value of z in cm.
 * @param fieldPtr torus the torus field (can be NULL).
 * @param fieldPtr torus the solenoid field (can be NULL).
 */

void createSVGImageFixedZ(char *path, double z, MagneticFieldPtr torus, MagneticFieldPtr solenoid)


/**
 * Create an SVG image of the fields at a fixed value of phi.
```

[13] It was an easy choice to go SVG rather than jpeg or png or some other format. SVG files are xml, so producing them is simply writing text files, rather than adding jpeg or png libraries that will result in you build procedure being a house O' cards. In addition, someone else already wrote exactly the minimal SVG code thet we need, in *C* available at https://github.com/CodeDrome/svg-library-c. Game, set, match, point. Okay, it's not all good news, the svg files are fairly big, but I don't care.

```
 * @param path the path to the svg file.
 * @param phi the fixed value of phi in degrees. For the canonical
 * sector 1 midplane, use phi = 0;
 * @param fieldPtr torus the torus field (can be NULL).
 * @param fieldPtr torus the solenoid field (can be NULL).
 */

void createSVGImageFixedPhi(char *path, double phi, MagneticFieldPtr torus, MagneticFieldPtr solenoid)
```

4.4.2 Make a Date

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);
```

A Programmer's API

This appendix contains, starting on the next page, the Doxygen generated API for the `cMag` package. Because it is an inserted pdf, it conatins its own pagenumbers. Sorry about that. Also, in listing files it prepends the path from the machine I used to generate the documentation. That's silly and I'm guessing there is some Doxygen consfiguration setting to stop that–but I am a Doxygen noob and have not had time to investigation. [14]

[14] Compared to Javadocs, Doxygen is pretty awful. Something like `Javadocs:Doxygen :: A Nice Cold Beer:Root Canal`. Just saying.

cMag

0.5

Generated by Doxygen 1.8.18

Chapter 1

File Index

1.1 File List

Here is a list of all files with brief descriptions:

Chapter 2

File Documentation

2.1 /Users/davidheddle/cmag/src/magfield.c File Reference

```
#include "magfield.h"
#include "magfieldutil.h"
#include "munittest.h"
#include "testdata.h"
#include <stdlib.h>
#include <math.h>
```

Functions

- void setAlgorithm (enum Algorithm algorithm)
- bool containsCylindrical (MagneticFieldPtr fieldPtr, double rho, double z)
- bool containsCartesian (MagneticFieldPtr fieldPtr, double x, double y, double z)
- void resetCell3D (Cell3DPtr cell3DPtr, double phi, double rho, double z)
- void resetCell2D (Cell2DPtr cell2DPtr, double rho, double z)
- void getFieldValue (FieldValuePtr fieldValuePtr, double x, double y, double z, MagneticFieldPtr fieldPtr)
- void getCompositeFieldValue (FieldValuePtr fieldValuePtr, double x, double y, double z, MagneticFieldPtr field1, MagneticFieldPtr field2)
- int getCompositeIndex (MagneticFieldPtr fieldPtr, int n1, int n2, int n3)
- void invertCompositeIndex (MagneticFieldPtr fieldPtr, int index, int ∗phiIndex, int ∗rhoIndex, int ∗zIndex)
- void getCoordinateIndices (MagneticFieldPtr fieldPtr, double phi, double rho, double z, int ∗nPhi, int ∗nRho, int ∗nZ)
- char ∗ nearestNeighborUnitTest ()
- char ∗ containsUnitTest ()
- char ∗ compositeIndexUnitTest ()
- FieldValuePtr getFieldAtIndex (MagneticFieldPtr fieldPtr, int compositeIndex)

Variables

- MagneticFieldPtr testFieldPtr
- enum Algorithm _algorithm = INTERPOLATION

2.1.1 Function Documentation

2.1.1.1 compositeIndexUnitTest()

```
char* compositeIndexUnitTest ( )
```

A unit test for the composite indexing

Returns

an error message if the test fails, or NULL if it passes.

2.1.1.2 containsCartesian()

```
bool containsCartesian (
            MagneticFieldPtr fieldPtr,
            double x,
            double y,
            double z )
```

This checks whether the given point is within the boundary of the field. This is so the methods that retrieve a field value can short-circuit to zero. NOTE: this assumes, as is the case at the time of writing, that the CLAS12 fields have grids in cylindrical coordinates and length units of cm.

Parameters

fieldPtr	
x	the x coordinate in cm.
y	the y coordinate in cm.
z	the z coordinate in cm.

Returns

true if the point is within the boundary of the field.

2.1.1.3 containsCylindrical()

```
bool containsCylindrical (
            MagneticFieldPtr fieldPtr,
            double rho,
            double z )
```

This checks whether the given point is within the boundary of the field. This is so the methods that retrieve a field value can short-circuit to zero. Note ther is no phi parameter, because all values of phi are "contained." NOTE: this assumes, as is the case at the time of writing, that the CLAS12 fields have grids in cylindrical coordinates and length units of cm.

Parameters

fieldPtr	
rho	the rho coordinate in cm.
z	the z coordinate in cm.

Returns

true if the point is within the boundary of the field.

2.1.1.4 containsUnitTest()

```
char* containsUnitTest ( )
```

A unit test for checking the boundary contains check.

Returns

an error message if the test fails, or NULL if it passes.

2.1.1.5 getCompositeFieldValue()

```
void getCompositeFieldValue (
            FieldValuePtr fieldValuePtr,
            double x,
            double y,
            double z,
            MagneticFieldPtr field1,
            MagneticFieldPtr field2 )
```

Obtain the combined value of two fields. The field is obtained by tri-linear interpolation or nearest neighbor, depending on settings.

Parameters

fieldValuePtr	should be a valid pointer to a FieldValue. Upon return it will hold the value of the field, in kG, in Cartesian components Bx, By, BZ, regardless of the field coordinate system of the maps, obtained from all the field maps that it is given in the variable length argument list. For example, if torus and solenoid point to fields, then one can obtain the combined field at (x, y, z) by calling getCompositeFieldValue(fieldVal, x, y, x, torus, solenoid).
x	the x coordinate in cm.
y	the y coordinate in cm.
z	the z coordinate in cm.
field1	the first field.
field2	the second field.

2.1.1.6 getCompositeIndex()

```
int getCompositeIndex (
            MagneticFieldPtr fieldPtr,
            int n1,
            int n2,
            int n3 )
```

Get the composite index into the 1D data array holding the field data from the coordinate indices.

Parameters

fieldPtr	the pointer to the field map
n1	the index for the first coordinate.
n2	the index for the second coordinate.
n3	the index for the third coordinate.

Returns

the composite index into the 1D data array.

2.1.1.7 getCoordinateIndices()

```
void getCoordinateIndices (
            MagneticFieldPtr fieldPtr,
            double phi,
            double rho,
            double z,
            int * nPhi,
            int * nRho,
            int * nZ )
```

Get the coordinate indices from coordinates.

Parameters

fieldPtr	the field ptr.
phi	the value of the phi coordinate.
rho	the value of the rho coordinate.
z	the value of the z coordinate.
nPhi	upon return, the phi index.
nRho	upon return, the rho index.
nZ	upon return, the z index.

2.1.1.8 getFieldAtIndex()

```
FieldValuePtr getFieldAtIndex (
            MagneticFieldPtr fieldPtr,
            int compositeIndex )
```

Get the field at a given composite index.

Parameters

fieldPtr	a pointer to the field.
compositeIndex	the composite index.

Returns

a pointer to the field value, or NULL if out of range.

2.1.1.9 getFieldValue()

```
void getFieldValue (
            FieldValuePtr fieldValuePtr,
            double x,
            double y,
            double z,
            MagneticFieldPtr fieldPtr )
```

Obtain the value of the field by tri-linear interpolation or nearest neighbor, depending on settings.

Parameters

fieldValuePtr	should be a valid pointer to a FieldValue. Upon return it will hold the value of the field in kG, in Cartesian components Bx, By, BZ, regardless of the field coordinate system of the map.
x	the x coordinate in cm.
y	the y coordinate in cm.
z	the z coordinate in cm.
fieldPtr	a pointer to the field map.

2.1.1.10 invertCompositeIndex()

```
void invertCompositeIndex (
            MagneticFieldPtr fieldPtr,
            int index,
            int * phiIndex,
            int * rhoIndex,
            int * zIndex )
```

This inverts the "composite" index of the 1D data array holding the field data into an index for each coordinate. This can be used, for example, to find the grid coordinate values and field components.

Parameters

fieldPtr	the pointer to the field map
index	the composite index into the 1D data array. Upon return, coordinate indices of -1 indicate error.
phiIndex	will hold the index for the first coordinate.
rhoIndex	will hold the index for the second coordinate.
zIndex	will hold the index for the third coordinate.

2.1.1.11 nearestNeighborUnitTest()

```
char* nearestNeighborUnitTest ( )
```

A unit test for the test field.

Returns

an error message if the test fails, or NULL if it passes.

2.1.1.12 resetCell2D()

```
void resetCell2D (
            Cell2DPtr cell2DPtr,
            double rho,
            double z )
```

Reset the cell based on a new location. If the location is contained by the cell, then we can use some cached values, such as the neighbors. If it isn't, we have to recalculate all.

Parameters

cell2DPtr	a pointer to the 2D cell.
rho	the transverse coordinate, in cm.
z	the z coordinate, in cm.

2.1.1.13 resetCell3D()

```
void resetCell3D (
            Cell3DPtr cell3DPtr,
            double phi,
            double rho,
            double z )
```

Reset the cell based on a new location. If the location is contained by the cell, then we can use some cached values, such as the neighbors. If it isn't, we have to recalculate all.

Parameters

cell3DPtr	a pointer to the 3D cell.
phi	the azimuthal angle, in degrees
rho	the transverse coordinate, in cm.
z	the z coordinate, in cm.

2.1.1.14 setAlgorithm()

```
void setAlgorithm (
            enum Algorithm algorithm )
```

Set the global option for the algorithm used to extract field values.

Parameters

algorithm	it can either be

2.1.2 Variable Documentation

2.1.2.1 _algorithm

```
enum Algorithm _algorithm = INTERPOLATION
```

2.1.2.2 testFieldPtr

```
MagneticFieldPtr testFieldPtr
```

2.2 /Users/davidheddle/cmag/src/magfielddraw.c File Reference

```
#include "magfielddraw.h"
#include "svg.h"
#include "mapcolor.h"
#include "magfieldutil.h"
```

Functions

- void createSVGImageFixedZ (char *path, double z, MagneticFieldPtr torus, MagneticFieldPtr solenoid)
- void createSVGImageFixedPhi (char *path, double phi, MagneticFieldPtr torus, MagneticFieldPtr solenoid)

2.2.1 Function Documentation

2.2.1.1 createSVGImageFixedPhi()

```
void createSVGImageFixedPhi (
            char * path,
            double phi,
            MagneticFieldPtr torus,
            MagneticFieldPtr solenoid )
```

Create an SVG image of the fields at a fixed value of phi.

Parameters

path	the path to the svg file.
phi	the fixed value of phi in degrees. For the canonical sector 1 midplane, use phi = 0;
fieldPtr	torus the torus field (can be NULL).
fieldPtr	torus the solenoid field (can be NULL).

2.2.1.2 createSVGImageFixedZ()

```
void createSVGImageFixedZ (
            char * path,
            double z,
            MagneticFieldPtr torus,
            MagneticFieldPtr solenoid )
```

Create an SVG image of the fields at a fixed value of z.

Parameters

path	the path to the svg file.
z	the fixed value of z in cm.
fieldPtr	torus the torus field (can be NULL).
fieldPtr	torus the solenoid field (can be NULL).

2.3 /Users/davidheddle/cmag/src/magfieldio.c File Reference

```
#include "magfieldio.h"
#include "magfieldutil.h"
#include <stdlib.h>
#include <time.h>
#include <math.h>
```

Functions

- MagneticFieldPtr initializeTorus (const char ∗torusPath)
- MagneticFieldPtr initializeSolenoid (const char ∗solenoidPath)
- void createCell3D (MagneticFieldPtr fieldPtr)
- void createCell2D (MagneticFieldPtr fieldPtr)
- void freeCell3D (Cell3DPtr cell3DPtr)
- void freeCell2D (Cell2DPtr cell2DPtr)

2.3.1 Function Documentation

2.3.1.1 createCell2D()

```
void createCell2D (
            MagneticFieldPtr fieldPtr )
```

Create a 2D cell, which is used by the solenoid, since the lack of phi dependence renders the solenoidal field effectively 2D. Note that nothing is returned, the field's 2D cell pointer is made to point at the cell, and the cell is given a reference to the field.

Parameters

fieldPtr	a pointer to the solenoid field.

2.3.1.2 createCell3D()

```
void createCell3D (
            MagneticFieldPtr fieldPtr )
```

Create a 3D cell, which is used by the torus. Note that nothing is returned, the field's 2D cell pointer is made to point at the cell, and the cell is given a reference to the field.

Parameters

fieldPtr	a pointer to the solenoid field.

2.3.1.3 freeCell2D()

```
void freeCell2D (
            Cell2DPtr cell2DPtr )
```

Free the memory associated with a 2D cell.

Parameters

cell2DPtr	a pointer to the cell.

2.3.1.4 freeCell3D()

```
void freeCell3D (
            Cell3DPtr cell3DPtr )
```

Free the memory associated with a 3D cell.

Parameters

cell3DPtr	a pointer to the cell.

2.3.1.5 initializeSolenoid()

```
MagneticFieldPtr initializeSolenoid (
            const char * solenoidPath )
```

Initialize the solenoid field.

Parameters

solenoidPath	a path to a solenoid field map. If you want to use environment variables, pass NULL in this parameter, in which case the code make two attempts two attempts at finding the field. The first will be to try the a path specified by the COAT_MAGFIELD_SOLENOIDMAP environment variable. If that fails, it will then check SOLENOIDMAP.

Returns

a valid field pointer on success, NULL on failure.

2.3.1.6 initializeTorus()

```
MagneticFieldPtr initializeTorus (
            const char * torusPath )
```

Initialize the torus field.

Parameters

torusPath	a path to a torus field map. If you want to use environment variables, pass NULL in this parameter, in which case the code make two attempts two attempts at finding the field. The first will be to try the a path specified by the COAT_MAGFIELD_TORUSMAP environment variable. If that fails, it will then check TORUSMAP.

Returns

a valid field pointer on success, NULL on failure.

2.4 /Users/davidheddle/cmag/src/magfieldutil.c File Reference

```
#include "magfield.h"
#include "magfieldio.h"
#include "magfieldutil.h"
#include "munittest.h"
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>
```

Functions

- double toDegrees (double angRad)
- double toRadians (double angDeg)
- bool sameNumber (double v1, double v2)
- void cartesianToCylindrical (const double x, const double y, double *phi, double *rho)
- void cylindricalToCartesian (double *x, double *y, const double phi, const double rho)
- void normalizeAngle (double *angDeg)
- double fieldMagnitude (FieldValue *fvPtr)
- double relativePhi (double absolutePhi)
- int getSector (double phi)
- void printFieldSummary (MagneticFieldPtr fieldPtr, FILE *stream)
- const char * fieldUnits (MagneticFieldPtr fieldPtr)
- const char * lengthUnits (MagneticFieldPtr fieldPtr)
- void printFieldValue (FieldValuePtr fvPtr, FILE *stream)
- MagneticFieldPtr createFieldMap ()
- void freeFieldMap (MagneticFieldPtr fieldPtr)
- void stringCopy (char **dest, const char *src)
- int randomInt (int minVal, int maxVal)
- int sign (double x)
- double randomDouble (double minVal, double maxVal)

- char ∗ conversionUnitTest ()
- char ∗ randomUnitTest ()
- int descBinarySearch (double ∗array, int lower, int upper, double x)
- int binarySearch (double ∗array, int lower, int upper, double x)
- int cmpfunc (const void ∗a, const void ∗b)
- void sortArray (double ∗array, int length)
- char ∗ binarySearchUnitTest ()

Variables

- double const TINY = 1.0e-8
- int mtests_run = 0
- const double PIOVER180 = M_PI/180.
- const char ∗ csLabels [ ] = { "cylindrical", "Cartesian" }
- const char ∗ lengthUnitLabels [ ] = { "cm", "m" }
- const char ∗ angleUnitLabels [ ] = { "degrees", "radians" }
- const char ∗ fieldUnitLabels [ ] = { "kG", "G", "T" }

2.4.1 Function Documentation

2.4.1.1 binarySearch()

```
int binarySearch (
            double * array,
            int lower,
            int upper,
            double x )
```

A binary search through a sorted array.

Parameters

array	an array sorted (ascending).
lower	pass 0 to this, it is here for recursion
upper	pass the length of the array - 1.
x	pass the value to search for.

Returns

-1 if the value is out of range. othewise return index [0..length-2] such that array[index] $<$ value $<$ array[index+1];

2.4.1.2 binarySearchUnitTest()

```
char* binarySearchUnitTest ( )
```

A unit test for the binary search

Returns

an error message if the test fails, or NULL if it passes.

2.4.1.3 cartesianToCylindrical()

```
void cartesianToCylindrical (
            const double x,
            const double y,
            double * phi,
            double * rho )
```

Converts 2D Cartesian coordinates to polar. This is used because the two coordinate systems we use are Cartesian and cylindrical, whose 3D transformations are equivalent to 2D Cartesian to polar. Note the azimuthal angle output is in degrees, not radians.

Parameters

x	the x component.
y	the y component.
phi	will hold the angle, in degrees, in the range [0, 360).
rho	the longitudinal component.

2.4.1.4 cmpfunc()

```
int cmpfunc (
            const void * a,
            const void * b )
```

A comparator for qsort

Parameters

a	one value
b	another value

Returns

2.4.1.5 conversionUnitTest()

```
char* conversionUnitTest ( )
```

A unit test for the conversions

Returns

an error message if the test fails, or NULL if it passes.

2.4.1.6 createFieldMap()

```
MagneticFieldPtr createFieldMap ( )
```

Allocate a field map with no content.

Returns

a pointer to an empty field map structure.

2.4.1.7 cylindricalToCartesian()

```
void cylindricalToCartesian (
            double * x,
            double * y,
            const double phi,
            const double rho )
```

Converts polar coordinates to 2D Cartesian. This is used because the two coordinate systems we use are Cartesian and cylindrical, whose 3D transformations are equivalent to 2D polar to Cartesian. Note the azimuthal angle input is in degrees, not radians.

Parameters

x	will hold the x component.
y	will hold the y component.
phi	the azimuthal angle, in degrees.
rho	the longitudinal component.

2.4.1.8 descBinarySearch()

```
int descBinarySearch (
            double * array,
            int lower,
            int upper,
            double x )
```

A binary search through a sorted array.

Parameters

array	an array sorted (descending).
lower	pass 0 to this, it is here for recursion
upper	pass the length of the array - 1.
x	pass the value to search for.

Returns

-1 if the value is out of range. othewise return index [0..length-2] such that array[index] $<$ value $<$ array[index+1];

2.4.1.9 fieldMagnitude()

```
double fieldMagnitude (
            FieldValue * fvPtr )
```

Get the magnitude of a field value.

Parameters

fvPtr	a pointer to a field value

Returns

return: the magnitude of a field value.

2.4.1.10 fieldUnits()

```
const char* fieldUnits (
            MagneticFieldPtr fieldPtr )
```

Get the field units of the magnetic field

Parameters

fieldPtr	a pointer to the field.

Returns

a string representing the field units, e.g. "kG".

2.4.1.11 freeFieldMap()

```
void freeFieldMap (
            MagneticFieldPtr fieldPtr )
```

Free the memory associated with a field map.

Parameters

fieldPtr	a pointer to the field

2.4.1.12 getSector()

```
int getSector (
            double phi )
```

Obtain the CLAS12 sector from the phi value

Parameters

phi	the azimuthal angle in degrees

Returns

the sector [1..6].

2.4.1.13 lengthUnits()

```
const char* lengthUnits (
            MagneticFieldPtr fieldPtr )
```

Get the length units of the magnetic field

Parameters

fieldPtr	a pointer to the field.

Returns

a string representing the length units, e.g. "cm".

2.4.1.14 normalizeAngle()

```
void normalizeAngle (
            double * angDeg )
```

This will normalize an angle in degrees. We use for normaliztion that the angle should be in the range [0, 360).

Parameters

angDeg	the angle in degrees. It will be normalized.

2.4.1.15 printFieldSummary()

```
void printFieldSummary (
            MagneticFieldPtr fieldPtr,
            FILE * stream )
```

Print a summary of the map for diagnostics and debugging.

Parameters

fieldPtr	the pointer to the map.
stream	a file stream, e.g. stdout.

2.4.1.16 printFieldValue()

```
void printFieldValue (
            FieldValuePtr fvPtr,
            FILE * stream )
```

Print the components and magnitude of field value.

Parameters

fvPtr	a pointer to the field value
stream	a file stream, e.g. stdout.

2.4.1.17 randomDouble()

```
double randomDouble (
            double minVal,
            double maxVal )
```

Obtain a random double in the range[minVal, maxVal]. Used for testing.

Parameters

minVal	the minimum value
maxVal	the maximum Value;

Returns

2.4.1.18 randomInt()

```
int randomInt (
            int minVal,
            int maxVal )
```

Obtain a random int in an inclusive range[minVal, maxVal]. Used for testing.

Parameters

minVal	the minimum value
maxVal	the maximum Value;

Returns

2.4.1.19 randomUnitTest()

```
char* randomUnitTest ( )
```

A unit test for the random number generator

Returns

an error message if the test fails, or NULL if it passes.

2.4.1.20 relativePhi()

```
double relativePhi (
            double absolutePhi )
```

Must deal with the fact that for a symmetric torus we only have the field between 0 and 30 degrees.

Parameters

absolutePhi	the absolut value of phi in degrees.

Returns

a phi relative to the midplabe, [-30, 30]

2.4.1.21 sameNumber()

```
bool sameNumber (
            double v1,
            double v2 )
```

The usual test to see if two floating point numbers are close enough to be considered equal. Test accuracy depends on the global const TINY, set to 1.0e-10.

Parameters

v1	one value.
v2	another value.

Returns

true if the values are close enough to be considered equal.

2.4.1.22 sign()

```
int sign (
            double x )
```

Sign function

Parameters

x	the value to check

Returns

-1, 0 or 1

2.4.1.23 sortArray()

```
void sortArray (
            double * array,
            int length )
```

Use built in quick sort to sort a double array in ascending order

Parameters

array	the array to sort
length	the length of the array

2.4.1.24 stringCopy()

```
void stringCopy (
            char ** dest,
            const char * src )
```

Copy a string and create the pointer

Parameters

dest	on input a pointer to an unallocated string. On output the string will be allocated and contain a copy of src.
src	the string to be copied.

2.4.1.25 toDegrees()

```
double toDegrees (
            double angRad )
```

Convert an angle from radians to degrees.

Parameters

angRad	the angle in radians.

Returns

the angle in degrees.

2.4.1.26 toRadians()

```
double toRadians (
            double angDeg )
```

Convert an angle from degrees to radians.

Parameters

angDeg	the angle in degrees.

Returns

the angle in radians.

2.4.2 Variable Documentation

2.4.2.1 angleUnitLabels

```
const char* angleUnitLabels[] = { "degrees", "radians" }
```

2.4.2.2 csLabels

```
const char* csLabels[] = { "cylindrical", "Cartesian" }
```

2.4.2.3 fieldUnitLabels

```
const char* fieldUnitLabels[] = { "kG", "G", "T" }
```

2.4.2.4 lengthUnitLabels

```
const char* lengthUnitLabels[] = { "cm", "m" }
```

2.4.2.5 mtests_run

```
int mtests_run = 0
```

2.4.2.6 PIOVER180

```
const double PIOVER180 = M_PI/180.
```

2.4.2.7 TINY

```
double const TINY = 1.0e-8
```

2.5 /Users/davidheddle/cmag/src/maggrid.c File Reference

```
#include "maggrid.h"
#include "magfieldutil.h"
#include "munittest.h"
#include <stdlib.h>
#include <math.h>
#include <time.h>
```

Functions

- GridPtr createGrid (const char ∗name, const double minVal, const double maxVal, const unsigned int num)
- char ∗ gridStr (GridPtr gridPtr)
- int getIndex (const GridPtr gridPtr, const double val)
- double valueAtIndex (GridPtr gridPtr, int index)
- char ∗ gridUnitTest ()

2.5.1 Function Documentation

2.5.1.1 createGrid()

```
GridPtr createGrid (
            const char * name,
            const double minVal,
            const double maxVal,
            const unsigned int num )
```

Create a uniform (equally spaced) coordinate coordinate grid.

Parameters

the	name of the coordinate, e.g. "phi".
minVal	the minimum value of the grid.
maxVal	the maximum value of the grid.
num	the number of points on the grid, including the ends.

Returns

a pointer to the coordinate grid.

2.5.1.2 getIndex()

```
int getIndex (
            const GridPtr gridPtr,
            const double val )
```

Get the index of a value.

Parameters

gridPtr	the pointer to the coordinate grid.
val	the value to index.

Returns

the index, [0..N-2] where, or -1 if out of bounds. The value should be bounded by values[index] and values[index+1].

2.5.1.3 gridStr()

```
char* gridStr (
            GridPtr gridPtr )
```

Get a string representation of the grid.

Parameters

gridPtr	the pointer to the coordinate grid.

Returns

a string representation of the grid.

2.5.1.4 gridUnitTest()

```
char* gridUnitTest ( )
```

A unit test for the coordinate grid code.

Returns

an error message if the test fails, or NULL if it passes.

2.5.1.5 valueAtIndex()

```
double valueAtIndex (
            GridPtr gridPtr,
            int index )
```

Get the value of the grid at a given index

Parameters

gridPtr	the pointer to the grid
index	the index

Returns

the value of the grid at the given index, or NAN if the index is out of range

2.6 /Users/davidheddle/cmag/src/main.c File Reference

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "magfield.h"
#include "magfieldio.h"
#include "munittest.h"
#include "magfieldutil.h"
#include "magfielddraw.h"
```

Functions

- int main (int argc, const char *argv[ ])

2.6.1 Function Documentation

2.6.1.1 main()

```
int main (
            int argc,
            const char * argv[ ] )
```

The main method of the test application.

Parameters

argc	the number of arguments
argv	the command line argument. Only one is processed, the path to the directory containing the magnetic fields. If that argument is missing, it will look in /Users/davidheddle/magfield.

Returns

0 on successful completion, 1 if any error occurred.

2.7 /Users/davidheddle/cmag/src/mapcolor.c File Reference

```
#include "mapcolor.h"
#include "magfieldutil.h"
#include <stdlib.h>
```

Functions

- char ∗ getColor (ColorMapPtr cmapPtr, double value)
- ColorMapPtr defaultColorMap ()
- void colorToHex (char ∗colorStr, int r, int g, int b)

2.7.1 Function Documentation

2.7.1.1 colorToHex()

```
void colorToHex (
            char * colorStr,
            int r,
            int g,
            int b )
```

Get the hex color string from color components

Parameters

colorStr	must be at last 8 characters
r	the red component [0..255]
g	the green component [0..255]
b	the blue component [0..255]

2.7.1.2 defaultColorMap()

```
ColorMapPtr defaultColorMap ( )
```

Get the default color map optimized for displaying torus and solenoid

Returns

he default color map.

2.7.1.3 getColor()

```
char* getColor (
            ColorMapPtr cmapPtr,
            double value )
```

Get a color from a color map.

Parameters

cmapPtr	a valid pointer to a color map.
value	the value to convert into a color.

Returns

the color in "#rrggbb" format.

2.8 /Users/davidheddle/cmag/src/svg.c File Reference

```
#include "svg.h"
```

Functions

- svg * svgStart (char *path, int width, int height)
- void svgEnd (svg *psvg)
- void svgCircle (svg *psvg, char *stroke, int strokewidth, char *fill, int r, int cx, int cy)
- void svgLine (svg *psvg, char *stroke, int strokewidth, int x1, int y1, int x2, int y2)
- void svgRectangle (svg *psvg, int width, int height, int x, int y, char *fill, char *stroke, int strokewidth, int radiusx, int radiusy)
- void svgFill (svg *psvg, char *fill)
- void svgText (svg *psvg, int x, int y, char *fontfamily, int fontsize, char *fill, char *stroke, char *text)
- void svgRotatedText (svg *psvg, int x, int y, char *fontfamily, int fontsize, char *fill, char *stroke, int angle, char *text)
- void svgEllipse (svg *psvg, int cx, int cy, int rx, int ry, char *fill, char *stroke, int strokewidth)

2.8.1 Function Documentation

2.8.1.1 svgCircle()

```
void svgCircle (
            svg * psvg,
            char * stroke,
            int strokewidth,
            char * fill,
            int r,
            int cx,
            int cy )
```

Draw a circle. All units are pixels.

Parameters

psvg	pointer to the svg information.
stroke	the outline color, usually in "#rrggbb" format.
strokewidth	border line width.
fill	the fill color, usually in "#rrggbb" format.
r	the radius.
cx	the x center
cy	the y center.

2.8.1.2 svgEllipse()

```
void svgEllipse (
            svg * psvg,
            int cx,
            int cy,
            int rx,
            int ry,
            char * fill,
            char * stroke,
            int strokewidth )
```

Draw an ellipse. All units are pixels.

Parameters

psvg	pointer to the svg information.
cx	the horizontal center.
cy	the vertical center.
rx	the horizontal radius.
ry	the vertical radius.
fill	the fill color, usually in "#rrggbb" format.
stroke	the outline color, usually in "#rrggbb" format.
strokewidth	the width of the outline.

2.8.1.3 svgEnd()

```
void svgEnd (
            svg * psvg )
```

Finalize the svg file and free all space.

Parameters

psvg	pointer to the svg information.

2.8.1.4 svgFill()

```
void svgFill (
            svg * psvg,
            char * fill )
```

Fill the whole image area.

Parameters

psvg	pointer to the svg information.
fill	the fill color, usually in "#rrggbb" format.

2.8.1.5 svgLine()

```
void svgLine (
            svg * psvg,
            char * stroke,
            int strokewidth,
            int x1,
            int y1,
            int x2,
            int y2 )
```

Draw a line. All units are pixels.

Parameters

psvg	pointer to the svg information.
stroke	the line color, usually in "#rrggbb" format.
strokewidth	the width of the line.
x1	x coordinate of start.
y1	y coordinate of start.
x2	x coordinate of end.
y2	y coordinate of end.

2.8.1.6 svgRectangle()

```
void svgRectangle (
            svg * psvg,
            int width,
            int height,
            int x,
            int y,
            char * fill,
            char * stroke,
            int strokewidth,
            int radiusx,
            int radiusy )
```

Draw a rectangle. All units are pixels.

Parameters

psvg	pointer to the svg information.
width	the width of the rectangle.
height	the height of the rectangle.
x	the left of the rectangle.
y	th top of the rectangle.
fill	the fill color, usually in "#rrggbb" format.
stroke	the outline color, usually in "#rrggbb" format.
strokewidth	the width of the outline.
radiusx	for rounding the corners.
radiusy	for rounding the corners.

2.8.1.7 svgRotatedText()

```
void svgRotatedText (
            svg * psvg,
            int x,
            int y,
            char * fontfamily,
            int fontsize,
            char * fill,
            char * stroke,
            int angle,
            char * text )
```

Draw some text. All units are pixels.

Parameters

psvg	pointer to the svg information.

Parameters

x	the baseline horizontal start
y	the baseline vertical start
fontfamily	the font family.
fontsize	the font size.
fill	the fill color, usually in "#rrggbb" format.
stroke	the outline color, usually in "#rrggbb" format.
angle	the rotation angle in degrees.
text	the text to draw.

2.8.1.8 svgStart()

```
svg* svgStart (
            char * path,
            int width,
            int height )
```

Initialize the svg file creation process.

Parameters

path	a path to the ouyput file.
width	the width of the image in pixels.
height	the height of the image in pixels.

Returns

a pointer to the svg object.

2.8.1.9 svgText()

```
void svgText (
            svg * psvg,
            int x,
            int y,
            char * fontfamily,
            int fontsize,
            char * fill,
            char * stroke,
            char * text )
```

Draw some text. All units are pixels.

Parameters

psvg	pointer to the svg information.
x	the baseline horizontal start
y	the baseline vertical start
fontfamily	the font family.
fontsize	the font size.
fill	the fill color, usually in "#rrggbb" format.
stroke	the outline color, usually in "#rrggbb" format.
text	the text to draw.

2.9 /Users/davidheddle/cmag/src/testdata.c File Reference

Variables

- double torusNN [33][6]
- double solenoidNN [34][6]

2.9.1 Variable Documentation

2.9.1.1 solenoidNN

```
double solenoidNN[34][6]
```

2.9.1.2 torusNN

```
double torusNN[33][6]
```

B Field Map File Format

Provided mostly for completeness, 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. [15]

[15] 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