1 OPAL Work Specification Document

Change History

Version	Changes	Author	Date
1	First draft	C.Rogers	06/08/2012

1.1 Project Summary

Project Title	OPAL Cyclotron Field Map Update
Main Issue	
Subtask Issues	
Project Lead	C. Rogers
Project Supervisor(s)	C. Rogers, S. Sheehy
Associated Manpower	

Table of Contents

1 OPAL Work Specification Document....1
1.1 Project Summary....1
2 Motivation and Overview....3
3 Task....3
3.1 SectorMagneticFieldMap....3
3.2 Ring-type beamline....4
3.3 Field Map Output....4
4 Existing Opal Code....5
4.1 OPAL-Cycl Field Map Implementation....5
4.2 OPAL-Cycl Call Structure....5
4.3 OPAL-Cycl Setup and User Interface....5
4.4 OPAL-T Field Map Abstraction....6
4.5 OPAL-T Call Structure....6
5 Existing MAUS Code....6
5.1 Meshing Routines....7
5.2 Interpolation Routines....7
5.3 Field Routines....8
5.4 User Interface and Field Map Placement....8
6 Task Breakdown....9
6.1 Implementation of 3D magnetostatic field map....9
6.2 Comment on Implementation of OPAL-T field maps as an OPAL-CYCL element....9
6.3 OpalRingDefinition....10
6.4 OpalRing....10
6.5 OpalFieldMapWrite....11
6.6 Modifications to ParallelCyclotronTracker....11
6.7 Testing....12
6.8 Documentation....12
6.9 Licence....12
7 Effort and Timescale....12
7.1 Effort Available....12
7.2 Major Milestones....13

2 Motivation and Overview

OPAL is a code for multiparticle tracking in the context of intense beams in cylcotron and linac geometries. OPAL has two modes of running, so-called “OPAL-T” mode and “OPAL-CYCL” mode. OPAL-CYCL was the original development product of the OPAL group and mainly used to simulate the PSI cyclotron. OPAL-T was subsequently developed to support design and simulation of the X-FEL machine. Although the two codes share some common features, much of the code was redeveloped in a more organised way for OPAL-T. Subsquently, several interested persons have been investigating the potential for using OPAL-CYCL for multiparticle tracking in FFAG type geometries.

The field map routine in OPAL-CYCL is a hard coded bilinear 2D field map interpolation followed by an RF cavity. OPAL-T uses a generalised field map abstraction, within which several options for interpolation have been implemented.

The simulation of FFAGs can be performed using 2D or 3D field maps. Required that the 3D field maps for e.g. the ERIT ring should be implemented for tracking in OPAL.

MAUS is an existing code used for modelling Neutrino Factories and the Muon Ionisation Cooling Experiment. MAUS has a number of routines for field mapping, including an existing trilinear interpolator for 3d field maps.

3 Task

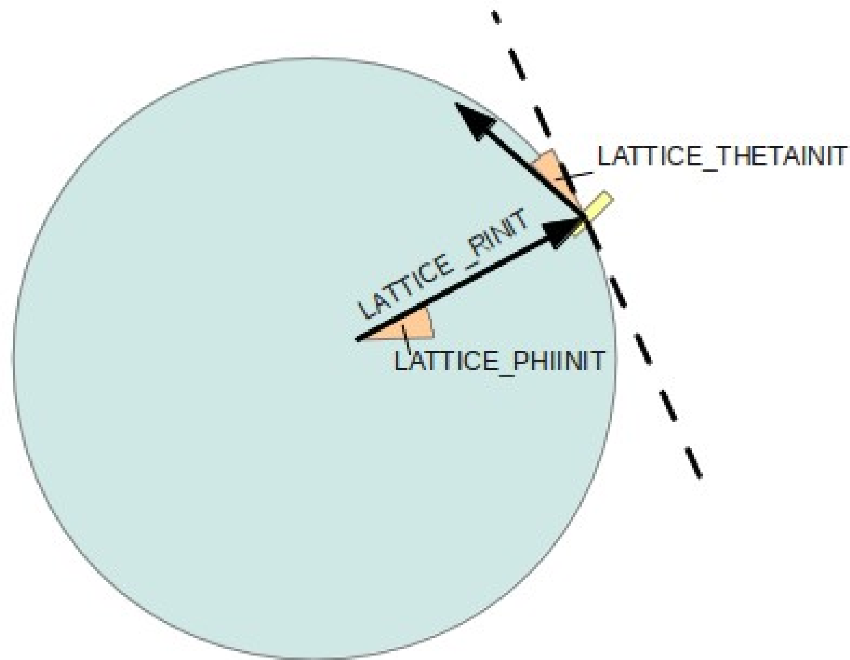
3.1 SectorMagneticFieldMap

The user should be able to add a new element SectorMagneticFieldMap that generates a sector magnet based on a read of a 3D field map and trilinear interpolation in each of (Bx, By, Bz). Sample input is

```
my_sector_magnet: SectorMagneticFieldMap, FMAPFN=”MyFieldMap.table”
```

where the bending angle, length and radius of curvature is determined from the field map file.

3.2 Ring-type beamline



Currently, the user can define a linear/beamline-type geometry using a command like

```
my_beamline: line=(drift_1,bend_1,quad_f,bend_2,drift_1,quad_d)
```

where dr1, b1, QF, etc are previously defined elements (like quadrupoles, dipoles and drifts). Additionally the user can define a ring-type geometry using a command like

```
my_ring: Cyclotron, TYPE="RING", CYHARMON=6, PHIINIT=-0.1, PRINIT=-
0.0022, RINIT=2043.5, SYMMETRY=8.0, RFFREQ=frequency,
FMAPFN="./s03av.nar";
```

An additional ability should be implemented to define a ring-type geometry in a manner drawing analogies from the beamline and Cyclotron declarations, e.g.

```
ring_def: ring_definition, CYHARMON=6, LATTICE_RINIT=2043.5,
LATTICE_PHIINIT=30, LATTICETHETA_INIT=-30, PHIINIT=-0.1, PRINIT=-
0.0022, RINIT=2043.5, SYMMETRY=8.0, RFFREQ=frequency

my_ring: line=(ring_def, drift_1,bend_1,quad_f,bend_2,drift_1,quad_d)
```

which would define an 8-cell ring with the first element placed at LATTICE_PHIINIT, LATTICETHETA_INIT and LATTICE_RINIT. Initial beam angle, radius and radial momentum defined as PHIINIT, RINIT, PRINIT respectively as in the standard Cyclotron definition.

3.3 Field Map Output

As a cross-check of the field mapping routines, it is desirable to write a 3D field map for some set of input coordinates . Sample input should be something like:

```
fm: 3d_field_map, START_PHI=0. END_PHI=90. START_R=1000., END_R=2000.,
START_Z=-100., END_Z=100., N_PHI=10, N_Z=10, N_R=10
```

Here the START_<> and END_<> indicate the start and end value of the map in each coordinate and the N_<> indicates the number of grid points in each dimension.

The output should be a standard format, say OPERA 3D format

```
<N_ROWS> <N_PHI> <N_R> <N_Z>
1 PHI [m]
2 R [m]
3 Z [m]
4 BPHI [T]
5 BR [T]
6 BZ [T]
0
<PHI> <R> <Z> <BPHI> <BR> <BZ>
... repeat for each row ...
```

where items in angle brackets represent data that should be filled in at run time.

4 Existing Opal Code

The structure of the existing code is described below.

4.1 OPAL-Cycl Field Map Implementation

The OPAL-Cycl field map routine is implemented in classic/5.0/src/AbsBeamline/Cylcotron.h,cpp.

Cyclotron.h contains a "BfieldData" struct holding field data and a "Cyclotron" class inheriting from "Component" that acts on the BfieldData.

The Cylcotron initialises by reading in a 2D field map. The file read is implemented for a number of different formats. Subsequently derivatives are calculated to 3rd order.

The routine takes a position in Cartesian coordinates and a time. The routine transforms to cylindrical coordinates; performs a bilinear interpolation to calculate a field in cylindrical coordinates (using the field and its first derivatives); transforms the field back to Cartesian coordinates; then adds contributions from several time varying Cartesian RF field maps. The routine returns true if the field contribution is 0 or false if the field contribution is non-zero. The RF field maps are instances of OPAL-T RF field maps (see below).

Accessors and mutators are also implemented.

4.2 OPAL-Cycl Call Structure

The field map routines are called by the tracking as implemented in src/Algorithms/ParallelCyclotronTracker.h,cpp. Two routines are implemented, ParallelCyclotronTracker::Tracker_RK4() and ParallelCyclotronTracker::Tracker_MTS(), each of which uses the "external" field to do numerical integration – 4th order Runge Kutta and 2nd order leap frog respectively. The field maps are contained in an STL list "FieldDimensions" of string component type, double[8] (that doesn't seem to be used, probably a bounding box) and Component* that are associated through nested STL pairs. Only the first Component of the beamline_list is consulted for field data. The rest are assumed to be IO components or equivalent.

4.3 OPAL-Cycl Setup and User Interface

The user interface for the OPAL-Cycl is implemented in src/Elements/OpalCyclotron.h,cpp which

inherits from the general UI class OpalElement. OpalCyclotron makes a CyclotronRep which is a type of Cyclotron (defined in classic/5.0/src/BeamlineCore/CyclotronRep.h,cpp).

Element definitions are registered through the OpalElement class implemented in src/Elements/OpalElement.h,cpp. Each attribute in the element is added to the static std::map < std::string, OwnPtr<AttCell> > OpalElement::attributeRegistry where the first item is the element name as defined in the input file, the second item is a smart pointer to a generic attCell, defined in src/Elements/AttCell.h,cpp. This is a base class for a typed string-value pairing.

OpalElement inherits from Element, defined in src/AbstractObjects/Element.h,cpp.

Elements are added into the FieldDimensions list by visit functions in the ParallelCyclotronTracker. ParallelCyclotronTracker holds a number of visit<object> functions e.g. visitCyclotron, each of which is called by an accept function within the object. The accept functions are called in turn by the TBeamline defined in classic/5.0/src/TBeamline.h,cpp, which is a collection of pointers to Elements (ElmPtr) generated by the input file parser.

4.4 OPAL-T Field Map Abstraction

The OPAL-T field map routines are implemented in classic/5.0/src/Fields. The Fieldmap class provides a pure abstract interface class with a number of routines that return field value and bounding box data appropriate for a Cartesian coordinate system, as well as templated routines to read files that contain field map data. RF field maps and various 1D and 2D magnetostatic field maps have been implemented.

Once read, the field maps are treated independently by different element types. For example, a solenoid will make a field expansion from a 1D field map based on a polynomial expansion of Bz and its derivatives.

4.5 OPAL-T Call Structure

Individual elements are loaded in the same way as in OPAL-Cycl. The ParallelTTracker is used for tracking, defined at src/Algorithms/ParallelTTracker.h,cpp. A different set of fields are permitted by defining a different set of visit functions on the tracker. Visit functions add elements to an OpalBeamline object defined at src/Elements/OpalBeamline.h,cpp. The OpalBeamline holds a method like

```
unsigned long getFieldAt(const Vector_t &pos,
                         const Vector_t &centroid, const double &t,
                         Vector_t &E, Vector_t &B);
```

that finds the element at a given position, performs coordinate transformations to transform into the local coordinate system of the element and sets the field to the field of that element. getFieldAt returns a value corresponding to different field conditions (has wakefield, has geometry, ...)

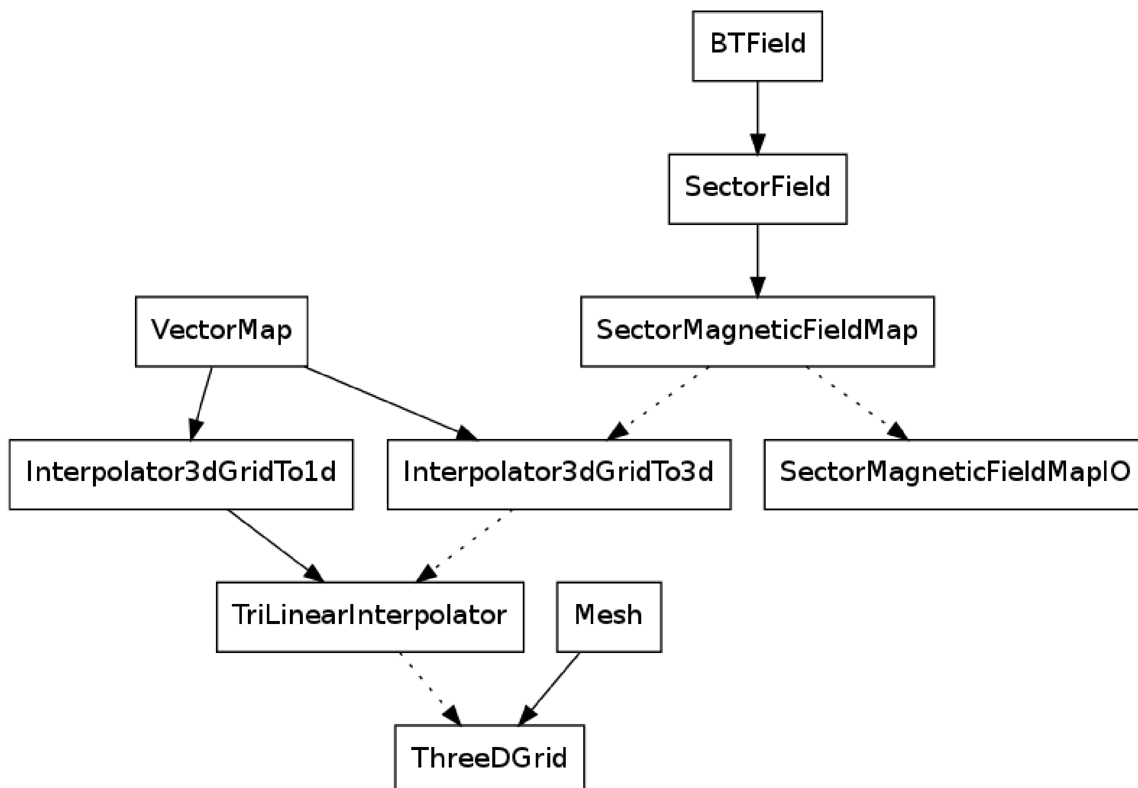
5 Existing MAUS Code

MAUS has a reasonably elaborate scheme for meshing, interpolation and field mapping. This improves code reusability; for example, some interpolation routines are shared between solenoid field maps interpolating from a 2d field map to produce a 3-vector, multipole field maps interpolating from a 3d field map to produce a 3-vector and electromagnetic field maps interpolating from a 3d field map to produce a 6-vector. The cost is a small performance impediment in the inner loop of particle tracking (several vtable lookups and pointer dereferences) and arguably an increase in code complexity.

MAUS underwent a rebranding exercise from its previous incarnation as G4MICE 2 years ago. New code is now required to obey certain style requirements, test coverage requirements and comment coverage. Routines are found either in the "legacy" area, indicating that they have not met the requirements for commenting, testing and style; or in the "common_cpp" area, indicating that they have met these requirements.

All non-legacy MAUS code is tested in "unit tests". Unit tests check the code in each class using specific hard-coded C code. The gtest library (Google test) is used to provide testing framework like comparators, test runners, etc. An additional level of application-level tests is implemented checking the integration of routines into the overall framework. Application and legacy test coverage is generally quite poor.

All non-legacy code is documented using doxygen-style comments. Typically a comment is provided for every function, even if only to indicate that the function performs the "obvious" action (like accessors or mutators).



Class diagram for MAUS: boxes represent classes. Full lines indicate inheritance relationship (parent class at the top/start of the line) while dashed lines indicate ownership (owned class at the bottom/end of the arrow)

5.1 Meshing Routines

MAUS has a number of meshing routines defined in src/legacy/Interface/Mesh.hh,cc. These routines describe regular rectangular and irregular triangular 1D, 2D and 3D grids. Iterators are supplied to iterate over all grid points. For this case, the relevant mesh is a regular 3D rectangular grid, ThreeDGrid. Routines are provided to convert between 3D index, 1D index, iterators and 3D positions within the grid. Meshes know nothing about the field value at each point in the mesh.

5.2 Interpolation Routines

MAUS has a number of interpolation routines describing interpolation off of meshes for various

different spacial dimensions, defined in src/legacy/Interface/Interpolator.hh,cc. Here we describe the 3d interpolation routines.

- VectorMap is an abstract base class that defines a mapping from some input vector of doubles to some output vector of doubles. Virtual functions are defined that return the dimension of the input and output spaces.
- Interpolator3dGridTo1d is an abstract class that specialises VectorMap to a mapping of a 3-vector to a 1-vector. A ThreeDGrid is required in the constructor.
- TriLinearInterpolator specialises Interpolator3dGridTo1d to make a tri-linear interpolation from field values on the rectangular 3D grid.
- Interpolator3dGridto3d is an abstract class that specialises VectorMap to a mapping of a 3-vector to a 3-vector. A 3D grid as well as field values on each of the grid points is required in the constructor. Additionally an interpolation routine definition is required in the constructor. Currently we use independent trilinear interpolators for each of the field values (Bx, By, Bz), although the structure outlined above makes this readily extensible to higher order interpolation or other algorithms such as FFT.

5.3 Field Routines

MAUS has an inheritance tree for field map routines also.

- BTField defined in src/legacy/BeamTools/BTField.hh,cc defines the abstraction for field map routines. Virtual functions are provided to access field values at a point, print a summary of the field, get bounding box size, get derivatives of the field map (e.g. for checking divB = 0), etc.
- SectorField defined in src/common_cpp/FieldTools/SectorField.hh,cc specialises BTField to provide additional routines for operation in a bent environment. For example, transformations are provided from cylindrical polar coordinates to Cartesian coordinates; transformations are provided to convert bounding boxes between polar coordinates and Cartesian coordinates. An additional function is provided to return field values in polar coordinates.
- SectorMagneticFieldMap defined in src/common_cpp/FieldTools/SectorMagneticFieldMap.hh,cc specialises SectorField with interfaces to the 3D interpolator and IO routines. Note that the grid in SectorMagneticFieldMap is on lines in (r, theta, y)
- SectorMagneticFieldMapIO defined in src/common_cpp/FieldTools/SectorMagneticFieldMap.hh,cc provides IO routines to generate an Interpolator3dGridto3d.

5.4 User Interface and Field Map Placement

MAUS additionally has routines to provide User Interfaces and to place field maps with arbitrary 3D position and rotation. These routines will not be ported to OPAL.

6 Task Breakdown

6.1 Implementation of 3D magnetostatic field map

The 3D field map routines from MAUS should be ported to OPAL.

- BTField inheritance will be replaced with MagneticFieldMap and child class function definitions will be modified appropriately.
- Where multiple classes have been placed into the same file these should be split into separate files.
- Dependencies on GSL and other third party libraries should be stripped out. There are no major dependencies on GSL, but a few small helper functions are invoked.
- MAUS exceptions will be replaced with OpalExceptions.
- Where comments are not sufficient (legacy code) comments should be improved. Typically, a long comment is expected explaining the purpose and data of a particular class. Additional comments are expected on every public member; the length of comments is dependent on the complexity of the function. Comments should be in Doxygen syntax.
- Style should be modified to fit OPAL style guide.
- Unit tests will not be ported.
- The following files will be added to classic/5.0/src/Fields
    - VectorMap.h, VectorMap.cpp
    - Interpolator3dGridTo1d.h, Interpolator3dGridTo1d.cpp
    - Interpolator3dGridTo3d.h, Interpolator3dGridTo3d.cpp
    - TriLinearInterpolator.h, TriLinearInterpolator.cpp
    - Mesh.h, Mesh.cpp
    - ThreeDGrid.h, ThreeDGrid.cpp
    - SectorField.h, SectorField.cpp
    - SectorMagneticFieldMap.h, SectorMagneticFieldMap.cpp
    - SectorMagneticFieldMapIO.h, SectorMagneticFieldMapIO.cpp

6.2 Comment on Implementation of OPAL-T field maps as an OPAL-CYCL element

The OPAL-T field maps are called assuming a linac-type geometry. Objects are placed serially. For example, one might place a Quadrupole, Drift, Quadrupole, Drift to set up a FODO cell. This is okay for a linac geometry, but for a ring geometry it is desirable to set additionally a radius and to check that the ring is closed (or otherwise). Also it should be noted that, as described previously, the tracking routines currently only check for fields in the first element of the beamline_list in the ParallelCyclotronTracker. As the ParallelCyclotronTracker is an extremely complicated piece of code, it is envisaged that this should be disturbed as little as possible.

6.3 OpalRingDefinition

A new class, OpalRingDefinition, should be implemented in src/Elements/OpalRingDefinition.h,cpp inheriting from OpalElement. OpalRingDefinition provides necessary information defining the set up of a ring geometry. OpalRingDefinition should define the following attributes

`CYHARMON`	Real	Defines the assumed harmonic number of the ring.
`LATTICE_RINIT`	Real	Defines the initial radius of the first element to be placed in the ring.
`LATTICE_PHIINIT`	Real	Defines the initial angle around the ring of the first element to be placed.
`LATTICETHETA_INIT`	Real	Defines the angle relative to the tangent of the ring for the first element to be placed.
`PHIINIT`	Real	Defines the initial angle around the ring of the beam.
`PRINIT`	Real	Defines an initial p_r momentum offset of the beam.
`RINIT`	Real	Defines the initial radius of the beam.
`SYMMETRY`	Real	Defines the symmetry properties of the lattice.
`RFFREQ`	Real	Defines the nominal RF frequency of the ring.

6.4 OpalRing

A new class, OpalRing should be added in src/Elements/OpalRing.h,cpp that should be analogous to OpalBeamline but additionally inheriting from Component. Ring should provide a function “apply” analogous to the Cyclotron::apply function and the OpalBeamline::getFieldAt function that returns the field at a given position in Cartesian coordinates by iterating over the element list, seeking the element at a particular position and setting the field from that element.

```
bool apply(const Vector_t &pos, const Vector_t &centroid,
           const double &t, Vector_t &E, Vector_t &B);
```

Visit functions should be modified in ParallelCyclotronTracker to append elements to the OpalRing also. The first element in the ParallelCyclotronTracker beamline_list and myElements list should always point to OpalRing. In the case that a Cyclotron is defined, this should be called through the OpalRing. Backwards compatibility will be maintained but the call to acquire field information should always now go through OpalRing enabling a more sophisticated ring geometry composed of multiple beamline elements to be implemented.

Additionally

- OpalRing should hold accessors and mutators to control the alignment of the first element (radius, angle in ring coordinates, angle with respect to the tangent of the ring).
- OpalRing should throw an exception if the ring is not closed or the total bending angle is not 360° (within a floating point tolerance of say 1e-9).
- OpalRing should throw an exception if the first element in the ring is not an OpalRingDefinition or a Cyclotron.

6.5 OpalFieldMapWrite

A new class, OpalFieldMapWrite, should be added to src/Elements/OpalFieldMapWrite.h,cpp. OpalFieldMapWrite

- Holds a pointer to the OpalRing (set by ParallelCyclotronTracker::visitFieldMapWrite)
- Loops over each of the points in the grid
- Transforms the position to Cartesian coordinates
- Calls OpalRing::apply at each of the positions defined in the Mesh and stores the field values
- Transforms the field to ring coordinates.
- Writes the data to a file

The write loop should be called by OpalRing once

The class should define the following attributes

START_PHI	Real	Defines the start phi angle of the field map file.
END_PHI	Real	Defines the end phi angle of the field map file.
START_R	Real	Defines the start radial position of the field map file.
END_R	Real	Defines the angle relative to the tangent of the ring for the first element to be placed.
START_Z	Real	Defines the initial angle around the ring of the beam.
END_Z	Real	Defines an initial p_r momentum offset of the beam.
N_PHI	Real	Defines the initial radius of the beam.
N_Z	Real	Defines the symmetry properties of the lattice.
N_R	Real	Defines the nominal RF frequency of the ring.
FIELDMAPWRITEFN	String	Name of the file to which field map data will be written.

6.6 Modifications to ParallelCyclotronTracker

visit methods should be added or modified in ParallelCyclotronTracker for the following Components:

- OpalRingDefinition: should modify the OpalRing; presumably can only be visited once OpalRing is defined.
- OpalFieldMapWrite: should be added to the OpalRing.
- OpalRing: set OpalRing in ParallelCyclotronTracker to be the first member of the fields list
- Cyclotron: change the Cyclotron to make it the top member in the OpalRing fields list

6.7 Testing

The following integration tests should be added:

- Track through a set of ERIT-style field map and checks that the closed orbit is in the expected position.
- Read in the ERIT-style field map and write it back out again; check that the input field map is the same as the output field map.

Cyclotron and other tests should continue to pass unmodified.

6.8 Documentation

New entries should be made for each of the new components added:

- OpalRing
- OpalRingDefinition
- OpalFieldMapWrite
- SectorMagneticFieldMap

6.9 Licence

Code will be distributed under the Modified BSD licence:

```
Copyright (c) 2012, Chris Rogers
All rights reserved.
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:
1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
3. Neither the name of STFC nor the names of its contributors may be
used to endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

7 Effort and Timescale

7.1 Effort Available

Chris Rogers will contribute at 50% level to this task

7.2 Major Milestones

Item	**Time**	**Due date (@ 50% FTE)**
Implementation of 3D magnetostatic field map	1 week	24 September
OpalRingDefinition	3 days	31 September
OpalFieldMapWrite	3 days	7 October
Modifications to ParallelCyclotronTracker	3 days	14 October
Testing (and debugging)	1 week	28 October
Documentation	3 days	2 November