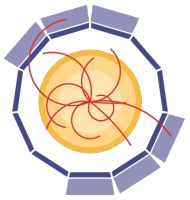

AIDA 2020

Advanced European Infrastructures for Detectors at Accelerators

DDRec

Reconstruction Interface for the DD4hep Geometry Description Toolkit

F.Gaede

This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168.

CERN, 1211 Geneva 23, Switzerland
Desy, 22607 Hamburg, Germany

Abstract

The reconstruction of particle tracks and clusters in an High Energy Physics detector requires information about the geometrical and material properties of the various tracking and calorimeter subdetectors. In general, a higher level view on the detector geometry is needed than for the purpose of simulating the detailed detector response with tools such as Geant4[4]. This higher level view typically involves the abstraction of detector layers, the corresponding measurement surfaces, accumulation of dead material along a path and conversion between cellIDs and positions. While in principle it is of course possible to extract this information from the detailed detector geometry model used for simulation, doing so would tightly couple the reconstruction code to the specific implementation of the simulation model. `DDRec` provides a generalized API for reconstruction that can be used to decouple the details of the `DD4hep` [1] detector geometry model from the reconstruction algorithms.

Document History		
Document version	**Date**	**Author**
1.0	**11/11/2014**	**Frank Gaede CERN/DESY**

Contents

1 Introduction

This manual introduces the DDRec package which is part of DD4hep and provides the high level view on the HEP detector geometry that is needed during reconstruction and analysis. In the detailed simulation of the response of a typical High Energy Physics detector very little information is needed in principle on the actual structure of the material distribution in the detector. This becomes obvious if one considers the fact that the Geant4 program is also used in medical applications where the human body is approximated by a voxelised phantom. During the reconstruction of particle trajectories and calorimeter clusters, in particular in the phase of pattern recognition, one typically regards the detector as an abstract structure of measurement surfaces or volumes that generally follow a layering structure. DDRec contains an API that provides this information for reconstruction algorithms, thereby decoupling the details of the actual simulation model used from the reconstruction code. The DDRec API provides the following functionality:

- description of measurement surfaces with coordinate systems for track finding and fitting
- description of non-active surfaces with material properties in order to take effects of multiple scattering and energy loss into account
- conversion of cellIDs assigned to simulated tracker and calorimeter hits to positions of readout cells and vice versa
- access to a list of materials between any two points inside the world volume of the detector
- access to the materials at any given point or along a straight line between two points
- averaged material properties for a list of materials
- computation of radiation and interaction lengths for detector layers, modules or arbitrary sections through the detector

In this manual we describe the different classes in DDRec and how they should be used in the detector geometry constructors as well as in the reconstruction code.

Doxygen code documentation

Please also refer to the code documentation that is created with doxygen for more details on the classes and their members. This documentation can be build with:

```
cmake -D INSTALL_DOC=ON ${path_to_dd4hep_source}
make install
```

and will then be available at

```
${DD4hepINSTALL}/doc/html/index.html
```

2 Surfaces

For fitting the trajectories of charged particle tracks one generally needs to know the measurement surfaces on which the hits where deposited. Additionally the material properties along the trajectory need to be known in order to correct for effects from multiple scattering and energy loss. DDRec provides an abstract interface for surfaces and materials in the namespace *DDSurfaces* and a concrete implementation, described below, in the namespace *DDRec*. This is done in order to separate interface and implementation and allow other software tools, e.g. tracking packages to just use the interface.

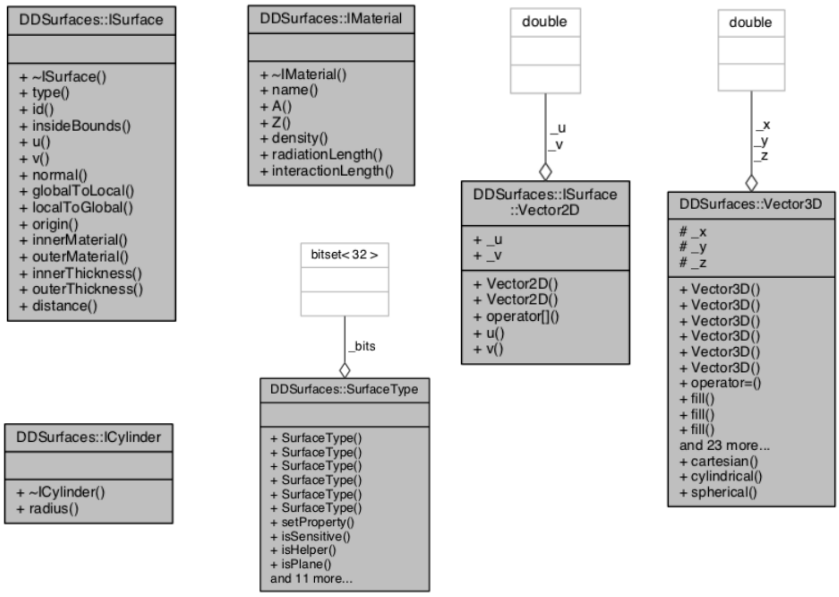

Figure 1: Classes in namespace DDSurfaces: abstract interfaces *ISurface, IMaterial, ICylinder* and helper classes *Vector3D, ISurface::Vector2D, SurfaceType.*

2.1 namespace DDSurfaces

The basic concept of surfaces in `DDRec` is expressed with the two main interfaces *ISurface* and *IMaterial*. They are shown together with helper classes in the *DDSurfaces* in figure:1 and are briefly described in the following.

ISurface: Defines the surface by means of an origin, a normal vector n and two, typically orthogonal, direction vectors u and v, where all of the vectors n,u,v might depend on the actual position on (or close to) the surface. In order to describe material properties two thicknesses are assigned to the surface - one in direction of the normal vector (outerThickness) and one in the opposite direction (innerThickness). There are two materials assigned to these thicknesses, where these materials could be averaged along the normal and thickness. The method *isInsideBounds* allows in principle to implement arbitrary bounds for the surface. Two methods allow the conversion between global 3d coordinates (on the surface) and local 2d coordinates in the coordinate system of the surface.

IMaterial: Interface to describe the relevant material properties: atomic number and weight, density and radiation- and interaction lengths. These can be real materials or averaged materials along a given direction and length (thickness assigned to the surface).

ICylinder: Simple interface for cylindrical surfaces adding the cyliner radius to a surface through multiple inheritance.

ISurface::Vector2D Helper struct inside ISurface for 2d vectors with coordinates u and v.

Vector3D: Generic 3d vector class with cartesian coordinates x,y,z that allows initialization from other 3d vector implementations or using cylindircal or spherical coordinates. Provides acces to quantities often needed, such as magnitude, transversal component, representation in non-cartesian coordinates.

SurfaceType: Helper class using an *std::bitfield¡32¿* to encode the following properties of the surface: *isCylinder, isPlane, isSensitive, isHelper* (dead material), *isParallelToZ, isOrthogonalToZ, isInvisible, isMeasurement1D.*

2.2 Surface implementation

DDRec provides classes that implement the interface defined in *DDSurfaces*. The main classes for implementing *ISurface* are shown in figure 2.

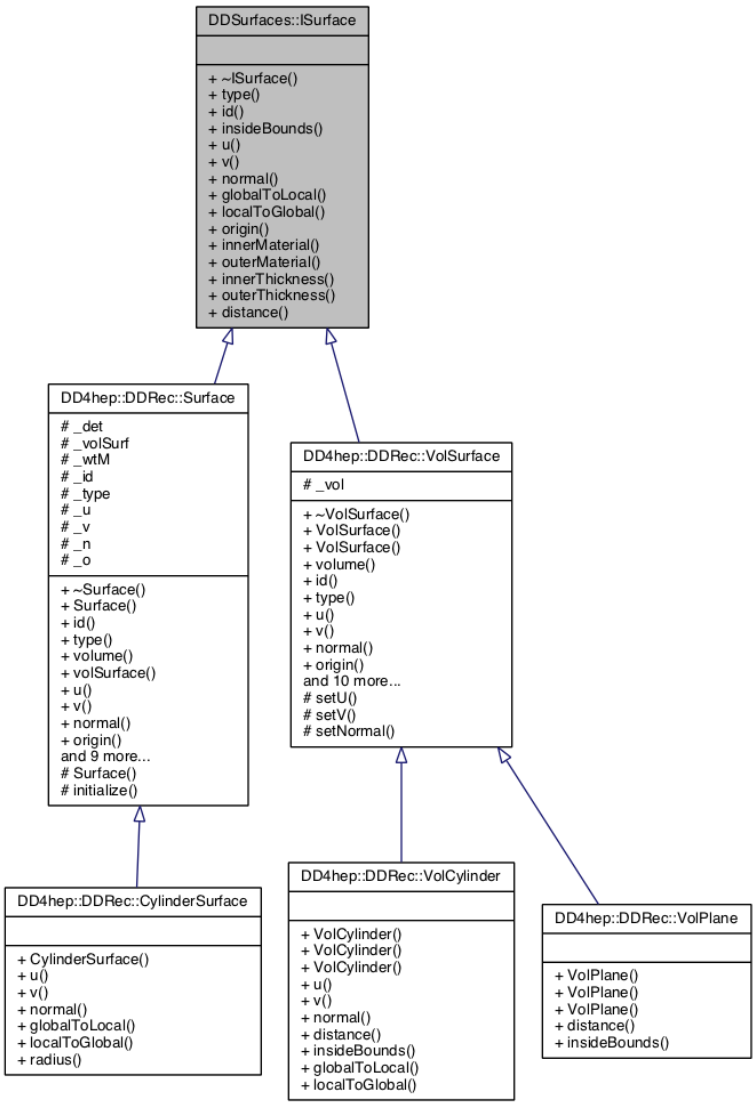

Figure 2: Class diagram with the main classes describing detector surfaces and their relations.

The implementation of the surfaces in DDRec is done in two parallel hierarchies of implementation classes. The first hierarchy is based on the *VolSurface* class which connects a surface with its surrounding volume. This volume then provide the boundaries of the surface and gives access to the local to global coordinate transformations inside the coordinate system of the volume. There are currently two concrete implementations: *VolCylinder* and *VolPlane* that can be used in the detector construction code as described in2.3. The second hierarchy is the actual implementation of the surface concept in DDRec to be used by reconstruction code as described in 2.4. It uses the extension and views concept described in the main DD4hep manual[2]. The *Surface* class has a *VolSurface* object and a *DetElement* and uses these to establish the local to global coordinate transforms, the surface boundaries and the material properties. The materials on both sides of the surface are the averaged materials along the

direction of the normal with the given thicknesses. It is thus possible to also take material effects into account for materials that lay outside of the actual volume the surface is attached to. This is in particular useful for compound materials that consist of a larger number of thin slices.

2.3 Using surfaces in geometry constructors

The surfaces that should be available in the reconstruction code, have to be assigned to their corresponding volume and detector element by the user in the detector geometry construction code. This is done by specifying the coordinate system and orientation of the surface inside the volume with vectors o,n,u and v and then instantiating one of the two types *VolCylinder* or *VolPlane* for a given volume. After the placement of the corresponding DetElement, the surface is added to the list of surfaces for this DetElement.
This is demonstrated in the following example code:

```
#include "DDRec/Surface.h"
//...
// base vectors for surfaces:
DDSurfaces::Vector3D o(0,0,0) ;
DDSurfaces::Vector3D u(1,0,0) ;
DDSurfaces::Vector3D v(0,1,0) ;
DDSurfaces::Vector3D n(0,0,1) ;

// --- loop over layers ----
// ...

  DD4hep::Geometry::Box  box( dimX/2, dimY/2, dimZ/2 ) ;
  DD4hep::Geometry::Volume vol( volumeName, lcdd.material( x_layer.materialStr() )) ;

  DD4hep::Geometry::DetElement layerDetElement( parentDetElement , "layer"+_toString(i,"_%02d") , det_id ) ;

  // add a measurement surface to the layer for every sensitive slice:

  DD4hep::DDRec::VolPlane surf( vol ,
                        DDSurfaces::SurfaceType(DDSurfaces::SurfaceType::Sensitive),
                        dimZ, dimZ,
                        u, v, n , o ) ;

  // place the layer
  DD4hep::Geometry::PlacedVolume pv = parentVol.placeVolume(  vol, layerPlacement );

  layerDetElement.setPlacement( pv ) ;

  DD4hep::DDRec::volSurfaceList( layerDetElement )->push_back( surf ) ;

// --- end loop over layers ----

```

In this example a planar (*VolPlane*) measurement (*SurfaceType(Sensitive)*) surface is attached to the box volume of a detector layer. The thickness of the surface corresponds to that of the box and is given by its half length in z (Vector3D n(0,0,1) runs along z). Thus the surface is completely contained in the box and no averaging of materials will be done, unless additional volumes are placed inside the box at a later stage. The origin of the coordinate system of the surface coinsides with that of the box (*Vector3D o(0,0,0);*) and the two measurememnt directions u,v run along the x and y axis of the box respectively.
The following code snipped shows the creation of a cylindrical surface attached to a tube volume for the inner field cage of a tpc. The radius of the cylindrical surface is given by the transversal component

of the origin vector *ocyl*. The volume *innerWallVol* is filled with air and will be later populated with slices of a compound material, thus the material properties will be averaged along the thickness of the tube.

```
1  //...
2  DD4hep::Geometry::Tube innerWallSolid(rInner ,rInner + dr_InnerWall ,dz_Wall / 2.0 ) ;
3
4  DD4hep::Geometry::Volume innerWallVol( "TPCInnerWallVol", innerWallSolid, materialAir ) ;
5
6  pv = tpc_motherLog.placeVolume( innerWallVol ) ;
7
8  DDSurfaces::Vector3D ocyl(  rInner + 0.5*dr_InnerWall , 0. , 0. ) ;
9
10 DD4hep::DDRec::VolCylinder surfI( innerWallVol ,
11                                    SurfaceType( SurfaceType::Helper ) ,
12                                    0.5*dr_InnerWall, 0.5*dr_InnerWall,
13                                    ocyl ) ;
14
15 volSurfaceList( tpc )->push_back(  surfI ) ;
16 //...
```

2.4 Using surfaces in reconstruction code

Accessing and using the surfaces in reconstruction code is very easy. There are two possibilities to access the surfaces:

- use the *DetectorSurfaces* view class to get a list of all surfaces that have been assigned to a particular *DetElement* object.
- or use the *SurfaceManager* class to get a list of all surfaces of a given *DetElement* and all its daughters.

The following code snippet uses the *SurfaceManager*, initialized with the world *DetElement*, to get a list of all surfaces defined for a given detector model. The surfaces are then printed to *std::cout* and filled into a map, using the surfaces ID as a key. For sensitive surfaces, attached to sensitive volumes, the ID will be that of the sensitve volume and thus such a map provides a very easy lookup from the hitID to its corresponding measurement surface.

```
1  // ...
2
3  DD4hep::Geometry::LCDD& lcdd = DD4hep::Geometry::LCDD::getInstance();
4
5  lcdd.fromCompact( inFile );
6
7  DD4hep::Geometry::DetElement world = lcdd.world() ;
8
9  // create a list of all surfaces in the detector:
10 DD4hep::DDRec::SurfaceManager surfMan(  world ) ;
11
12 const DD4hep::DDRec::SurfaceList& sL = surfMan.surfaceList() ;
13
14 // map of surfaces
15 std::map< long64, DD4hep::DDRec::Surface* > surfMap ;
16
17 for( DD4hep::DDRec::SurfaceList::const_iterator it = sL.begin() ; it != sL.end() ; ++it ){
18
19   DD4hep::DDRec::Surface* surf =  *it ;
```

```
20
21    std::cout << " ------------------------- "
22              << " surface: "  << *surf         << std::endl
23              << " ------------------------- "  << std::endl ;
24
25    surfMap[ surf->id() ] = surf ;
26  }
27
```

And similarily this code uses the *DetectorSurfaces* class to just access the surfaces for a particular detector element:

```
1   // ...
2
3   DD4hep::Geometry::DetElement ladderDE = lcdd.detector("VXD_layer02_ladder42") ;
4
5   // create surfaces
6   DD4hep::DDRec::DetectorSurfaces ds( ladderDE ) ;
7
8   const DD4hep::DDRec::SurfaceList& detSL = ds.surfaceList() ;
9
10  for( DD4hep::DDRec::SurfaceList::const_iterator it = detSL.begin() ; it != detSL.end() ; ++it ){
11
12    DD4hep::DDRec::Surface* surf =  *it ;
13
14    std::cout << " ------------------------- "
15              << " surface: "  << *surf         << std::endl
16              << " ------------------------- "  << std::endl ;
17  }
```

2.5 Visualizing detector surfaces

The detector surfaces and the vectors defining their coordinate system can be visualized with the program *teveDisplay* that is part of `DD4hep` . In a future version of `DD4hep` this visualization might be included in `DDEve` . Currently a full 3d view of the detector surfaces as well as a $\rho - \phi$-view and a ρ-z view are available. See figure 3.

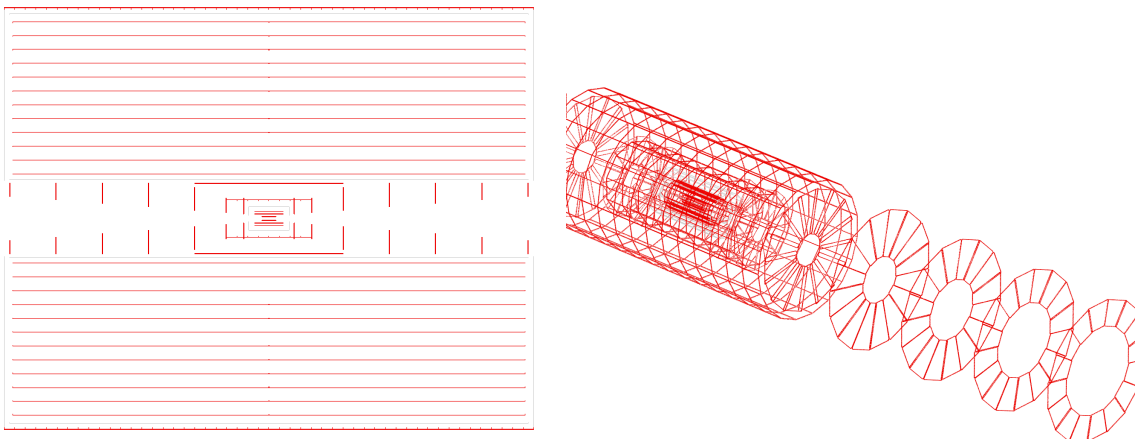

Figure 3: Example of surface visualization. Left: ρ-z view of the tracking surfaces in the ILD detector, Right: 3D view of the surfaces in the inner tracking detectors in ILD.

3 Materials

The surfaces classes described above provide a way that allows to augment a detector geometry description with a high level view on the detector that should be sufficient for most reconstruction tasks, such as pattern recognition, track fitting and calorimeter reconstruction as in a particle flow algorithm. However they require that care has been taken to assign all relevant surfaces with corresponding thicknesses to the volumes and detector elements. For cases where this is not possible or where other reconstruction geometries should be instantiated, `DDRec` provides the possibility to access the materials at any given point in the world volume of the detector or to retrieve a list of materials along a straight line between any two points.
This is done with the class *MaterialManager*, which also allows to create an averaged material for a list of materials (*MaterialVector*). The usage of this class is simple and best demonstrated with an example:

```
1
2  DD4hep::Geometry::LCDD& lcdd = DD4hep::Geometry::LCDD::getInstance();
3
4  lcdd.fromCompact( inFile );
5
6  DDSurfaces::Vector3D p0( x0, y0, z0 ) ;
7  DDSurfaces::Vector3D p1( x1, y1, z1 ) ;
8
9  DD4hep::DDRec::MaterialManager matMgr ;
10
11 const DD4hep::DDRec::MaterialVec& materials = matMgr.materialsBetween( p0 , p1  ) ;
12
13 std::cout  << std::endl
14            << " #######  materials between the two  points : "
15            << p0 << "*cm  and " << p1 << "*cm :  "
16            << std::endl ;
17
18 double sum_x0 = 0 ;
19 double sum_lambda = 0 ;
20 double path_length = 0 ;
21 for( unsigned i=0,n=materials.size();i<n;++i){
22
23   DD4hep::DDRec::Material mat =  materials[i].first  ;
24   double length = materials[i].second  ;
25
26   double nx0 = length / mat.radLength()  ;
27   sum_x0 += nx0 ;
28
29   double nLambda = length / mat.intLength()  ;
30   sum_lambda += nLambda ;
31
32   path_length += length ;
33
34   std::cout << "      "               << mat
35             << " thickness: "         << length
36             << " path_length:"        << path_length
37             << " integrated_X0: "     << sum_x0
38             << " integrated_lambda: " << sum_lambda
39             <<  std::endl ;
40 }
41
```

Creation of an averaged material:

```
// ...
const DD4hep::DDRec::MaterialVec& materials = matMgr.materialsBetween( p0 , p1  ) ;

const DD4hep::DDRec::MaterialData& avMat = matMgr.createAveragedMaterial( materials ) ;

std::cout << " averaged Material : " << " Z: " << avMat.Z() << " A: " << avMat.A()
          << " densitiy: "           << avMat.density()
          << " radiationLength: "    << avMat.radiationLength()
          << " interactionLength: "  << avMat.interactionLength()
          << std::endl ;
```

There is a utility program *print_materials* that can be used to debug detector geometries:

```
$ print_materials
usage: print_materials compact.xml x0 y0 z0 x1 y1 z1
       -> prints the materials on a straight line between the two given points ( unit is cm)
```

Note: accessing the materials using the *MaterialManager* is a rather costly operation and should only be done at the initialization phase of a reconstruction program for caching material properties !

4 IDDecoder

Sensitive volumes in a DD4hep geometry model are assigned a unique volume-ID. This ID is then used by the corresponding *DDSegmentation* object to create a unique cellID for tracker and calorimeter hits, allowing to uniquely match hits to their sensitive volumes and also to their *DetElements* if they have been defined appropriately. During reconstruction tasks, including digitization of simulated hits, one often needs to convert between a cellID assigned to the hit and the position of the corresponding detector cell. For example one could write out simulated calorimeter hits without position information in order to save disk space and retrieve the position information based on the cellID. Another application might be a clustering algorithm where one looks for hits in the neighbor cells of a given hit.
The functionality to do this is provided by the *IDDecoder* class with the following interface:

```
class IDDecoder {
public:
   /// Default constructor using the name of the corresponding readout collection
  IDDecoder(const std::string& collectionName);

   /// Default constructor using a readout object
  IDDecoder(const Geometry::Readout& readout);

   /// Destructor
  virtual ~IDDecoder();

   /// Returns the cell ID from the local position in the given volume ID.
  CellID cellIDFromLocal(const Geometry::Position& local, const VolumeID volumeID) const;

   /// Returns the global cell ID from a given global position
  CellID cellID(const Geometry::Position& global) const;

   /// Returns the global position from a given cell ID
  Geometry::Position position(const CellID& cellID) const;

   /// Returns the local position from a given cell ID
  Geometry::Position localPosition(const CellID& cellID) const;

```

```
24      /// Returns the volume ID of a given cell ID
25     VolumeID volumeID(const CellID& cellID) const;
26
27      /// Returns the volume ID of a given global position
28     VolumeID volumeID(const Geometry::Position& global) const;
29
30      /// Returns the placement for a given cell ID
31     Geometry::PlacedVolume placement(const CellID& cellID) const;
32
33      /// Returns the placement for a given global position
34     Geometry::PlacedVolume placement(const Geometry::Position& global) const;
35
36      /// Returns the subdetector for a given cell ID
37     Geometry::DetElement subDetector(const CellID& cellID) const;
38
39      /// Returns the subdetector for a given global position
40     Geometry::DetElement subDetector(const Geometry::Position& global) const;
41
42      /// Returns the closest detector element in the hierarchy for a given cell ID
43     Geometry::DetElement detectorElement(const CellID& cellID) const;
44
45      /// Returns the closest detector element in the hierarchy for a given global position
46     Geometry::DetElement detectorElement(const Geometry::Position& global) const;
47
48      /// Calculates the neighbours of the given cell ID and adds them to the list of neighbours
49     void neighbours(const CellID& cellID, std::set<CellID>& neighbours) const;
50
51      /// Checks if the given cell IDs are neighbours
52     bool areNeighbours(const CellID& cellID, const CellID& otherCellID) const;
53 }
```

5 Detectors

To be done ...

References

[1] M. Frank et al, "DD4hep: A Detector Description Toolkit for High Energy Physics Experiments", International Conference on Computing in High Energy and Nuclear Physics (CHEP 2013), Amsterdam, Netherlands, 2013, proceedings.

[2] M. Frank et al, "DD4hep: A Detector Description Toolkit for High Energy Physics Experiments", Users manual (DD4hepManual.pdf).

[3] R.Brun, A.Gheata, M.Gheata, "The ROOT geometry package", Nuclear Instruments and Methods **A** 502 (2003) 676-680.

[4] S. Agostinelli et al., "Geant4 - A Simulation Toolkit", Nuclear Instruments and Methods **A** 506 (2003) 250-303.