Development of a Geant4 Solid for Stereo Mini-jet Cells in a Cylindrical Drift Chamber

Kotoyo Hoshina[a], *Keisuke Fujii*[a1], and *Osamu Nitoh*[b],

[a] High Energy Accelerator Research Organization(KEK), Tsukuba, 305-0801, Japan
[b] Tokyo University of Agriculture and Technology, Tokyo 184-8588, Japan

Abstract

Stereo mini-jet cells will be indispensable components of a future e^+e^- linear collider central tracker such as JLC-CDC. There is, however, no official Geant4 solid available at present to describe such geometrical objects, which had been a major obstacle for us to develop a full Geant4-based simulator with stereo cells built in. We have thus extended Geant4 to include a new solid (`TwistedTubs`), which consists of three kinds of surfaces: two end planes, inner and outer hyperboloidal surfaces, and two so-called twisted surfaces that make slant and twisted ϕ-boundaries. Design philosophy and its realization in the Geant4 framework are described together with algorithmic details. We have implemented stereo cells with the new solid, and tested them using geantinos and Pythia events ($e^+e^- \to ZH$ at $\sqrt{s} = 350$ GeV). The performance was found reasonable: the stereo cells consumed only 25% more CPU time than ordinary axial cells.

---

[1]Corresponding authhor.
E-Mail address: fujiik@jlcuxf.kek.jp
TEL: +8-298-64-5373
FAX: +8-298-64-2580

1   Introduction

Experiments at a future linear e^+e^- collider such as JLC[1] will open up a novel possibility to reconstruct all the final states in terms of fundamental particles (leptons, quarks, and gauge bosons). This involves identification of heavy unstable particles such as W, Z, and t through jet invariant-mass measurements. High resolution energy flow measurements will thus be crucial, necessitating high resolution tracking and calorimetry as well as good track-cluster matching to avoid double counting. A large cylindrical drift chamber with small jet cells (JLC-CDC[2]) is our choice for a candidate central tracking device to fulfill these requirements. Good track-cluster matching requires, however, small track extrapolation errors, which in turn demand high r-ϕ and z resolutions. The latter forces us to introduce cells consisting of stereo wires, since the z resolution in charge division or time difference readout is typically 1 % of the wire length or worse, which is a few cm in the linear collider use.

We have already studied and published hardware aspects of common problems in designing stereo-wire geometry for a long cylindrical drift chamber with small jet cells[4]. In order to finalize the chamber design so as to achieve the best attainable energy flow resolution, however, we need to carefully optimize the layout of axial and stereo cells through detailed Monte Carlo simulations. Considering the recent advance of object-oriented technology in high energy physics software development, we have thus started the development of a full Monte Carlo simulator called JUPITER[3] based on Geant4[5].

The axial layers were easy to implement, using a standard Geant4 solid called `G4Tubs`, which is a ϕ segment of a cylinder. On the other hand, there is currently no official Geant4 solid available to describe stereo cells, which had been a major obstacle for us to install stereo cells into the full simulator. We have thus extended Geant4 to include a new solid (`TwistedTubs`), which comprises three kinds of bounding surfaces: two end planes, inner and outer hyperboloidal surfaces, and two so-called twisted surfaces that make slant and twisted ϕ boundaries. Although `TwistedTubs` was developed under the JUPITER environment, it is actually a general purpose Geant4 extension. This paper describes the design philosophy of `TwistedTubs`, its realization in the Geant4 framework, and algorithmic details, together with results of its performance test.

The paper is organized as follows. We begin with a brief account of geometrical parameters that determine the configuration of a stereo cell, and then review the basic procedure to add a new solid to Geant4. The subsequent two sections are devoted to descriptions of design philosophy, implementations, and algorithmic details of `TwistedTubs`, which is followed by presentation of test results and discussions. Finally section 8 summarizes our achievement and concludes this paper.

2   Geometry of Stereo Mini-jet Cell

In this section we introduce, following the convention of [4], the stereo-geometrical parameters that will be needed in subsequent sections.

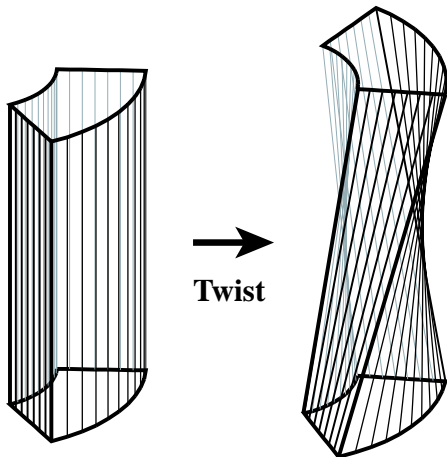Consider a cylindrical tube consisting of two cylindrical layers of axial field-shaping

Figure 1: An exaggerated illustration of a stereo cell as formed by twisting an axial cell.
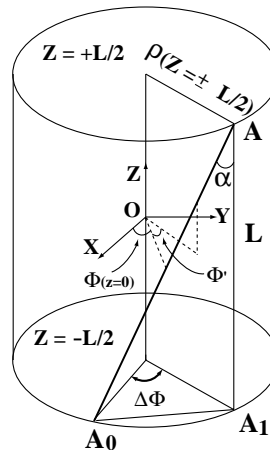

Figure 2: 3-dimensional view of a single stereo wire.

wires strung across two disc-shaped end plates at some radii, ρ_{in} and ρ_{out}. An axial mini-jet cell is a segment cut out from this cylindrical tube by ϕ boundaries formed by cathode wires. As illustrated in Fig. 1, twisting one end plate by a twist angle $\Delta\phi$ with the other end fixed turns the axial mini-jet cell into a stereo mini-jet cell. Inspection of the figure tells us that the stereo cell consists of three pairs of three kinds of surfaces: a pair of hyperboloidal surfaces setting inner and outer radial boundaries, another pair of so-called twisted surfaces making left and right azimuthal boundaries, and yet another pair of fan-shaped flat surfaces closing the positive and negative z ends of the cell.

By construction, any of the four side walls of the stereo cell can be regarded as a locus of a stereo wire sweeping through the surface. The geometry of a given side surface can thus be completely determined by the equation for a single representative wire chosen from the stereo wires forming that surface. As depicted in Fig. 2, the representative stereo wire is uniquely specified by the radius at the ends $\rho(z = \pm L/2)$ or that at the center $\rho_c \equiv \rho(z = 0)$, the projected wire length L to the chamber axis, and the twist angle $\Delta\phi$. Notice that $\Delta\phi$ is signed and measured from A_0 to A_1. The stereo angle α, which is defined as an angle between AA_0 and AA_1, is also signed, having the same sign as $\Delta\phi$.

The stereo angle α can now be written in terms of $\rho(z = \pm L/2)$, $\Delta\phi$, and the projected wire length (L):

$$\alpha = \tan^{-1}\left(\frac{2\rho(z = \pm L/2)}{L}\sin\left(\frac{\Delta\phi}{2}\right)\right) \tag{2.1}$$

It is obvious from Fig. 2 that both the azimuthal angle and the radial position of the stereo wire become z-dependent:

$$\begin{aligned}
\phi(z) &= \phi(z = 0) + \phi' \\
&= \phi(z = 0) + \tan^{-1}\left[\left(\frac{2z}{L}\right)\tan\left(\frac{\Delta\phi}{2}\right)\right]
\end{aligned} \tag{2.2}$$

2

$$\rho(z) = \sqrt{(\rho(z=0))^2 + (z \tan \alpha)^2}$$

$$= \sqrt{(\rho(z=\pm L/2))^2 + (z^2 - (L/2)^2) \tan^2 \alpha}, \qquad (2.3)$$

where z is measured from the middle of the chamber along the chamber axis as Fig. 3 indicates.

Now we consider a hyperboloidal surface obtained as a locus of the straight line given by Eqs. 2.2 and 2.3 by sweeping $\phi(z=0)$ from ϕ_{left} to ϕ_{right} with $\rho(z=0)$ fixed. Let the outward normal to the hyperboloidal surface at a surface point $\mathbf{x} = (x, y, z)$ be $\mathbf{n}$. Apparently the outward normal $\mathbf{n}$ is in the ρ-z plane containing $\mathbf{x}$ and is perpendicular to the tangential vector thereat. Eq. 2.3 tells us that the cross section of the hyperboloidal surface by the ρ-z plane becomes a hyperbola given by

$$\begin{cases} \rho = \sqrt{(\rho(z=0))^2 + (z \tan \alpha)^2} \\ z = z. \end{cases}$$

The tangential vector we need is thus obtained by differentiating this equation with respect to z. Normalizing it to $|\mathbf{x}|$, we obtain

$$\text{the tangential vector} = (z \tan^2 \alpha, \ \rho), \qquad (2.4)$$

which leads to the outward normal in the ρ-z plane: $\mathbf{n}_\rho = (\rho, -z \tan^2 \alpha)$. Recalling that ρ in the ρ-z plane corresponds to (x, y) in the x-y plane, we now obtain

$$\mathbf{n} = (x, y, -z \ \tan^2 \alpha). \qquad (2.5)$$

The inward normal is anti-parallel with this and can be obtained by simply changing the signs of all the components.

On the contrary to the hyperboloidal surface, a twisted surface is a locus of the straight line given by Eqs. 2.2 and 2.3, when $\rho_c \equiv \rho(z=0)$ is swept from $\rho_{c,in} \equiv \sqrt{\rho_{in}^2 - ((L/2) \tan \alpha)^2}$ to $\rho_{c,out} \equiv \sqrt{\rho_{out}^2 - ((L/2) \tan \alpha)^2}$ with $\phi(z=0)$ fixed, where ρ_{in} and ρ_{out} are the inner and outer radii at the end planes as defined before. By construction, any point on such a twisted surface can be specified by two parameters (ρ_c and z) in a local coordinate system for which x-axis is chosen such that $\phi(z=0) = 0$, as depicted in Fig. 4:

$$\begin{cases} x = \rho_c \\ y = \rho_c \tan \phi' = \rho_c \kappa z \\ z = z, \end{cases} \qquad (2.6)$$

where $\tan \phi'$ is given, through Eq. 2.2, by

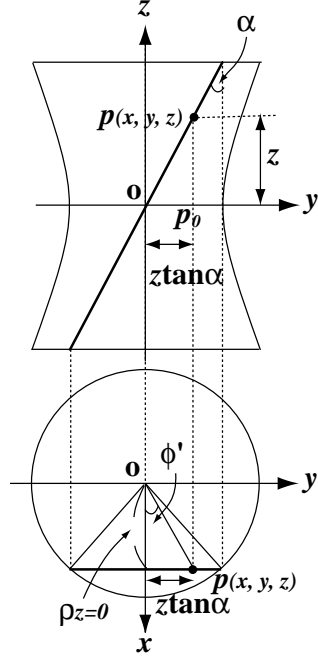$$\tan \phi' = \frac{2z}{L} \cdot \tan \left(\frac{\Delta \phi}{2} \right) \qquad (2.7)$$

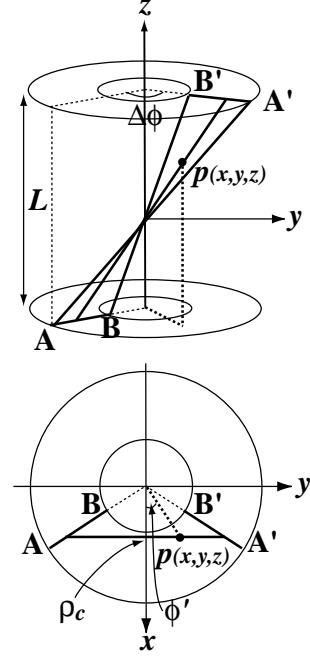Figure 3: Hyperboloidal surface and a typical line (wire) contained in it.

Figure 4: Similar picture to Fig. 3 for a twisted surface.

while κ is defined to be

$$\kappa \equiv \frac{2}{L} \cdot \tan\left(\frac{\Delta\phi}{2}\right). \tag{2.8}$$

Notice that the $\Delta\phi$ must be smaller than π, implying that $\rho_c > 0$ in this coordinate system. Eq. 2.6 shows that, in addition to the lines corresponding to cathode wires, the twisted surface contains another kind of straight lines that appear as the sections of different z-slices of the twisted surface. Each of these lines corresponds to a ϕ boundary of a z-slice of a stereo cell and passes through the z-axis. At the point $\mathbf{x}$ on the surface, a pair of these two kinds of straight lines passing through it divides the tangential directions at that point into four regions: the surface turns from convex to concave or vice versa as crossing the boundaries. In other words, the twisted surface has a saddle shape[2].

The two kinds of straight lines contained in the twisted surface are useful when we construct a normal to the twisted surface as we shall see below. In general a normal to the surface at a given surface point $\mathbf{x} = (x, y, z)$ can be formed as a vector product of two linearly independent tangential vectors at that point. In our case such tangential vectors

---

[2]It is well known that the hyperboloidal surface has also a saddle shape. In this case, the two straight lines over which the curvature changes its sign are in the directions of stereo angles of $\pm\alpha$.

can be readily obtained by partially differentiating Eq. 2.6 with respect to $x = \rho_c$ and z:

$$\mathbf{e}_z \equiv \begin{pmatrix} \frac{\partial x}{\partial z} \\ \frac{\partial y}{\partial z} \\ \frac{\partial z}{\partial z} \\ \frac{\partial z}{\partial z} \end{pmatrix} = \begin{pmatrix} 0 \\ \rho_c \kappa \\ 1 \end{pmatrix}$$

$$\mathbf{e}_x \equiv \begin{pmatrix} \frac{\partial x}{\partial x} \\ \frac{\partial y}{\partial x} \\ \frac{\partial z}{\partial x} \end{pmatrix} = \begin{pmatrix} 1 \\ \kappa z \\ 0 \end{pmatrix}.$$

(2.9)

The first of the above two tangential vectors, $\mathbf{e}_z$, can virtually be identified as one of the cathode wires forming the twisted surface. On the other hand, the second one, $\mathbf{e}_x$ can be viewed as a ϕ boundary of a z-slice of a stereo cell. Using Eq. 2.9, we can now easily form the surface normal at the surface point $\mathbf{x}$ by taking the vector product of these two tangential vectors.

Armed with the equations given in this section, we shall, in the next two sections, prepare classes to represent the three kinds of surfaces: `J4HyperboloidalSurface`s, `J4TwistedSurface`s, and `J4FlatSurface`s, and assemble them to form a Geant4 solid class (`J4TwistedTubs`) to describe our stereo mini-jet cells of JLC-CDC.

3   How to Add a New Solid to Geant4

According to Geant4 User's Guide for Toolkit Developers[5], every Geant4 solid has to inherit from a base class called `G4VSolid`. This base class has the following pure virtual functions:

<u>G4VSolid</u>

 `G4double DistanceToIn(const G4ThreeVector &p)`
     to calculate the minimal (or shorter) distance to the solid from an outside point (`p`),

 `G4double DistanceToIn(const G4ThreeVector &p, const G4ThreeVector &v)`
     to calculate the exact distance from the outside point `p` in the direction of a velocity vector `v` to the solid,

 `G4double DistanceToOut(const G4ThreeVector &p)`
     to calculate the minimal (or shorter) distance to the solid from an inside point `p`,

 `G4double DistanceToOut(const G4ThreeVector &p, const G4ThreeVector &v,`
     `const G4bool calcNorm=FALSE, G4bool *validNorm=0, G4ThreeVector *n)`
     to calculate the exact distance from the inside point `p` to the solid along a velocity vector `v`,

 `G4ThreeVector SurfaceNormal(const G4ThreeVector &p)`
     which returns the outward unit normal at a surface point `p` (or, if `p` is not on the surface, at the surface point that is the closest from `p`),

5

EInside Inside(const G4ThreeVector &p)
> to judge whether a space point p is inside, or outside, or on the surface of the solid and returns kInside, or kOutside, or kSurface, accordingly,

G4bool CalculateExtent(const EAxis pAxis, const G4VoxelLimits &pVoxelLimit, const G4AffineTransform &pTransform, G4double &pMin, G4double &pMax) const
> to calculate minimum and maximum extents (pMin and pMax) of the solid in the direction of a given coordinate axis pAxis within the limits specified by pVoxelLimit under an Affine transformation pTransform, and

G4GeometryType GetEntityType() const
> to identify geometry type of the solid (necessary for persistency and STEP interface, but otherwise unused).

The first five involve surfaces that bound the solid and can be calculated on a surface-by-surface basis. On the other hand the remaining three require information of the solid as a whole. Any new user defined solid thus has to be equipped with these functions. Geant4 also provides BREPS[3] classes for the purpose of constructing a solid from boundary surfaces. BREPS are, however, primarily designed to facilitate implementation of a solid with a complicated shape through interface like STEP to a CAD system. Since the stereo cell geometry can be handled analytically as explained in the subsequent sections, we decided to develop a new dedicated solid class for it.

4 Design of J4TwistedTubs

Before coding our new solid "J4TwistedTubs", we set the following guide line to fulfill the required functionality discussed in the last section:

1. **J4TwistedTubs** should be implemented as a collection of bounding surfaces. It will thus have to have, as its data member, an array of pointers to the instances of the corresponding surface classes.

2. All of these surface classes must inherit from an abstract class named "J4VSurface", which carries generic information on each surface and sets basic interfaces to all the surface classes that inherit from it.

3. Each surface is orientable and should know which side is outside, but the calculation of the distance to the surface from a point p should not depend on whether the point is inside or outside of the solid, since the distance should only depend on the shape of the surface in question.

4. It is then the role of "DistanceToIn" or "DistanceToOut" methods of its parent J4VSurface class to judge the distance depending on the context such as the angle between the surface normal and the velocity vector v. This way, we can avoid repetition of the same code.

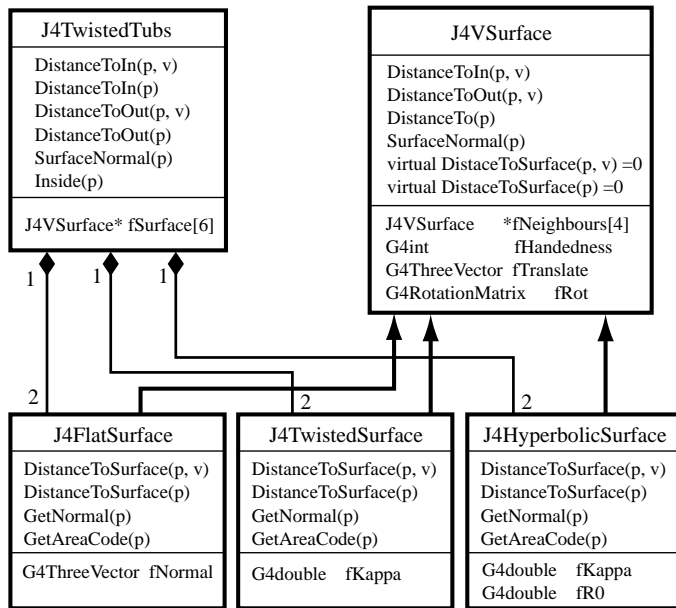---

[3]Boundary REPresented Solid.

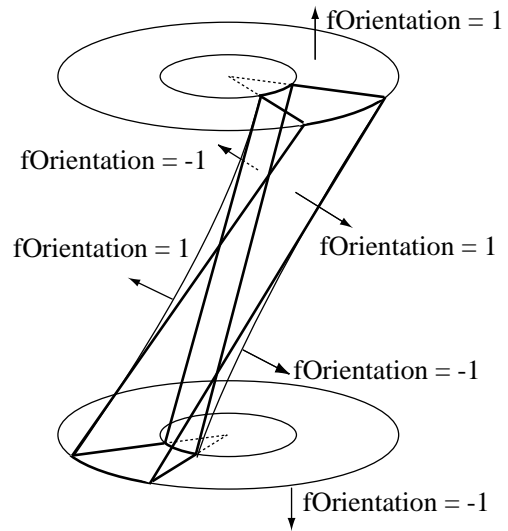Figure 5: A class diagram in UML of J4TwistedTubs

Figure 6: Values of fOrientation data member of J4TwistedTubs

5. All J4TwistedTubs has to do will then be to simply invoke "DistanceToIn" or "DistanceToOut" for each of the bounding surfaces and choose the best[4]. The surface normal can then be calculated by calling the corresponding GetNormal method for the selected surface.

As described in section 2, J4TwistedTubs consists of three pairs of three kinds of surfaces. We named these three kinds of bounding surfaces J4HyperboloidalSurface, J4TwistedSurface, and J4FlatSurface, all of which are descendants of J4VSurface. Interrelation of these classes is illustrated in Fig. 5 as a class diagram. On the other hand, Fig. 6 shows the outward direction of each surface as stored in a data member called fOrientation.

As sketched above, the abstract base class J4VSurface has the following methods:

J4VSurface

G4int DistanceToSurface(const G4ThreeVector &p, G4ThreeVector *xx,
    G4double *distance, G4int *areacode) = 0
    which calculates the distance (*distance) from a point p to the surface as well as
    the point (*xx) of the closest approach to the surface and an *areacode given by
    GetAreaCode explained below, and then returns the number of the points of closest
    approach that is always 1.

---

[4]The best is usually the one that returned the shortest distance. One exception is the case in which a particle is coming into the solid from a corner or edge of the solid.

`G4int DistanceToSurface(const G4ThreeVector &p, const G4ThreeVector &v,`
`G4ThreeVector xx[], G4double distance[], G4int areacode[],`
`G4bool isvalid[], EValidate validate) = 0`
which calculates the distance(s) (`distance[]`) along the velocity vector `v` to the
surface from the point `p`, the intersection(s) (`xx[]`) of the particle track with the
surface, and the areacode(s) (`areacode[]`) of intersection(s), and then returns the
number of intersections. If so flagged by `validate`, it does boundary check for each
of the candidate intersection(s) and passes the test result through `isvalid[]`. The
return value then becomes the number of valid intersection(s).

`G4ThreeVector GetNormal(const G4ThreeVector &xx, G4bool isglobal) = 0`
which calculates and returns the normal at a surface point `xx`. The surface point
`xx` and the normal to be returned are assumed to be given in the coordinate system
of its mother solid or in the local frame of the surface in question, depending on
whether `isglobal` is `TRUE` or not, respectively.

`G4Int GetAreaCode(const G4ThreeVector &xx, G4bool withTol) = 0`
which decides whether the intersection `xx` given by `DistanceToSurface(p,v)` is
on a boundary, or at a corner, or inside or outside of the surface with or without
tolerance as flagged by `withTol`, and then returns a corresponding area code.

`G4double DistanceToIn(const G4ThreeVector &p, const G4ThreeVector &v,`
`G4ThreeVector &xx)`
which invokes `DistanceToSurface(p,v)` of its descendant concrete class and judges
the validity of each of the resultant intersection(s) and the corresponding distance(s),
examining the sign of the distance, the angle of the particle velocity `v` to the surface
normal at the intersection point on the surface. If the intersection is on a boundary
or at a corner of the surface, it also checks the angle to the surface normal(s) of
the adjacent surface(s). It then returns the distance if valid, or infinity otherwise,
together with the valid intersection `xx`, if any.

`G4double DistanceToOut(const G4ThreeVector &p, const G4ThreeVector &v,`
`G4ThreeVector &xx)`
which invokes `DistanceToSurface(p)` of its descendant concrete class and judges
the validity of each of the resultant intersections(s) and the corresponding dis-
tance(s), taking into account the sign of the distance and the angle between the
surface normal and the particle velocity `v`, and then returns the distance to the
caller if valid, or infinity otherwise, with the valid intersection passed through `xx`,
if any.

`G4double DistanceTo(const G4ThreeVector &p)`
which just passes the return value of `DistanceToSurface(p)` without any additional
judgment. This functions is used in `DistanceToIn(p)` or `DistanceToOut(p)` of
`J4TwistedTubs`.

Notice that the first four of these member functions are pure virtual and should be implemented in its derived concrete classes. Since we have already explained how to construct a surface normal in section 2 and since description of `GetAreaCode` would only involve technical details, we will concentrate, in what follows, on how we implemented the base class `J4VSurface` and the first two that are the `DistanceToSurface` functions with and without the particle velocity, for each of the three derived surface classes: `J4HyperboloidalSurface`, `J4TwistedSurface`, and `J4FlatSurface`.

5   Algorithm and Implementation

5.a   J4VSurface

Being the base class for all the surface classes that make `J4TwistedTubs`, the `J4VSurface` class plays the following four major roles: (1) it carries data members to store generic information on a surface, (2) it standardizes interface for generic methods to be implemented by its descendant, (3) it provides tools commonly used by any derived surface class, and (4) it steers the distance calculation in `DistanceToIn` and `DistanceToOut`, by filtering the return values from `DistanceToSurface` of its derived class. We now describe these points below in some more detail.

5.a.1   Generic Data Members

`J4VSurface` has the following *protected* data members to store generic information on its derived surface class.

- Two axes of the surface

- Maximum and minimum limits of the surface along the two axes

- The rotation matrix and the translation vector to transform the surface from its mother solid (`J4TwistedTubs`) coordinate system to the local frame attached to the surface

- The last value of the distance to the surface from a point `p`

- The last value of the surface normal

On the other hand, the next five data members are *private*, since they are basically used internally by `J4VSurface`:

- Four pointers to neighboring surfaces

- Four positions of the corner points

- An instance of a local class that stores information on the four boundary line segments

- The name of the surface

- A pointer to its mother solid (`J4TwistedTubs`)

Notice that `J4VSurface` currently assumes the existence of four corners and consequently four boundary line segments for its derived surface class. It is, however, relatively easy to extend it to support variable number of neighboring surfaces.

5.a.2 Common Utility Functions

`G4double DistanceToLine(const G4ThreeVector &p, const G4ThreeVector &x0, const G4ThreeVector &d, G4ThreeVector &xx)`
> to calculate the distance from a point `p` to a line specified by a reference point `x0` on it and the line direction vector `d`, together with the point `xx` of the closest approach on the line. It then returns the distance.

`G4double DistanceToPlane(const G4ThreeVector &p, const G4ThreeVector &x0, const G4ThreeVector &n0, G4ThreeVector &xx)`
> to calculate the distance from `p` to a plane specified by a normal `n0` and a reference point `x0` on it, together with the point `xx` of the closest approach to the surface. It then returns the distance.

`G4double DistanceToPlane(const G4ThreeVector &p, const G4ThreeVector &x0, const G4ThreeVector &t1, const G4ThreeVector &t2, G4ThreeVector &xx, G4ThreeVector &n)`
> to calculate the distance from `p` to a plane and the point `xx` of the closest approach to the plane, where the plane is specified by a reference point `x0` and two linearly independent vectors `t1` and `t2` on the plane. It then returns the distance.

`G4double DistanceToBoundary(G4int areacode, G4ThreeVector &xx, const G4ThreeVector &p)`
> to calculate the distance from `p` to a boundary specified by `areacode`, and the point `xx` of the closest approach to the boundary. Its return value is the distance.

`G4int AmIOnLeftSide(const G4ThreeVector &me, const G4ThreeVector &ref, G4bool withTol)`
> to judge relation between two vectors `me` and `ref` by comparing their ϕ angles. If `me` lies clearly on the left side of `ref`, namely `me` has a ϕ angle smaller than that of `ref`, it returns 1. If `withTol` is `TRUE` and if `me` lies within $\pm 0.5 \times$`kAngTolerance` ($= 0.5 \times 10^{-9}$) radians of `ref`, it returns 0. Otherwise, it returns -1.

5.a.3 DistanceToIn and DistanceToOut

DistanceToIn(p,v)

On invocation, `DistanceToIn(p,v)`[5] proceeds according to the following algorithm:

---

[5]In what follows, we often leave out arguments which are irrelevant in the explanation of algorithmic

1. Initialize a temporary data buffer to store the current best distance and intersection to `kInfinity`.

2. Then call `DistanceToSurface(p,v)` and get a set of distance(s) (D) and intersection(s) ($\mathbf{X}$): the number of intersections is at most 2 in our case.

3. If a candidate D and its corresponding $\mathbf{X}$ satisfy the following conditions, discard this candidate and move on to the next, if any:

    - D is negative, or
    - the scalar product between the outward surface normal at $\mathbf{X}$ and the particle velocity $\mathbf{v}$ is positive, or
    - $\mathbf{X}$ is outside of the surface.

4. If $\mathbf{X}$ is inside of the surface and if D is smaller than the last value stored in the temporary data buffer defined in step 1, update the stored D and $\mathbf{X}$ and move on to the next intersection, if any.

5. If $\mathbf{X}$ is on the boundary of the surface within $\pm 0.5 \times$`kCarTolerance`($= 0.5 \times 10^{-9}$), we need to judge whether the particle is really entering the solid or just grazing the solid and flying away. In this case, invoke the `DistanceToSurface(p,v)` method of the (two) neighboring surface(s) (two if the intersection is at a corner, one otherwise) and get a set of distance(s) (D') and intersection(s) ($\mathbf{X}'$), and if $\mathbf{X}'$ satisfies the following conditions, exit `DistanceToIn` immediately by returning `kInfinity` as the distance to the surface originally in question.

    - If $\mathbf{X}'$ is inside of a neighboring surface[6], or
    - if the $\mathbf{X}'$ is on the same boundary (or at the same corner) as $\mathbf{X}$ and if the scalar product of the outward surface normal at $\mathbf{X}'$ and the particle velocity $\mathbf{v}$ is positive[7].

6. If $\mathbf{X}$ survives all the tests for all the neighboring surfaces, check whether the current D is smaller than that stored in the temporary data buffer, and if so, update the stored D and $\mathbf{X}$.

7. Return the stored distance in the temporary data buffer.

The flow chart of `DistanceToIn` is shown in Fig. 7.

DistanceToOut(p,v)

On the other hand, the algorithm for `DistanceToOut(p,v)` is less complicated as described below:

---

aspects of each function, as long as there is no possibility of confusion.

[6] In this case this neighboring surface should be the one through which the particle enters the solid.

[7] A particle cannot come into a solid from the edge (or corner) if the scalar product is positive for any of the surfaces forming the edge (or corner).
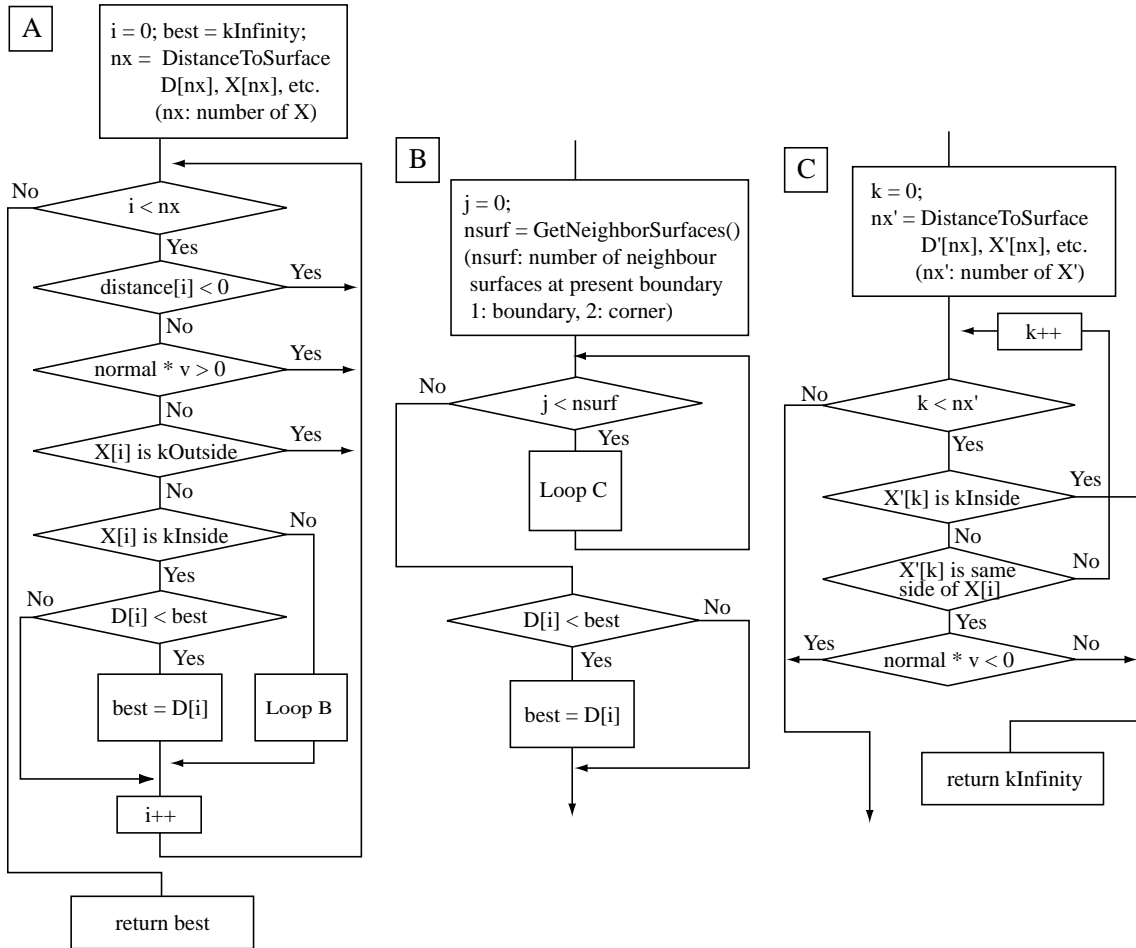
Figure 7: Flow chart of DistanceToIn

1. Initialize a temporary data buffer to store the current best distance and intersection to `kInfinity`.

2. Call `DistanceToSurface(p,v)` and get a set of distance(s) (D) and intersection(s) ($\mathbf{X}$): the number of intersections is at most 2 in our case.

3. If D and $\mathbf{X}$ meet the following conditions, reject this candidate and move on to the next:

   - D is negative, or
   - the scalar product of the outward surface normal at $\mathbf{X}$ and the particle velocity v is negative, or
   - $\mathbf{X}$ is outside of the surface.

4. If D is smaller than the stored value in the temporary buffer, update the stored D and $\mathbf{X}$.

5. Return the best distance stored in the temporary buffer.

DistanceTo(p)

`DistanceTo(p)`, which is called by the `DistanceToIn(p)` and `DistaceToOut(p)` methods of `J4TwistedTubs`, just returns `DistanceToSurface(p)` which is implemented in the derived surface class.

5.b   J4HyperboloidalSurface

Geant4 provides a solid class called `G4Hype` to represent a hyperboloidal volume with its symmetry axis parallel to the z-axis. Since `J4HyperboloidalSurface` follows the basic algorithm to calculate the inward or outward normal and the minimal distance to a hyperboloidal surface, we only sketch it here.

5.b.1   DistanceToSurface(p, v)

The distance to a hyperboloidal surface along a velocity $\mathbf{v} = (v_x, v_y, v_z)$ from a point $\mathbf{p} = (p_x, p_y, p_z)$ can be obtained as follows. Define the intersection of a particle track with the hyperboloidal surface to be $\mathbf{X} = \mathbf{p} + t\mathbf{v} = (p_x + tv_x, p_y + tv_y, p_z + tv_z)$, where t, the time, equals the distance we want, provided that v is normalized to unity. From Eq. 2.2, we then have a quadratic equation for t:

$$(p_x + tv_x)^2 + (p_y + tv_y)^2 = r_{z=0}^2 + (p_z + v_z)^2 \tan^2 \alpha. \tag{5.10}$$
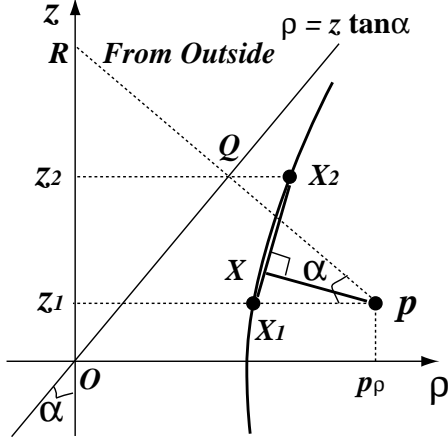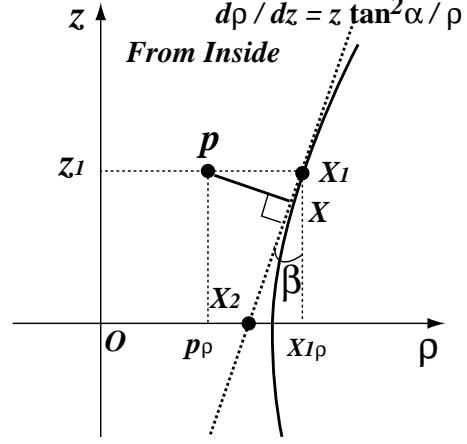
Figure 8: Distance from outside

Figure 9: distance from inside

5.b.2 DistanceToSurface(p)

`DistanceToSurface(p)` is to calculate the minimum distance to the surface in question. Since this function is primarily used to set the safety radius not to cross any volume boundaries, it is allowed to underestimate the minimum distance. In the calculation of this minimum distance we can thus approximate the surface curve by a properly chosen straight line. Taking advantage of z-symmetry, we only have to consider the region of $z \geq 0$.

Distance from Outside

In this case, the proper line can be chosen in the following way:

1. Find the surface point ($\mathbf{X}_1$) which has the same z as $\mathbf{p}$.

2. Draw a line from $\mathbf{p}$ which is perpendicular to the asymptotic line $\rho = z \tan \alpha$ and let their intersection be Q.

3. Find the surface point ($\mathbf{X}_2$) that has the same z as Q.

4. Connect $\mathbf{X}_1$ and $\mathbf{X}_2$

Then, by construction, the distance from $\mathbf{p}$ to the line $\mathbf{X}_1\mathbf{X}_2$ never exceeds the minimum distance to the hyperboloidal surface. The z component of $\mathbf{X}_2$ can be obtained geometrically from Fig. 8:

$$
\begin{aligned}
z_2 &= OQ \ \cos\alpha \\
OQ &= OR \ \cos\alpha \\
OR &= z_1 + p_\rho \ \tan\alpha.
\end{aligned}
\tag{5.11}
$$

Distance from Inside

In this case, a proper line will be the tangential line at the surface point $\mathbf{X}_1$ which has the same z component as $\mathbf{p}$ (see Fig. 9). The tangential vector is given by Eq. 2.4.

14

5.c J4FlatSurface

5.c.1 DistanceToSurface(p, v)

The distance to a plane along a velocity vector $\mathbf{v}$ from a point $\mathbf{p}$ is readily obtained from

$$t = -\mathbf{n} \cdot (\mathbf{p} - \mathbf{x}_0)/|\mathbf{n}|, \tag{5.12}$$

where $\mathbf{n}$ is the normal to the plane and $\mathbf{x}_0$ is the reference point on the plane.

5.c.2 DistanceToSurface(p)

Since the two planar boundary surfaces of `J4TwistedTubs` are perpendicular to the z-axis, which means its normal is in the direction of the z-axis. All `DistanceToSurface(p)` has to do is simply to return the difference between z-components of $\mathbf{p}$ and the plane in question.

5.d J4TwistedSurface

5.d.1 DistanceToSurface(p, v)

The distance to a twisted surface from a point $\mathbf{p} = (p_x, p_y, p_z)$ along a velocity vector $\mathbf{v} = (v_x, v_y, v_z)$ can be obtained in a similar manner to that for `J4HyperboloidSurface`. Substituting $\mathbf{X} = (X_x, X_y, X_z) = \mathbf{p} + t\mathbf{v}$ in Eq. 2.6, we have the following quadratic equation:

$$(\kappa v_x v_z)t^2 + ((v_x p_z + v_z p_x)\kappa - v_y)t + \kappa p_x p_z - p_y = 0 \tag{5.13}$$

for the distance t to the surface.

Notice that, although this quadratic equation may contain a solution for which $X_x = \rho_c < 0$, such a situation should not take place in practice, since the twist angle cannot exceed $\Delta\phi = \pi$ without breaking wires. This requires $X_x = \rho_c$ in Eq. 2.6 be positive. Notice also that, for a given stereo angle, the twist angle $\Delta\phi$ increases with the wire length. In solving Eq. 5.13, we thus need to set limits on z as determined by the endcap locations. These validity checks can be made for the obtained crossing point $\mathbf{X}$ after solving the above equation.

5.d.2 DistanceToSurface(p)

The exact equation for the minimal distance to the twisted surface from a point $\mathbf{p}$ is biquadratic. As in the case of `J4HyperboloidalSurface`, what we really need is, however, not the exact minimal distance. Its approximation will do, provided that it never exceeds the true minimal distance. We thus developed an approximation method that uses the distance to an appropriately chosen plane instead of solving the exact equation. Now the question is how we find such a plane to approximate the twisted surface. Fig. 10 illustrates the procedure to find it.

Since we should never overestimate the distance, the plane has to lie on the same side as the point $\mathbf{p}$ with respect to the twisted surface, at least in the vicinity of the
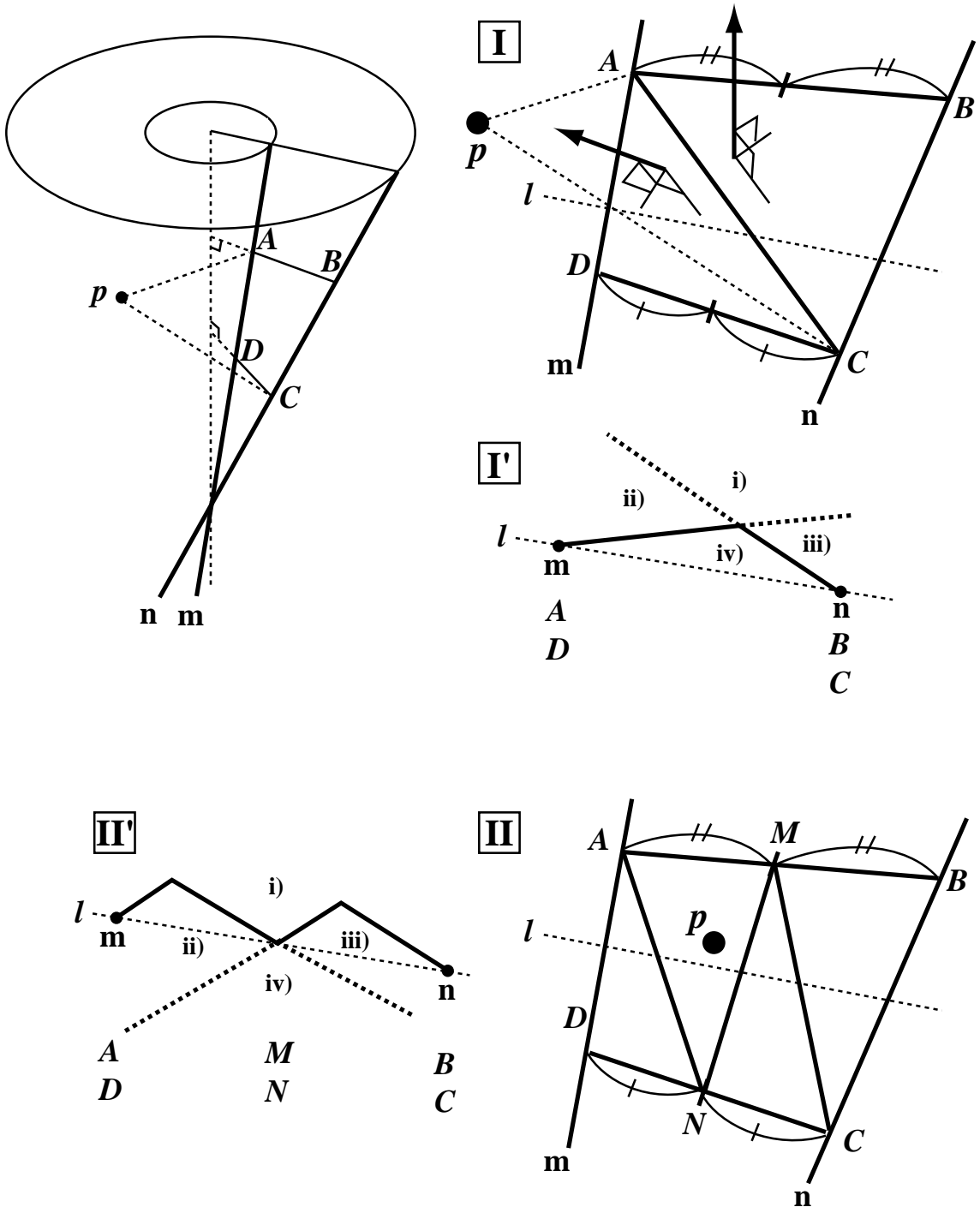
Figure 10: Procedure to choose an appropriate plane to approximate the minimum distance to the twisted surface

point of the closest approach to the plane. In order to fulfill this condition, essential is the following observation about the geometric features of the twisted surface. As stressed in section 2 the twisted surface contains two kinds of straight lines given by $\mathbf{e}_z$ and $\mathbf{e}_x$ in Eq. 2.9: one that can be regarded as one of the field wires making up the twisted surface and the other that can be taken as a ϕ boundary of a z-slice of a stereo cell. Lines AD and BC of Fig. 10 are the first type ($\mathbf{e}_z$-type), while AB and DC are the second type ($\mathbf{e}_x$-type). We can thus cut out, from the twisted surface, a segment $ABCD$ that is bounded by four straight lines. Recall that this segment has a saddle shape: it is concave along the diagonal AC in Fig. 10, while it is convex along the other diagonal BD[8].

Our goal is to choose a plane that is placed in front of both the current point p and the twisted surface and to calculate the distance to that plane. Once we find a diagonal which does not cross the twisted surface, we can immediately span a plane with the diagonal and an adjacent side (AB or DC) which is entirely lies on the surface[9]. Hence our first task is to select an appropriate diagonal along which the surface is concave. The procedure in `DistanceToSurface(p)` to select such a diagonal can be itemized as follows:

1. Let A and C be the points of the closest approach from the point p to the inner and outer boundary lines (m and n in Fig. 10-I). If the z components of A and C do not sandwich the z component of p, move A or C along the boundary lines until its z component matches that of p.

2. Choose D and B on lines m and n, respectively, in such a way that lines AB and DC are perpendicular to the z-axis and thus entirely contained in the twisted surface.

3. First check if diagonal AC satisfies our requirement by examining the sign of product of the following three test variables:

   (a) the signed twist angle $\Delta\phi$,

   (b) an orientation index, which is 1 if p is on the negative ϕ side of the twisted surface, 0 if p is on the surface within $\pm 0.5 \times$`kAngTolerance` ($= 0.5 \times 10^{-9}$) radians, and -1 otherwise, and

   (c) the z coordinate of A minus that of C.

   If the product is positive, accept the diagonal. If it is 0, which means either (b) is 0 or (c) is 0. If (b) is 0, p is on the surface (or is very close to the surface). In this case, immediately exit `DistanceToSurface(p)` by returning 0 as the minimal distance to the twisted surface. Else if (c) is 0, return the distance to line AC from p. Finally if the product turns out to be negative, which means that the diagonal is invalid crossing the twisted surface, exchange A and D, and C and B, respectively.

Diagonal AC can now be used to set up two planes: plane ADC and ABC. The normal to these planes are indicated in Fig. 10-I, and should point into the hemisphere that

_____

[8]Of course, which is convex or concave flips, depending on from which side you are looking.
[9]Remember the twisted surface can also be regarded as a locus of line AB in Fig. 4.

contains p. The subsequent distance calculation has two branches, depending on the relative location of p with respect to these two candidate planes.

Case A: p is in region i), or ii), or iii) of Fig. 10-I'

In these cases, all `DistanceToSurface(p)` needs to do is to return the smallest positive distance.

Case B: p is in region vi) of Fig. 10-I'

In this case, we must re-define the planes so that at least one of them lies behind point p. Let M and N be the middle points of AB and CD, respectively. Since line MN is fully contained in the twisted surface, we can now define new quadrangles $ADNM$ and $MNCB$. We can then define four planes ADN, AMN, MNC, and MBC. Planes ADN and MBC, however, coincide with ADC and ABC, respectively, and have hence been tested already. Therefore, we just have to test the remaining two planes. Again there are two cases to consider, depending on the relative location of point p with respect to the planes (see Fig. 10-I'),

Case B-1: p is in region i) of Fig. 10-II'
    In this case, `DistanceToSurface(p)` should just return the smaller of the distance to AMN and that to MNC.

Case B-2: p is in region ii) or iii) of Fig. 10-II'
    In these cases, the situation is essentially the same as in iv) of Fig. 10-I'. Compare the distances to AMN and MNC and split into two the one that gave a negative distance.

Notice that, by construction, p must not fall into region iv) of Fig. 10-II'. If it happens, therefore, we just abort the program.

The procedure for B-2 is actually implemented as a recursive call: `J4TwistedSurface` is equipped with its own `DistanceToPlane`, which calls itself recursively until the situation changes to B-1 and returns a positive distance. In this way `DistanceToSurface` can get the desired distance by a single call to `DistanceToPlane`, when case B takes place.

6   Test of J4TwistedTubs

In order to test the new solid class `J4TwistedTubs`, we prepared a test program (for stereo cells) as follows. We first constructed a world volume with a cubic shape of 6m × 6m × 6m and put, at its center, a cylindrical tube-type layer with inner and outer radii of 30cm and 130cm, respectively, and a length of 260cm. In this cylindrical layer, we placed a `J4TwistedTubs` object, which has a twist angle ($\Delta\phi$) and a ϕ-width of $\pi/3$, inner and outer radii at the endcaps of 50cm and 100cm, respectively, and a z-length of 200cm. Into this mother `J4TwistedTubs` volume, installed were two daughter `J4TwistedTubs` objects
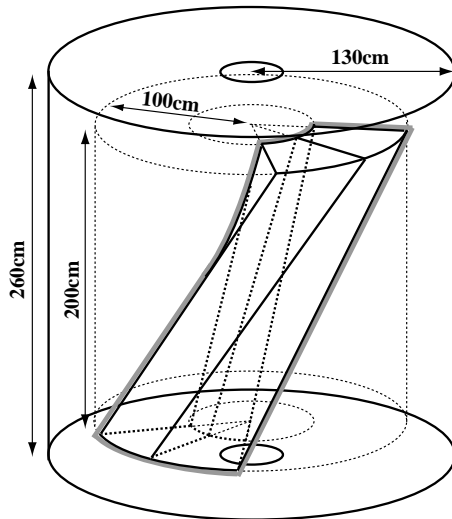
Figure 11: Schematic view of the test geometry

which have half the ϕ-width but otherwise the same geometry. All of these volumes are made of air. A schematic view of the test geometry is shown in Fig. 11.

In order to evaluate the computing speed of `J4TwistedTubs`, we also prepared another test program (for axial cells) with the mother and the two daughter `J4TwistedTubs` objects replaced by ordinary cylindrical tubs (`G4Tubs`).

6.a Test with Geantinos

Geant4 provides a hypothetical particle called "*geantino*" designed primarily for geometry tests. A geantino does not interact with any materials and flies along a straight line trajectory. It only makes a hit when it crosses a boundary of volumes which are *sensitive*[10]. We first carried out a geantino test of `J4TwistedTubs`, using the aforementioned geometry test program with only the daughter `J4TwistedTubs` objects made sensitive. Fig. 12 is a 3-dimensional view of hit points made by 10000 geantinos injected from 4π steradian around the surrounding cylinder. As seen from the figure all the hit points are correctly on the surface of the daughter `J4TwistedTubs` volumes.

6.b Test with Higgs Events

Unlike geantinos, a real charged particle makes a curved trajectory in a central tracker due to the magnetic field applied to it, and interacts with various detector materials, producing, for instance, low energy tracks such as δ-rays which might complicate the particle tracking through the detector volumes. In order to stress-test our `J4TwistedTubs`

---

[10]Geant4 generates a hit when (1) a physics process took place on a flying particle or (2) its track crossed a boundary of a volume registered as a sensitive detector.
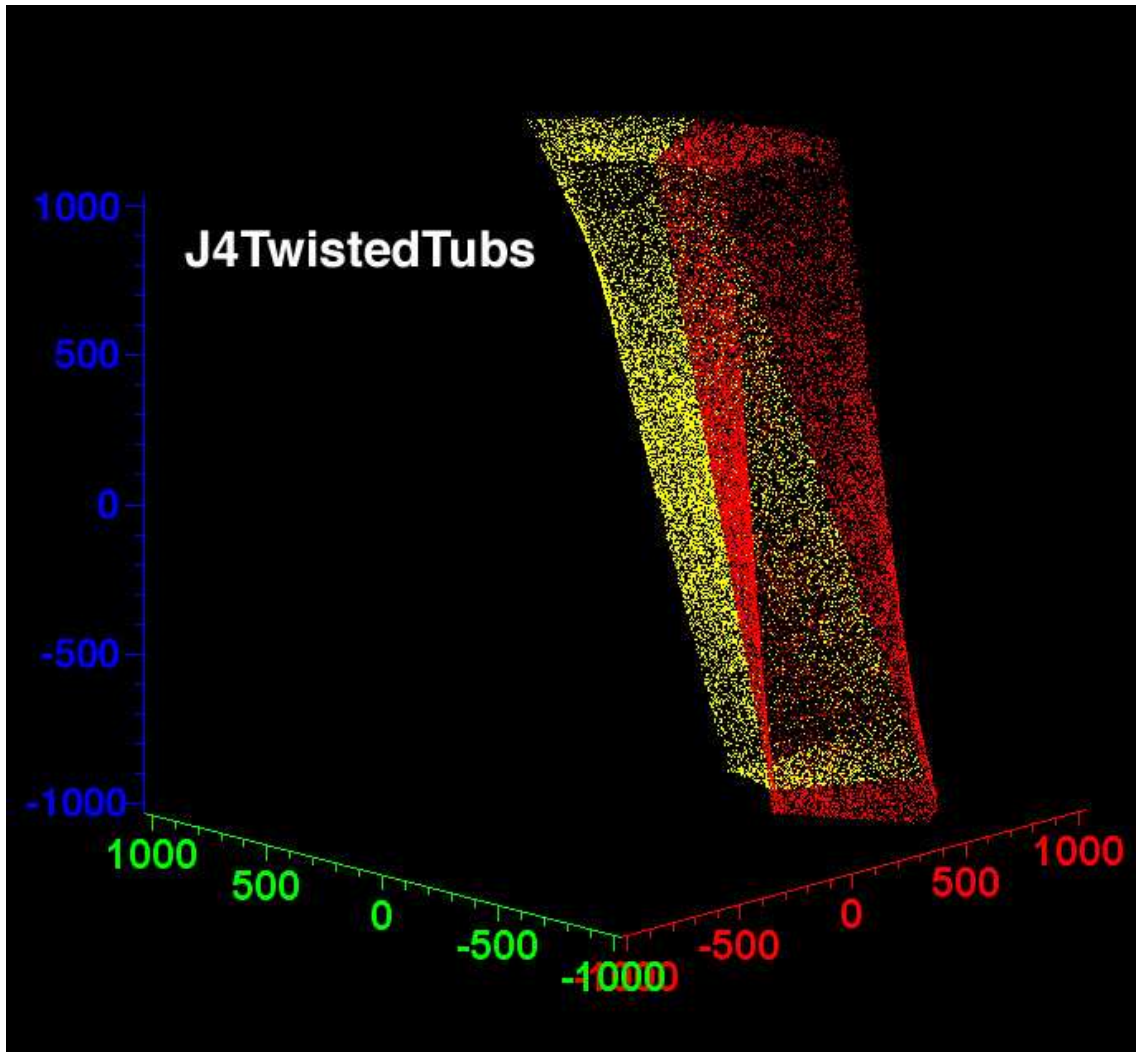
Figure 12: 3-dimensional view of hit points of geantinos on the daughter `J4TwistedTubs` volumes
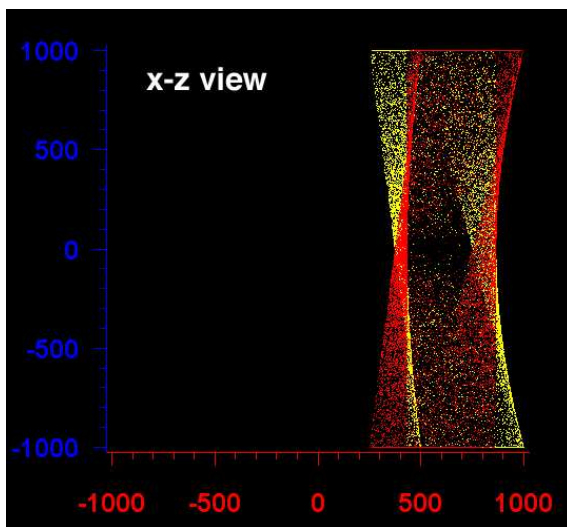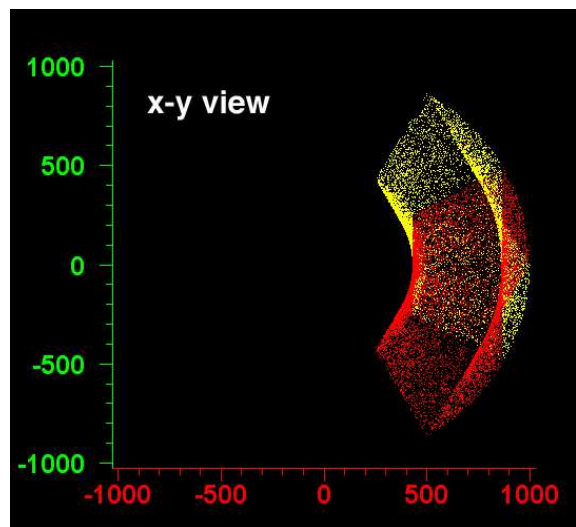

Figure 13: x-z view

Figure 14: x-y view

20

under more realistic environment, we also tried Higgs events ($e^+e^- \to ZH$ at $\sqrt{s} = 350$ GeV) generated by Pythia[6]. In this case, the number of tracks with a transverse momentum of 1 GeV or greater is around 50 per event. Because of the lack of a calorimeter in this test program, however, relatively low energy tracks curl up and pass through the sensitive volumes multiple times. Taking into account this curl-up effect and the geometrical acceptance of the J4TwistedTubs volumes being about $(\pi/3)/(2\pi) = 1/6$, the average number of tracks passing through the twisted volumes was estimated to be around 10 per event. Processing of 1000 Higgs events, which thus correspond to roughly 10000 tracks hitting the sensitive volumes, took 11647 seconds on a Power Macintosh 800MHz (Dual CPU) with 1GB memory for the J4TwistedTubs test program. The same test took 9242 seconds on the same platform for the axial cell test program. The 25% extra CPU time consumed by J4TwistedTubs is acceptable for our purpose, considering the expected CPU time necessary for calorimeter simulation.

7   Discussion

So far we treated the twisted surface analytically in J4TwistedSurface. We found, however, Eq. 5.13, which is the key equation for our analytic treatment, suffers from roundoff errors in the situation sketched below.

In general a roundoff error enters as a result of a subtraction of two numbers that share many common digits. In the case of J4TwistedSurface, this happens when v_x or v_z or both become small, and consequently the second term of the discriminant of the quadratic equation Eq. 5.13 becomes negligible. Geometrically $\mathrm{v}_x = 0$ corresponds to an extreme case in which the velocity vector of the particle is contained in a plane that is spanned by a straight line in the direction of a wire and another straight line which is parallel with the z-axis. On the other hand, $\mathrm{v}_z = 0$ is the case in which the velocity vector is in the plane that is perpendicular to the z-axis. In both of these cases, the problem becomes purely 2-dimensional and hence the equation becomes linear.

When the quadratic equation is used in such a case, DistanceToSurface(p, v) might return a crossing point that falls short of or goes beyond the surface by more than kCarTolerance. The next step might then end up with an impossible situation where DistanceToIn (DistanceToOut) would be called from inside (outside) of the volume, confusing the tracking program in Geant4.

When the calculated crossing point (X_{orig} in Fig. 15) is found to be more than kCarTolerance-off the surface, we apply the following Newtonian method in order to avoid such inconsistency. We first approximate the twisted surface, at the surface point that has the same local x and z coordinates (X_{surf}), by a tangential plane spanned by two straight lines contained in the twisted surface: one in the wire direction and the other in the radial direction. We then recalculate the crossing point (X_{new}) with this tangential plane. We iterate this procedure until the recalculated crossing point is found on the surface within the tolerance.
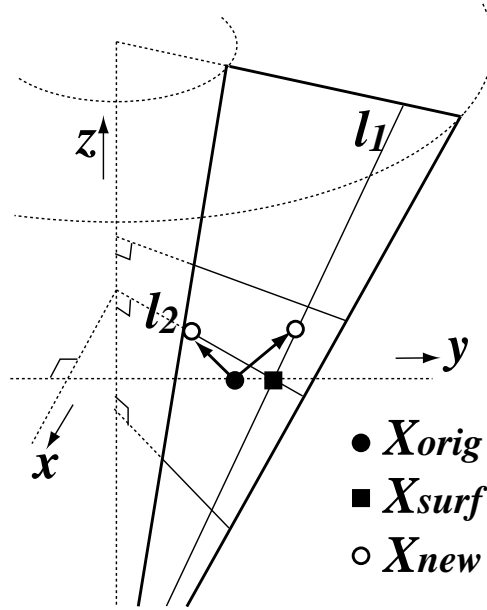
Figure 15: Procedure to fix the estimated intersection (X_{orig}) when it falls short of the surface due to a roundoff error. The surface point (X_{surf}) has the same x and z coordinates as X_{orig}. We approximate the twisted surface by the tangential plane at X_{surf}, which is spanned by two straight line sections l_1 and l_2, and calculate a new crossing point (X_{new}) with this plane. In the limit of $v_z = 0$ or $v_x = 0$, for which the numerical instability is expected, the new intersection will be on l_2 or l_1, respectively, and therefore the approximation will become exact.

8 Summary and Conclusion

We have developed a new Geant4 solid called `J4TwistedTubs` in order to handle stereo mini-jet cells of a cylindrical drift chamber like JLC-CDC. This new solid consists of three kinds of surfaces, each of which is represented respectively by `J4HyperboloidalSurface`, `J4FlatSurface`, and `J4TwistedSurface`. These three surface classes correspond to inner and outer hyperboloidal surfaces, two end planes, and two so-called twisted surfaces that make slant and twisted ϕ-boundaries, respectively.

There has been no Geant4 object for the twisted surface. In this paper, we have thus explained the algorithmic details of our new surface class (`J4TwistedSurface`) for it. It should also be emphasized that all of these three surface classes are derived from a single base class called `J4VSurface`, which greatly facilitated the distance and surface normal calculations.

We have implemented stereo cells with the new solid, and tested them using geantinos and Pythia events ($e^+e^- \rightarrow ZH$ at $\sqrt{s} = 350$ GeV). The stereo cells consumed 25% more CPU time than ordinary axial cells did. We found this acceptable, considering the expected CPU time necessary for calorimeter simulation.

Acknowledgments

References

[1] JLC group, KEK Report 92-16, December, 1992.

[2] ACFA Linear Collider Working Group, KEK Report 2001-11, August (2001), http://www-jlc.kek.jp/subg/offl/jim/index-e.html

[3] Proceedings of the APPI Winter Institute, KEK Proceedings 2002-08, July (2002)

[4] N.Khalatyan, K.Fujii, H.Okuno,T.Abe, K.Hoshina, Y.Kato, Y.Kurihara, H.Kuroiwa, T.Matsui, O.Nitoh, A.Sugiyama, K.Takahashi, T.Watanabe, T.Yoshida, Nucl. Inst. and Meth. **A428** (1999) 403.

[5] http://wwwinfo.cern.ch/asd/geant4/G4UsersDocuments/UsersGuides/ ForToolkitDeveloper/html/index.html

[6] http://www.thep.lu.se/t̃orbjorn/Pythia.html