0.1 Vertex Finding - ZVTOP

ZVTOP[1] is a proven vertex finding algorithm developed by D Jackson and used at the SLD experiment[2]. ZVTOP was chosen for this study due to its previous use for phenomenological studies of a linear collider (e.g.[3],[4],[5],[6]) and the local presence of the initial developer of the algorithm. ZVTOP consists of two complementary vertex finding methods, ZVRES and ZVKIN[7], which are detailed below. The use of two algorithms enables coverage of all jet topologies.

0.1.1 ZVRES

First the ZVRES method of ZVTOP is detailed. ZVRES uses mainly topological (as opposed to kinematic) information and is most suited to decays where there is more than one seen track from each vertex. The algorithm proceeds by locating points in space where vertices are likely to exist through use of a heuristic function $V(\mathbf{r})$, as defined below, that is a function of track density at a given point. Each maximum in $V(\mathbf{r})$ is used as a candidate vertex location. Ambiguities in track to vertex association are then resolved by comparing the magnitude of $V(\mathbf{r})$ between vertices and how compatible each track is with each candidate vertex.

Differences in the algorithm presented here compared to the original ZVTOP paper are due to subsequent developments in the SGV FORTRAN implementation that had been used for previous studies. These changes were retained in the current version so that comparisons could be made between studies performed with this version and the SGV FORTRAN implementation.

Before detailing the steps of the algorithm it is necessary to define some of its constituent parts. The effect of parameters (in italics) introduced is discussed after the detail of the algorithm.

Vertex Function

Function $V(\mathbf{r})$ is the heuristic function used as an indicator of the likelihood of a true vertex at r. It is based on a function $f_i(\mathbf{r})$ that is defined for each track and which gives a measure of the likelihood of the track truly passing though point r. It is therefore a tube with a Gaussian cross section with a width proportional to the error on the track that follows the line of the track. The Gaussian is unnormalised so that the function retains the value of 1 when co-incident with the path of the track. In the original ZVTOP paper it is defined as:

$$f_i(\mathbf{r}) = \exp\{-\frac{1}{2}[(\frac{x' - (x'_0 + \kappa y'^2)}{\sigma_T})^2 + (\frac{z - (z_0 + \tan(\lambda)y')}{\sigma_L})^2]\}$$

Note that this is a parabolic approximation to the track trajectory where x' and y' are such that the track momentum is parallel to the y' axis at the track's point of closest approach to the IP; x'_0 and z_0 are the co-ordinates at this point. Note that y'_0 does not appear as the y' axis has been made parallel to the track. κ is the curvature in the xy plane and λ is the track's dip angle of the track in the yz plane with respect to the y axis. The values σ_L and σ_T are the track errors in the z direction and xy plane respectively. For the version of ZVTOP developed for these studies, the parabolic approximation was removed and replaced with:

$$f_i(\mathbf{r}) = \exp\{-\frac{1}{2}(\mathbf{r} - \mathbf{p})\mathbb{V}^{-1}(\mathbf{r} - \mathbf{p})^T\},$$

where $\mathbf{p}$ is the point of closest approach of the track to $\mathbf{r}$ and $\mathbb{V}$ is the covariance matrix of the track at $\mathbf{p}$. In this case, $f_i(\mathbf{r})$ follows the helix exactly rather than following the parabolic approximation.

These independent track functions are combined to produce the vertex func-

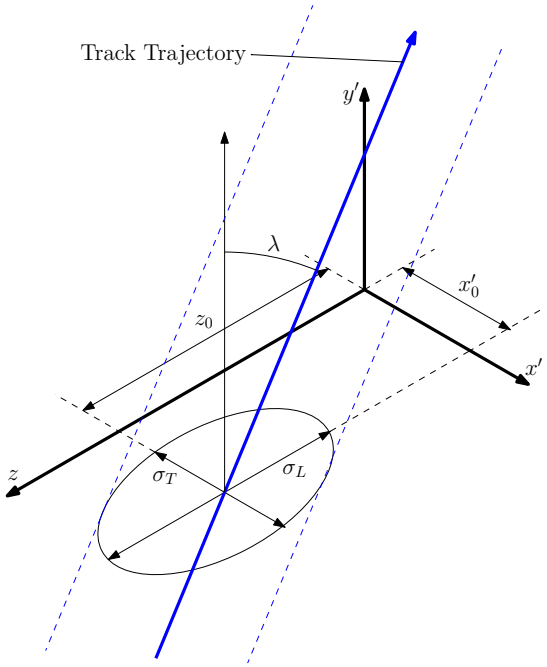

Figure 1: Parabolic approximation variables

tion that gives a measure of the likelihood of a track co-incidence at any point:

$$V(\mathbf{r}) = \sum_{i=1}^{N} f_i(\mathbf{r}) - \frac{\sum_{i=1}^{N} f_i^2(\mathbf{r})}{\sum_{i=1}^{N} f_i(\mathbf{r})} \tag{1}$$

This function tends to zero at points that are only near one track, and has maxima at the coincidences of two or more tracks. Therefore large values of $V(\mathbf{r})$ indicate points in detector space that are likely to contain true vertices. For example, in a region very close to three tracks, $V(\mathbf{r}) \to 3 - (3/3) = 2$. This represents the most basic form of $V(\mathbf{r})$; it is modified to suppress maxima near fake vertices by two mechanisms: the addition of an IP object and a weighting of the volume around the jet axis.

The IP object is added by defining $f_0(\mathbf{r})$ for the IP analogously to $f_i(\mathbf{r})$ for

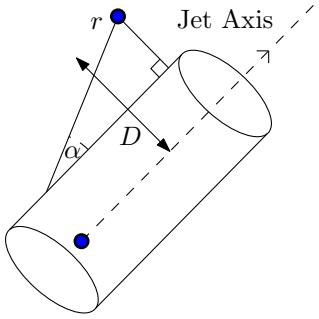

Figure 2: Jet Axis Weighting

the tracks:

$$f_0(\mathbf{r}) = \exp\{-\frac{1}{2}(\mathbf{r}-\mathbf{p})\mathbb{V}^{-1}(\mathbf{r}-\mathbf{p})^T\},$$

where $\mathbf{p}$ is the IP position and $\mathbb{V}$ the IP covariance. This $f_0(\mathbf{r})$ is added to $V(\mathbf{r})$ with a weight K_{IP}:

$$V(\mathbf{r}) = K_{IP}f_0(\mathbf{r}) + \sum_{i=1}^{N} f_i(\mathbf{r}) - \frac{K_{IP}f_0^2(\mathbf{r}) + \sum_{i=1}^{N} f_i^2(\mathbf{r})}{K_{IP}f_0(\mathbf{r})\sum_{i=1}^{N} f_i(\mathbf{r})}$$

This ensures a large peak dominates $V(\mathbf{r})$ at the IP location, such that fake vertices near the IP will be subsumed into the IP. RESOL A weighting is implemented to suppress maxima that are far removed from the jet axis, which are likely to be fake. This takes the form of a cylinder around the jet axis in which $V(\mathbf{r})$ is unchanged and outside of which $V(\mathbf{r})$ reduces in proportion to the angle α between the side of the cylinder and the line from the base of the cylinder to $\mathbf{r}$ (see figure 2):

$$V(\mathbf{r}) \rightarrow \begin{array}{ll} V(\mathbf{r}) & D \leq 50\mu m \\ V(\mathbf{r})\exp(-K_\alpha\alpha^2) & D > 50\mu m \end{array}, \tag{2}$$

The parameter K_α controls how biased $V(\mathbf{r})$ is towards the jet core, and hence is made proportional to the jet momentum.

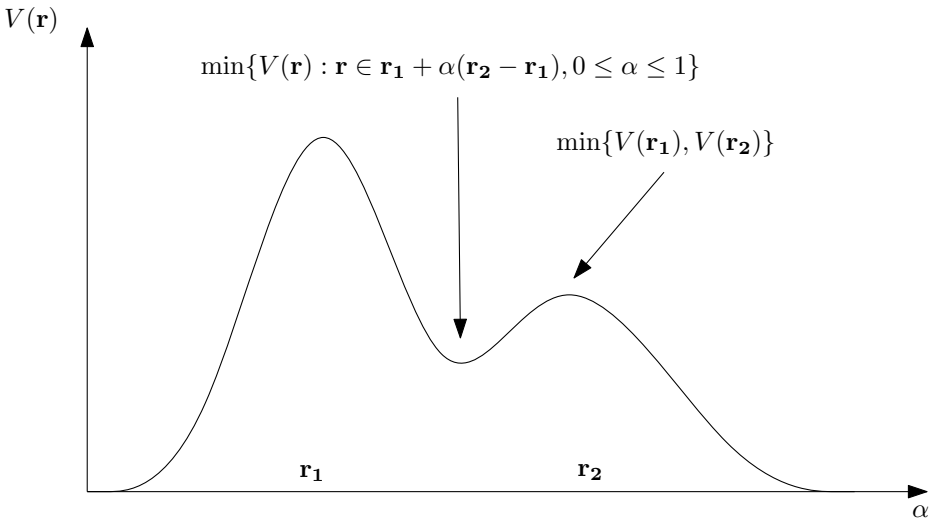

Figure 3: Resolution Criterion

The final $V(\mathbf{r})$ used is the one including IP and jet-axis weightings, but these are an optional part of the algorithm. Any given vertex has two values of $V(\mathbf{r})$: $V(\mathbf{r_{VERT}})$ as measured at the fitted vertex position; and $V(\mathbf{r_{MAX}})$, the local maximum in $V(\mathbf{r})$ found by gradient ascent from the fitted vertex position. These are rarely the same point. For example if two tracks are fit, but the vertex fit is near a third track, the third track will pull the maximum in $V(\mathbf{r})$ away from the two-track fit.

Vertex Resolution

Determining whether two maxima in $V(\mathbf{r})$ are resolved (i.e. if they represent two distinct vertices) is a key part of ZVRES. Two vertices are resolved if:

$$\frac{\min\{V(\mathbf{r}) : \mathbf{r} \in \mathbf{r_1} + \alpha(\mathbf{r_2} - \mathbf{r_1}), 0 \leq \alpha \leq 1\}}{\min\{V(\mathbf{r_1}), V(\mathbf{r_2})\}} < R_0 \tag{3}$$

where R_0 is a threshold parameter (See figure 3). In other words, vertices are resolved if the ratio of the minimum $V(\mathbf{r})$ between the vertices to the lower $V(\mathbf{r})$ at either vertex is lower than the threshold. In this way if the vertices have no significant valley in $V(\mathbf{r})$ between them they are not resolved.

When two vertices are to be merged, the one with the smallest $V(\mathbf{r_{MAX}})$ is removed and its tracks added to the other. If the removed vertex contained the interaction point, it is assigned to the other vertex.

Algorithm

The general procedure followed by ZVRES is shown in figure 4 and described below.

Generate two track candidates

The algorithm proceeds as follows: A vertex is created for each possible pairing of the tracks in the input jet. These vertices are fit and retained if they meet the following criteria:

Each track's contribution to the fitted vertex χ^2 must be less than parameter χ_0^2;

the value of $V(\mathbf{r})$ at the fitted vertex position ($V(\mathbf{r_{VERT}})$) is greater than a threshold parameter V_0.

If information about the IP is to be used, a further set of vertices is created, each consisting of the IP and one jet track. Again these are fitted and retained if the track or IP's contribution to the fitted vertex χ^2 is less than χ_0^2.

Note that for two-object fits such as these, the χ^2 of both objects in the fit is equal, so these criteria are equal to a cut on vertex χ^2 of $2\chi_0^2$. A track can be associated with many of the candidate vertices at this stage, for example at the IP there will be many of these two-track vertices that will eventually be merged if they are compatible.

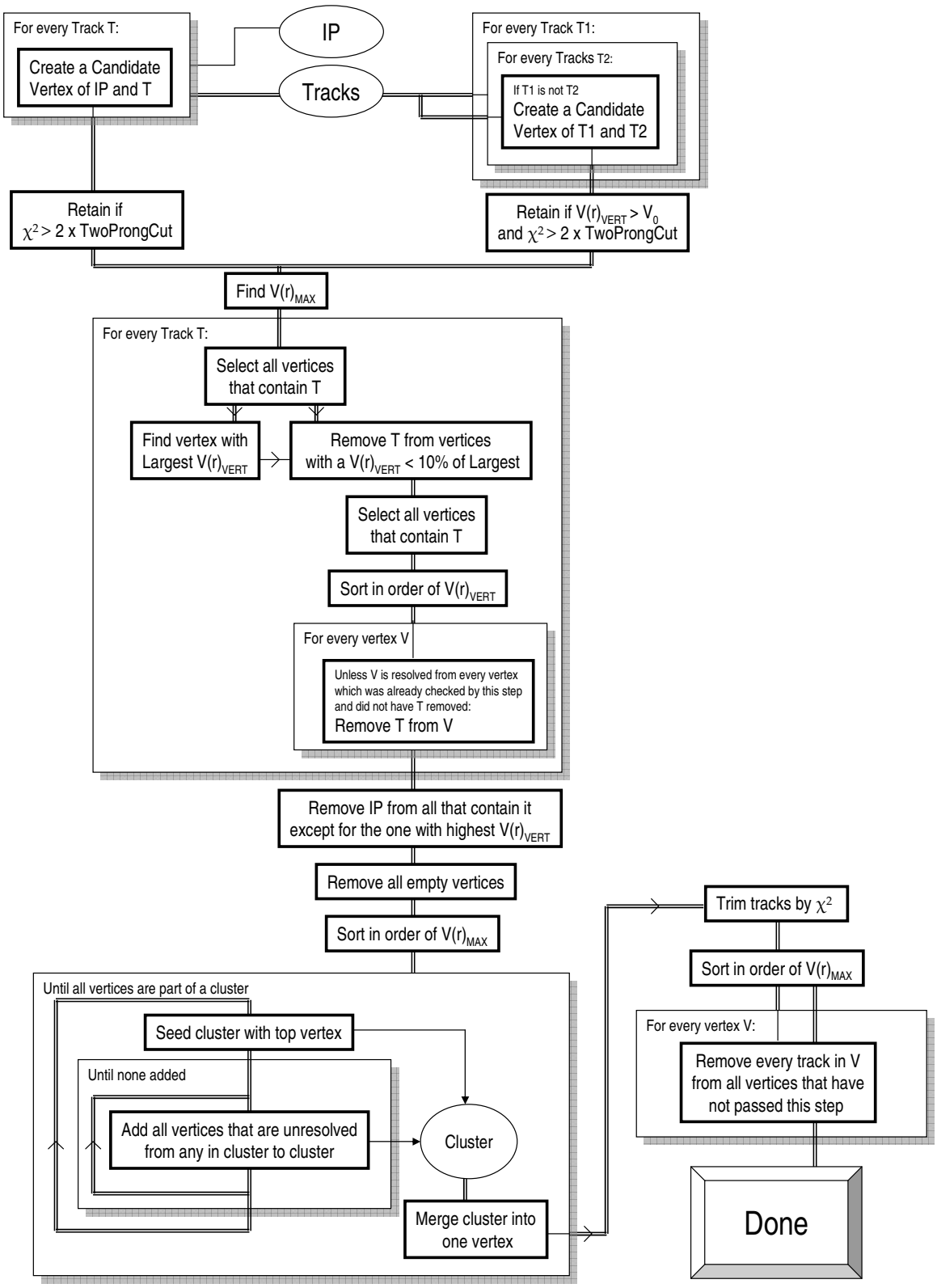

Figure 4: Flow diagram for ZVRES

Remove tracks

Tracks are now removed from vertices. During this process the vertices are not refit and those with only one track left are not discarded.

For each track T, the vertices that contain T are found. Track T is removed from all of these vertices that have a $V(\mathbf{r_{VERT}})$ less than 10% of the one with the highest $V(\mathbf{r_{VERT}})$. The remaining vertices that contain T are then sorted in order of $V(\mathbf{r_{VERT}})$ and considered in descending order. Track T is removed from the currently considered vertex if the vertex is unresolved from any previously considered vertex (for track T) that did not have T removed. The result of this is that for a pair of vertices that are unresolved only the one with highest $V(\mathbf{r_{VERT}})$ will retain the track.

The IP is removed from all vertices except that with the highest $V(\mathbf{r_{VERT}})$ that contains the IP. At this point, all candidate vertices that have zero parts left (tracks or IP) are discarded. Note that this leaves candidate vertices that have only one track, but which still retain their position from the initial two-track fit.

Cluster candidates

Each unresolved set of candidate vertices is now merged into a single vertex. This is performed by taking a seed candidate vertex and finding all vertices unresolved from it, then finding all vertices unresolved from those repeatedly until no more are found to be unresolved. All vertices found in this unresolved set are then merged. From the remaining vertices another seed is picked and this process repeated until all vertices have been considered. Note that the resolution of vertices in this step is performed using the position of $V(\mathbf{r_{MAX}})$ rather than $V(\mathbf{r_{VERT}})$.

Remove tracks by χ^2

Tracks are then cut from the vertices based on their χ^2 contribution. For each

vertex, if the track with the highest χ^2 contribution to the vertex has a χ^2 contribution above the cut threshold ($\chi^2_{0\mathrm{TRIM}}$), it is removed and the vertex refit. This is repeated till the track with the highest χ^2 contribution is below the cut threshold or the vertex no longer defines a point in space (i.e. has less than two tracks and no ip object). After this χ^2 trimming, all vertices that no longer define points in space are removed.

Resolve Ambiguities

The last stage is to remove any remaining ambiguities in track allocation where a track is contained in more than one vertex. Ambiguities are resolved by retaining the track in the vertex with highest $V(\mathbf{r_{MAX}})$ out of all vertices that contain that track. In descending order of $V(\mathbf{r_{MAX}})$, the candidate vertex's tracks are removed from all candidate vertices with a smaller value of $V(\mathbf{r_{MAX}})$.

Parameters

K_{IP} is the weight of the IP object in the vertex function. Due to the vertex function being used to resolve vertices, an increased value of K_{IP} will merge vertices near the IP with the IP. This can be useful for suppression of fake vertices, but note that this should not be necessary if the IP's covariance is correct as a larger covariance has the same effect as a higher K_{IP}. The default value is 1.

K_α controls the opening angle of the cone in which $V(\mathbf{r})$ is non-zero. A larger value makes a less divergent cone. As lower-energy jets are less collimated, this value is made proportional to the jet energy. The default value is 0.125 times the jet energy in GeV.

R_0 is the resolution criterion. A higher value will lead to increased merging of vertices that are spatially close. The default value is 0.6.

χ^2_0 and V_0 determine which of the $\frac{1}{2}N(N+1)$ possible two-object fits are immediately discarded at the beginning of the algorithm; a higher value of χ^2_0

or lower V_0 increases the number considered but will lead to fake vertices. The default values are 10 and 0.001 respectively.

$\chi^2_{0\mathrm{TRIM}}$ controls which tracks are rejected before the final ambiguity resolution, a higher value reduces the number of tracks rejected. The default value is 10.

0.1.2 ZVKIN

The ZVKIN method uses kinematic information about the input tracks in an attempt to identify those vertices that would normally be lost due to a lack of tracks seen in the detector. For example, from the decay vertex of $B^0 \rightarrow D^- e^+$, only the e^+ would be seen as the D would subsequently decay. Usually vertices such as this would be lost as, with only one track, the location of the vertex is undefined. As tracking is not possible for neutrals, many vertices fall into this category. Where the subsequent D vertex is found, the lone B daughter track can be easily identified as it will appear to come from a point close to the line joining the D vertex to the IP. In this case, the vertex can be recovered easily, after ZVRES for example. However in $\sim 23\%$ per cent of these cases (**??**) the D decay vertex produces one or zero seen tracks, leading to two vertices neither of which is found.

The ZVKIN algorithm is shown diagrammatically in (5). It overcomes even this seemingly pathological case by exploiting the fact that decays from an initial particle produced at the interaction point lie on a relatively straight line following the initial particle's momentum. At a centre-of-mass energy corresponding to the Z resonance, the D decay point in a $B \rightarrow D$ decay chain is is on average 4200 μm from the IP with the intermediate B decay point on average only displaced 46 μm from the line joining the IP and the D decay vertex. This corresponds to an angle of $\sim 10mr$ between the lines from the interaction point

to the B and D vertices. FIGURE To use this information the algorithm adds a fake track to the set of tracks to be vertexed that represents the unseen B track. This track is known as the "ghost track" as it represents the resurrected B track. The algorithm then proceeds to cluster two-track coincidences using fitted vertex probability to guide a "build-up" clustering algorithm.

Finding the ghost track

Unlike the helical event tracks, the ghost track is linear with constant width covariance and passes though the interaction point of the event. The principle is to choose the direction and width of the track such that its vertices with the rest of the tracks in the jet have a reasonable χ^2, as this would be the case for the B track had it been observed. This direction and width are found by a two-stage iterative chi-squared minimisation. The first step ensures a ghost track that points in approximately the correct direction. Firstly a linear track G is created such that it passes through the interaction point and has unit momentum in the direction of the input jet momentum. It is created with a constant circular error ellipse of width $25\mu m$. From this initial direction, G is swivelled in both the θ and ϕ directions whilst still passing through the interaction point. These angles are varied so as to minimise χ^2_{S1}:

$$\chi^2_{S1} = \sum_i \begin{array}{ll} \chi^2_i & L_i \geq 0 \\ (2\chi^2_{0i} - \chi^2_i) & L_i < 0 \end{array} ,$$

where χ^2_i is the χ^2 of the vertex formed by track i and track G, L_i is the distance from the interaction point to this vertex projected onto track G, signed so that if the vertex is in front of the interaction point (with respect to the jet) it is positive. This is simply the dot product of the momentum of track G and the vertex position. The value χ^2_{0i} is the χ^2 of the vertex formed by track i and

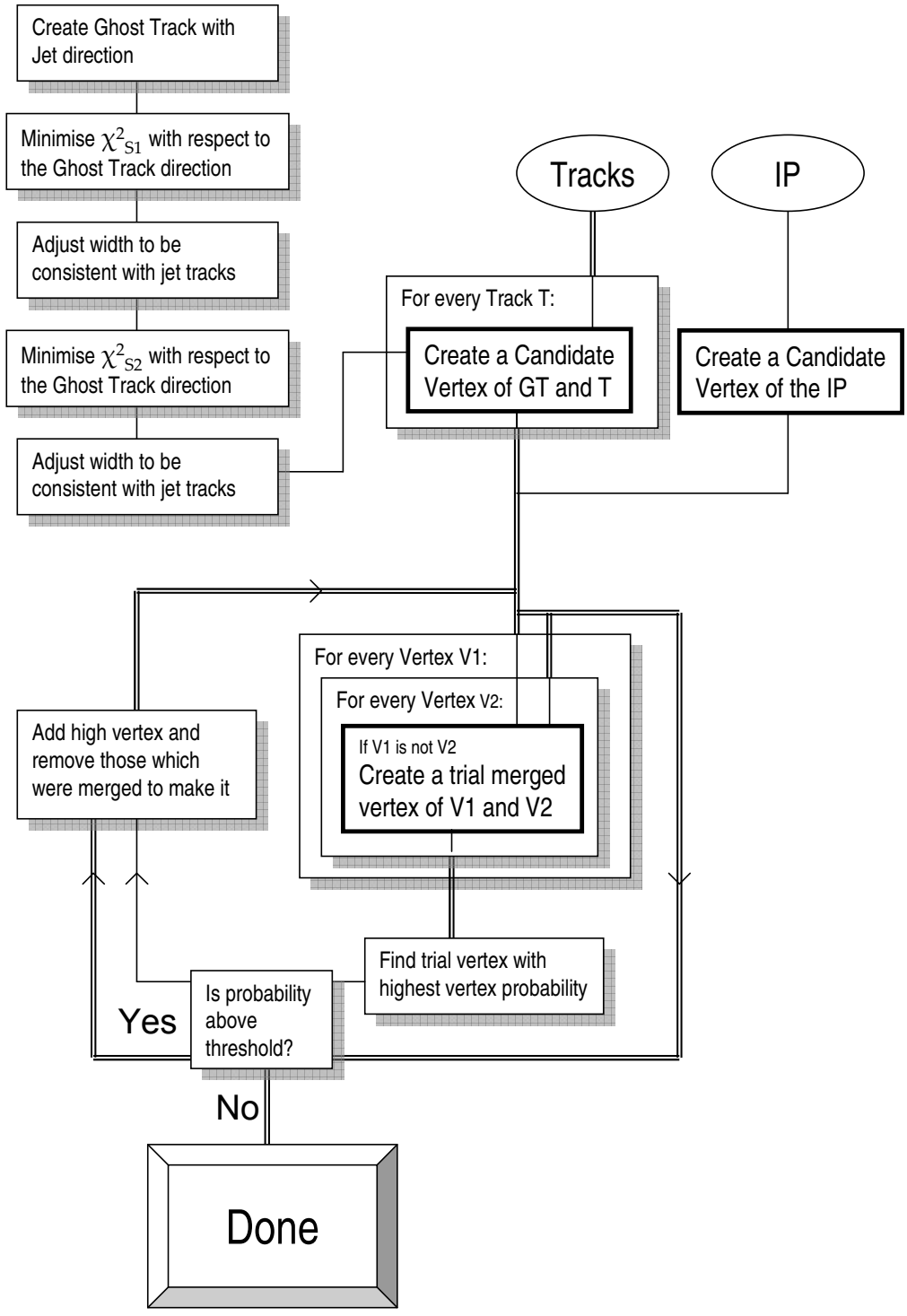

Figure 5: Flow diagram for ZVKIN

track G where the vertex position is constrained so that L_i is zero (equivalent to fitting with an ellipse of the ghost-track width at the interaction point). If the initial direction of G is at a large angle to the true B direction, some of the tracks may have vertices including G that are behind the interaction point ($L_i < 0$), hence this term of the χ_i^2 is constructed to push G away from these vertices to the true B direction. Note that this mechanism works only if G is initially in the same hemisphere as the true B decay direction. Also note that as the minimisation proceeds the values of χ^2 and L_i change continuously. χ_{S1}^2 is minimised until movements of 0.1 $mrad$ in both θ and ϕ give no improvement.

After this initial minimisation, G should be a rough approximation of the true B line of flight. At this point its width is adjusted so that every track with $L_i > 0$ when vertexed with G gives $\chi_i^2 \leq 1$. In practise only the track with largest χ_i^2 needs to be taken into consideration as if this has $\chi_i^2 \leq 1$ all other tracks will also. If this width is less than parameter GW_min, it is set to GW_min. The width of the ghost track at this point is an indication of how much the tracks conform to a straight-line decay, for example if a straight line could be drawn through all jet tracks, track G would have width GW_min.

A second round of minimisation is then performed with this new width and without the mechanism to move away from tracks with negative L_i. The value to minimise becomes:

$$\chi_{S2}^2 = \sum_i \begin{array}{ll} \chi_i^2 & L_i \geq 0 \\ \chi_{0i}^2 & L_i < 0 \end{array}$$

.

The minimisation is again performed to a level of 0.1 $mrad$. After this the width of G is again adjusted in an identical manner. At this point, G is the best approximation of the B line of flight and has a width consistent with having a vertex with all tracks ($L_i < 0$) at the level of $\chi_i^2 \leq 1$, ie it will form a good

vertex with all tracks in front of the interaction point.

Build vertices using the ghost track

Track G is now used as a normal track to find vertices in a build-up vertexing algorithm. Vertex probability is used as the criterion to cluster vertices. Due to the previous setting of the width of G, true vertices will have a flat probability distribution from 0.0 to 1.0. False vertices peak close to zero. Note that for this to happen the correct number of degrees of freedom must be used, which is $2N$ when fitting N tracks with the IP and $2N-2$ when fitting N tracks with G.

For N tracks, a set of $N+1$ vertices V are formed consisting of each track combined with G and one which is the interaction point. The vertex probability of each possible pairing of vertices from V is calculated (note that if the ghost track and the IP are in the combined vertex ,the ghost track is removed). The combined vertex that gives the highest probability is added to V and the vertices that combined to create it removed from V. Hence the number of vertices in V is reduced by 1. This process of combining the two vertices that are most probable continues until no combination gives a vertex probability greater than 1% or there is only one vertex left. At this point the tracks G and IP have been clustered into likely vertices. The ghost track is now removed from all vertices without refitting them. Note that this may leave some vertices with only 1 track.

0.2 Code Development

0.2.1 Motivation

Originally ZVTOP was implemented in the SLD FORTRAN framework, the ZVRES method of this implementation was adapted for the OPAL experiment at

LEP and then further adapted for ILC studies in the FORTRAN frameworks of BRAHMS, SIMDET and SGV. A direct non object-oriented translation to Java was attempted but suffered from performance problems. (Not sure of ref) The FORTRAN code base was poorly documented, becoming increasingly difficult to maintain, with five different track parameterisations. There were also many undocumented cuts and differences *details?* from the original ZVTOP paper which were introduced by various authors. BLAH IMPLICATIONS

0.2.2 Implementation Methodology

Current effort in the European ILC community is centred around the modern C++ framework of MARLIN and LCIO (REF). While technically possible and quicker to reuse the FORTRAN version of ZVTOP by means of a C++ wrapper or to directly translate it to C++, these were not seen as desirable as they would impede further development. Reimplementing the algorithms in an object-oriented fashion results in code that is easier to document, understand, maintain and develop. The aim was a set of processors for the MARLIN framework that integrated well with LCIO and the other processors being developed for tracking etc. in the ILC community

It was therefore decided to redevelop the code from scratch in C++ using the original ZVTOP paper as the starting point without reference to the FORTRAN code. Any differences in the output between the new and original versions when run on identical input tracks would then indicate either a bug in the new version or a difference between the original paper and the FORTRAN implementation.

ZVTOP was analysed and a unified modelling language (UML) class diagram (REF) for the code was developed (Figure 6 shows the simplified class diagram for the major parts of the code). The aim of this was to identify the classes to be implemented, their methods and how they collaborate to implement the

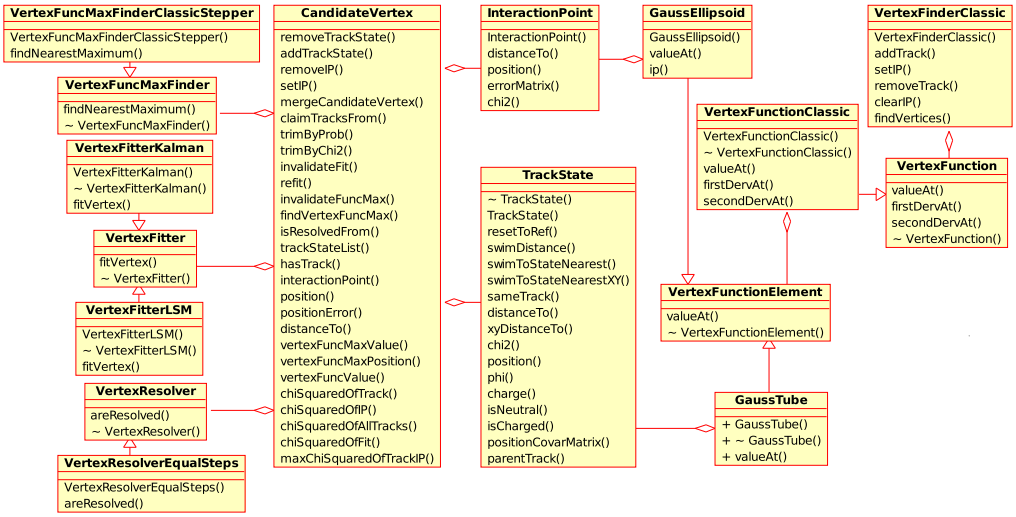

Figure 6: Simplified UML class diagram for ZVTOP

algorithm. Care was taken to enable possible future vertexing algorithms to be built on top of the code such that the algorithmic functionality of ZVTOP was decoupled from the functionality of track and vertex manipulations. It was vital to analyse the algorithms in this fashion before coding to avoid introducing unnecessary coupling and complication. The C++ language was chosen because the MARLIN[8] framework in which the algorithms run is C++ based. For the linear algebra and vector operations needed, the BOOST[9] library was selected as no standard had been agreed in the ILC community, and BOOST consists of statically compiled templates which could be portably distributed with the code. Where possible, the Standard Template Library [10] was used for containers and algorithms to avoid unnecessary coding and testing. Established best practise was followed for C++ such as the use of const and passing by reference for composite objects. Memory management was performed by a lightweight template framework (classes $MemoryManager\langle T\rangle$ and $MetaMemoryManager$) written by the author which is based on objects existing either for the duration of an event (usually physics objects) or a run (usually algorithmic objects). De-

tailed documentation is provided with the code as both in line comments and doxygen[11] blocks in header files.

The code was developed from the bottom up, starting with the underlying manipulations of tracks and vertices. Firstly, code for track propagation was implemented. This code underlies both algorithms and vertex fitting and hence must be fast and robust; to enable ZVKIN it needs to support both linear and helical tracks. Helical tracks were parametrised in the LCIO[12] format. An important concept for track manipulations is that of swimming (also known as propagating). The track is parametrised by the distance travelled along it in the $r\phi$ plane ,s, relative to its point of closest approach to some fixed point (usually the interaction point). The act of swimming is to vary s; for example, to swim the track forward one cycle of the helix would be equivalent to adding the circumference of the helix to s.

The following swimming methods were created as part of the *TrackState* class:
propagation of a point along a track, with accompanying propagation of the error on the track;
determination of the point on a track which is closest to a given point in detector space, both in the $r\phi$ plane and 3D;
determination of the point on a given track which is closest to another given track, again in the $r\phi$ plane and 3D.

For linear tracks, these operations are implemented by simple algebra, however for helices simple analytical solutions are not available. The following methods were implemented.

Swimming to the point of closest approach in the $r\phi$ plane is performed analytically by finding the point of closest approach on the circle that the track

follows and using the cycle of the helix in the z direction that is nearest the IP (in the direction of the momentum of the particle i.e. a 3/4 cycles forward is chosen over 1/4 backwards). This is a safe assumption to make as particles below a momenta of 100 MeV are cut by default for both algorithms.

Swimming to the point of closest approach in 3D is performed by initially swimming to the point of closest approach in 2D as these are close for helices of large radius and for small values of z. The following iterative procedure is then applied to find the 3D point. The distance from the track to the point is parametrised as a function $p(s)$, where s is the distance travelled along the track. The desired minima in $p(s)$ will correspond to the roots of $dp(s)/ds$. A Taylor expansion of $dp(s)/ds$ is taken at the current point and its roots found; s is set to the root which has the smallest $p(s)$. This process is repeated until the change in s is smaller than the desired swimming precision (By default this is $0.1\mu m$).

Swimming a track to its point of closest approach to another track is also performed iteratively using the point-swimming methods described above. The first track is continually swum to a point on the second track, which is moved along the second track in the direction that brings the points closer together until a minimum is found.

TrackState $TrackState$also contains methods for calculating the distance of the track to a point in both the $r\phi$ plane and $3D$, and to calculate the χ^2 of the track to a point by Eq. (**??**).

Where possible, operations on *TrackStates* are evaluated lazily and transparently cached by the *TrackState* itself; they are computed only when the result is actually needed and only computed once.

An analogous class *InteractionPoint* exists for the interaction point;p as this is stationary it provides only its location, covariance matrix and χ^2 to a point.

The methods for calculating $f_i(r)$ as used in the ZVRES algorithm were implemented by the *GaussTube* class which uses the *TrackState* class to implement equation XX in ZVRES above by swimming to the point of closest approach to r and using two-dimensional residual and error matrices identical to those described in the calculation of χ^2 above. The same function is implemented for the interaction point by the *GaussEllipsoid* class. Both of these are subclasses of *VertexFunctionElement*, an interface (abstract base class) which gives $f_i(\mathbf{r})$ without the caller needing to know if a tube or ellipse is being queried. These elements are then used to calculate $V(\mathbf{r})$ by the *VertexFunction* class which combines their output by Eq. (1) above and applies the interaction point ellipsoid and jet weighting.

The key class of object used by both algorithms in ZVTOP is the *CandidateVertex*. This provides methods for manipulating a set of *TrackState* objects, and optionally an *InteractionPoint*, which define a point in space. *TrackState* and *InteractionPoint* objects can be added and removed from the *CandidateVertex*, which automatically refits itself using a *VertexFitter* class. Two *CandidateVertices* can be merged into one, the *TrackStates* and *InteractionPoint* from one are added to the other (ensuring no duplication) and the first removed. A *CandidateVertex* can claim tracks and the *InteractionPoint* it contains as being exclusive to itself, removing them from other *CandidateVertices* as in the final stage of the ZVRES algorithm. There is also a method for the *CandidateVertex* to remove tracks from itself in order of their χ^2 until some threshold is reached.

For fitting, finding the maximum in $V(\mathbf{r})$ and resolving vertices the *CandidateVertex* uses separate objects which fulfil the interface classes *VertexFitter*, *VertexFuncMaxFinder* and *VertexResolver*; the objects to be used can be specified per-vertex if needed.

A simple least squares fitter was implemented - *VertexFitterLSM* which uses

a simple iterative descent method to minimise the χ^2 of the vertex. This class uses a general purpose function minimiser $FunctionMinimiser\langle T\rangle$. Two object fits are performed analytically, and other fits are seeded by the spatial average of the $1/2(N(N-1))$ two-object fits.

VertexFuncMaxFinderClassicStepper and *VertexResolverEqualSteps* were implemented as in the original FORTRAN code. The position of $V(\mathbf{r_{MAX}})$ is found by starting at the fitted vertex point and minimising in the along the x axis in $2\mu m$ steps , then similarly for the y and z axis. The step size is reduced and the axes minimised in turn until the step size is less than one micron. For resolving vertices if the distance between r_1 and r_2 is less than $10\mu m$ the vertices are considered unresolved. Note that this sets the minimum decay length that can be found. For greater distances, $V(\mathbf{r})$ is sampled at ten evenly distributed points on the line from r_1 to r_2, with the minimum being used as the numerator in Eq. (3)

The algorithms of ZVTOP were implemented in the classes *VertexFinderClassic* and *VertexFinderGhost*, which implement ZVRES and ZVKIN respectively. Each is assigned a set of *TrackStates* and an *InteractionPoint*. The method *findVertices()* then returns a set of *CandidateVertices* that are the output of vertexing algorithm.

The ZVKIN algorithm is further sub-divided so that the finding of the g-host track is performed in an independent class *GhostFinderStage1*. The subsequent clustering is performed by *VertexFinderGhost*.

0.2.3 Testing

Testing was performed concurrently with development. Initially the track and vertex-function classes were tested in a standalone executable with manually specified tracks; by fixing track errors and varying the distance between tracks,

the fitter's χ^2 and covariance matrix output could be tested. Pathological track combinations (e.g. coincident, very low pt) could be tested to check the robustness of swimming and fitting operations.

Testing of the ZVRES algorithm was more involved; in order to identify regressions from the standard FORTRAN implementation it was necessary to be able to pass the same input through both FORTRAN and the new implementation. Using the version of ZVRES in the SGV framework was the easiest way to achieve this as SGV is a simple fast Monte Carlo framework that has been used for previous ILC studies. It was relatively simple to link compiled C++ code to the SGV executable so that both could be run in the same executable. This process identified several bugs in the swimming and vertexing code. After fixing these, it was found that the new implementation on average outperformed the FORTRAN. ZVKIN was not tested as a working version was not readily available. The code was then integrated into the Marlin/LCIO framework using the working classes detailed in chapter X. In order to fully test this integration, an LCIO interface was written for SGV so that identical track input could still be compared. At this stage the code was profiled using the *Valgrind*[13] framework which identified that $\sim 90\%$ of the execution time was spent swimming while fitting, and that the time taken is exponentially dependent on the track multiplicity (Figure 7). *Valgrind* also identifies memory leaks, of which none were found.

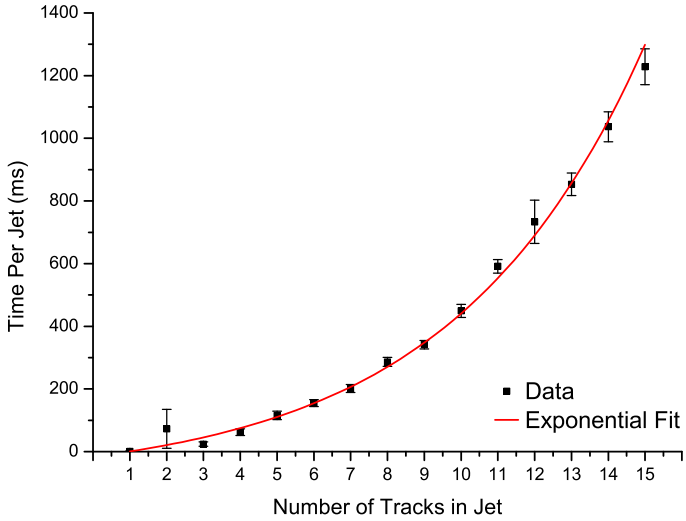

Figure 7: Run time of ZVRES in ms as a function of the number of tracks in the input jet

Bibliography

[1] D.J. Jackson, Nucl. Instrum. Meth. A388 (1997) 247.

[2] SLD-Vertex Detector, F.E. Taylor, Prepared for 28th International Conference on High-energy Physics (ICHEP 96), Warsaw, Poland, 25-31 Jul 1996.

[3] W. Walkowiak, (2001), hep-ex/0110039.

[4] G.B. Yu et al., (2003), hep-ex/0309041.

[5] Higgs Working Group of the Extended ECFA/DESY Study, K. Desch, (2003), hep-ph/0311092.

[6] S.M. Xella Hansen et al., LC-PHSM-2003-061.

[7] J. Thom, SLAC-R-585.

[8] F. Gaede, Nucl. Instrum. Meth. A559 (2006) 177.

[9] J. Walter and M. Koch, The Boost Linear Algebra library, http://www.boost.org/libs/numeric.

[10] A. Stepanov and M. Lee, The standard template library, HP Laboratories Technical Report 95-11(R.1), November 14, 1995.

[11] D. van Heesch, Doxygen documentation generator, http://www.doxygen.org.

[12] F. Gaede et al., (2003), physics/0306114.

[13] N. Nethercote and J. Seward, Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, June 2007.