5-1993

Accurate Verification of Five-Axis Numerically Controlled Machining

Jerome L. Quinn
Dartmouth College

ACCURATE VERIFICATION OF FIVE-AXIS NUMERICALLY CONTROLLED MACHINING

Jerome L. Quinn

Technical Report PCS-TR93-191

5/93

Accurate Verification of Five-Axis Numerically Controlled Machining

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

by

Jerome L. Quinn

Dartmouth College

Hanover, New Hampshire

May, 1993

Examining Committee:

(chairman) Robert L. Drysdale

Donald Kreider

Dennis Healy

Robert B. Jerard

Dean of Graduate Studies

ABSTRACT OF THE DISSERTATION

Current automated machining systems are composed of a number of components to aid in bringing a surface from design to physical completion. Numerically controlled (NC) milling machines are used to cut parts out of stock. Programming these machines to cut a desired surface is still largely a matter of experienced human participation. Therefore, the need exists to verify that tool programs produce the desired part.

We present recent developments in the verification of NC tool programs. Many of these methods rely on approximating the stock material as vectors whose lengths reflect the amount of uncut material at any point. This allows simulation of 3-axis machining to be carried out efficiently, because the intersection process is simple to compute. Some machines are capable of 5-axis tool movements which are more versatile, but verification of these programs is difficult due to the complexity of the tool motion.

We show several techniques by which it is possible to determine the intersection of 5-axis tool movements and guarantee the accuracy of the results. These techniques can be integrated into current NC machining verification systems to allow checking of 5-axis programs. We then evaluate the relative performance of implementations on test data and real world data.

ACKNOWLEDGEMENTS

I would like to thank the people who have contributed and helped me complete this work, and my degree. First off, my advisor, Scot Drysdale, has been central to the process. The committee members, Don Kreider, Dennis Healy, and Bob Jerard have also been very helpful. In addition, I would like to extend special thanks to Ernst Snapper, who has given me much of his time and expertise in helping me think about and solve the problem presented. Finally, I would like to thank my friends at Dartmouth and especially my wonderful wife, Lisa Baker, for the support and encouragement I have received along the way.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

With the advent of computers, the promise of fast, accurate and automatic machining of parts was born. Pioneering research into numerically controlled (NC) machining in the late 1940s and early 1950s, MITs Servomechanism Labs developed a language for specifying geometry and cutting movements, called APT. This gave NC programmers a consistent interface to NC milling machines.

APT's geometrical formatting has been supplanted as CAD systems have developed to allow more refined and interactive surface definition and editing. Generating cutting tool movements has also been simplified, allowing significant increases in NC programming productivity. However, the basic process of machining is still similar. A designer creates a part description and then an experienced NC programmer generates cutting tool paths. They are then checked for accuracy, often by cutting models out of wood or plastic, a time-consuming and error-prone method [20].

Despite this, automated machining has resulted in improved efficiency and economic benefit. Ultimately, however, the complete automated NC machining process, from the input of a part specification to the fabrication and checking of correctness of the final part, would be a completely automatic sequence of operations, requiring no intervention. This would free up human designers to do the creative work without worrying about the details necessary to implement it.

1.1 The goal of an automated machining system

A complete automated NC machining system would be one that takes a part description and does everything required to manufacture the desired part. The chain of steps needed to accomplish this is not insignificant and each step is an involved procedure, calling for complex, interrelated decisions. The system must be able to make decisions about such things as methods for clamping a block of material while machining is going on as well as determining the different positions the block should be held and in what order. It has to select the cutting tools to use and how to use them most effectively.

Let us take a closer look at what is involved in the machining process. [20] offers an effective breakdown of the sequence of events:

Part Design Design of the part is usually accomplished within a CAD system today. These systems are widely available and have interactive capabilities that make them relatively simple to use.

Surface evaluation The part description must be analyzed to locate features that will significantly affect the machining process. For example, this stage should detect slots, bosses and large changes in surface orientation so that they can be compensated for during the generation of the NC program.

Machining Planning To cut the part from a piece of stock requires answers to several questions. For example, the system must decide how to clamp the part and how many different positions are necessary to allow machining of the complete product. The types and sizes of cutting tools must be selected, as well as the overall sequence of events. This requires knowledge and reasoning about a large number of details concerning the particular milling machine.

Tool Movement Generation Once the strategy has been plotted, the actual cuts to be made must be calculated. This means that an NC program must be created that will carve the desired surface. In actuality, the goal will be to machine a part that is within some tolerance of the desired result when everything is finished.

Simulation and Verification The NC program should be checked to guarantee that it produces the desired surface to within acceptable tolerance limits. Geometric correctness of the cutting program must be verified. Allowing the body of the milling machine to hit the stock or itself during the cutting could result in very expensive damage to both machine and stock. If the final result is out of tolerance or there is forbidden contact between surfaces, the generator can then replan the NC program. The loop of generation and simulation may repeat until everything is satisfactory.

Dynamic Simulation The previous step verifies the geometric accuracy of the program. This one analyzes the forces involved. The more material removed in a given time, the more force between the material and the tool. These forces cause deflections in both surfaces, and the result is further deformed. However, faster feedrates (speed of material removal) result in less machining time, reducing the costs. These factors must be balanced and feedrates set accordingly.

Determine Cost Once the feedrates have been determined, calculation of cutting time is straightforward. Once the costs of machining have been calculated, a go ahead or stop command can be issued.

Execute Machining Program Ideally, the machining equipment could sense the actual forces and adjust the feedrates to keep within tolerance.

Evaluate Final Result The part that has just been cut should be checked to make sure that it meets the requirements set out at the beginning of the manufacturing process.

As we can see, building a system that can do everything is a fairly formidable assignment. Each step poses challenges of its own. We will examine geometric verification in more detail.

1.2 Verification

The goal of verification is straightforward. Given a part description, an NC program of tool movements, and a workpiece that is to be shaped, determine if the NC program shapes the workpiece into a copy of the part description, and where it fails to do so.

This of course is a simplistic statement of the problem. In manufacturing, the question is really how close is the final product to the desired part, as opposed to are they the same. Usually, the manufacturing process introduces small errors into the desired surface. Therefore, verification must actually determine if the machined product is within some small tolerance of the part, inside or out, for all points on the surface. In addition, verification must be prepared to identify the locations where gouging has occurred and where too much excess material has been left behind, if we wish to correct the tool program.

Another objective of verification is to check that the tool doesn't gouge the mounting hardware or other parts of the mill. In addition, detection of any tool movement that causes the non-cutting surfaces to hit workpiece material is desirable. Otherwise, an expensive repair may be in order. We may also like to know the volume of material that a tool movement removes to allow for later dynamic simulations. These last desired features call for simulation of the cutting process, not just determining the final product of the milling.

To accomplish these goals, we must represent the components involved in the actual cutting process. First, the part description is needed. We must have a representation for the workpiece to be machined. The swept volume of each tool movement must be available in some usable representation (some representations are much more useful than others for certain purposes). And if we wish to examine interference of surfaces that should not come into contact such as the machine body, we must have representations of those surfaces as well.

Given this information, we must provide a method for removing the swept volume of each tool movement from the workpiece model. The tool movements have to be processed in order if we wish to do simulation of the workpiece and environment at any time, as opposed to just the finished product.

Let's take a closer look at the means developed for performing verification.

1.2.1 Methods of Verification

There are several ways one can go about performing verification. A commonly available technique is to display the tool paths on top of the part surface. This has the advantage of being quite simple to implement, as well as being easy on the computer. Then verification could be performed by visually comparing tool paths and the part surface. However, only gross errors are likely to be found in this manner. It is also time intensive. Since manufacturing often requires tight tolerances, something better is needed. It also makes sense to have the computer doing the work, both for the sake of speed and reliability.

Increasing the computer's involvement in the verification process, we can check for interference between the part surface and the swept envelope of each tool movement. This will locate tool movements that gouge the part surface, but has several disadvantages. Since no model of the workpiece is kept, it is inconvenient to compare the machined result to the desired part surface. Each tool movement can keep track of its impact on the surface. but the overall effect is not known. In addition. since this is not a simulation, material removal cannot be determined.

The two methods mentioned above are not really simulations of the machining process [20]. To perform all of the tasks of verification. a geometric model of the machining process must be maintained. Solid Modeling and Discrete Modeling are the two major methods of simulating machining.

Solid Modeling

Solid modeling was developed as a way of representing complex surfaces built up as boolean combinations of simple objects. The machining operation is just this — an object (the workpiece) from which other objects (the tool movement envelopes) are subtracted. An accurate model of the current state of the workpiece is maintained at all times, and material removed by each tool movement can be determined. Solid modeling systems for NC simulation and verification were investigated by Voelcker and Hunt in [34] and [17] and by Frishdal in [12].

Using solid modeling to do simulation and verification has a down side. however. Simulation requires performing intersections between surfaces, a very computationally intensive operation. In addition, if the surfaces are too complicated, no known algorithm for performing the intersection may exist. Later work has addressed this problem by trying to spatially subdivide the problem to reduce the intersection operations necessary. On the verification side. comparison between the finished workpiece and the desired part is nontrivial. Even if they are exactly the same, floating point errors will most likely prevent that from being realized. Therefore, a method of measuring the nearness of the two surfaces is needed. also requiring potentially expensive surface-based operations.

Discrete Modeling

While solid modeling has the advantage of maintaining an exact representation of the machining process, it is computationally difficult. By trading total accuracy for simple computation, we may obtain an alternate approach. The general concept of discrete modeling is to represent an object by an approximation constructed of points, triangles or other simple objects that can be dealt with easily. The part surface and/or the workpiece are often represented by a set of points, each having an associated vector. The advantage is that intersections and other computations involving points and vectors are easy and fast.

Several researchers have proposed methods of NC simulation and verification using discrete approximations. These proposals can be divided into two groups. Z-buffer algorithms and surface normal algorithms.

Z-buffer Algorithms

An early approach to simulation by Anderson [1] was designed to detect if the tool holder hit the workpiece in three-axis machining. He set up a rectangular array, each grid cell containing a height value, to represent the workpiece. For each tool movement, the grid squares that fall under its path are checked against the height of the tool and reduced if the tool cuts the remaining stock in that square. This is closely related to the Z-buffer algorithm used for graphics displays.

The amount of material removed can be approximated with this method, as can the final shape of the workpiece. This means that gouges and excess material can be found. A regular grid greatly simplifies the calculations, but means that grid size is directly related to the accuracy of the simulation. Another disadvantage is that material can only be removed from above. This precludes five-axis cutting and undercutting with the tool.

Wang [35], Van Hook [33], and Atherton [2] each updated the Z-buffer to perform simulation and verification. The methods allow a view direction to be chosen. That is the direction in which the workpiece will store height values. Wang stores multiple Z values at each pixel, which allows him to simulate five-axis machining. He then approximates the tool movement envelope by a polyhedral approximation. Scan line operations then allow quick determination of the cutting depths of the tool movement at each pixel. The values in the resulting swept envelope Z-buffer are then used to update the workpiece representation. Since the surface values were originally used to create starting entries in the workpiece Z-buffer, comparing the finished workpiece and the desired part is easy.

Van Hook's approach is similar, except that a pixel image of the cutting tool is precomputed. Then it can be subtracted from the workpiece Z-buffer, finally giving the finished surface. Because the tool envelope image is precomputed, the tool axis is not allowed to change during machining, limiting the method to three-axis cutting operations. Atherton extended this approach to allow five-axis cutting.

All these approaches have advantages in common. Because of the regular nature of the Z-buffer grids, intersections with the tool envelopes are simple and fast. The values of the surface and workpiece are maintained at each pixel, allowing easy comparison of the final product to the desired part. In addition, gouges and excess material left behind can be easily found. Finally, the removal of material is quite easy to compute.

However, they suffer from common deficits as well. The simulations are view dependent, so they must be redone if an error doesn't show up in a given direction.

The magnitude of a given error at a given pixel is variable, depending on the orientation of the surface at that point. If the surface is nearly parallel to the Z direction there, the error value will not correctly reflect the size of the gouge or bump. The accuracy of the simulation is dependent on the size of the grid. To space samples on the part surface evenly, pixels must be closer together when the surface normal is nearly perpendicular to the Z direction. Lastly, preparing the starting Z-buffer requires finding the intersection of lines in the Z direction at each pixel with the part surface. This is referred to as the inverse point problem.

In the case of the parametric curved surfaces often used in part descriptions. solving the inverse point problem often requires iterative methods to get an answer. taking a fair bit of computational effort. For more complex surfaces, errors may be introduced at this step.

Accuracy is a troubling problem for methods using a regular grid. Grid sizes must be based on the tightest spacing necessary over the surface. This results in many unnecessary samples in areas that don't need it them. Drysdale and Jerard [8] propose a method similar to those above, but with variable spacing. They choose points on the part surface combined with vectors at each point to represent the workpiece. In addition, they show how to determine the space allowable between points on the surface depending on the desired accuracy of the simulation, the size and shape of the cutting tool, and the local surface curvature. In addition, they generate the sample points parametrically. This allows them to avoid the inverse point problem. When the axis of every tool movement is the same, highly efficient calculations can be had by picking the Z direction to coincide with the tool axis. This means that surface vectors only have to intersect the tool bottom [19].

Although technically no longer a Z-buffer, the point set is used in a similar manner. As in Z-buffer approaches, each successive tool movement is intersected with the surface vectors, reducing the height when an intersection occurs. Although a regular grid is no longer used, efficient computation is achieved by placing points into a grid of buckets, essentially a 2D hash table. Tool movements then find the buckets they intersect and perform intersection calculations with the vectors in those buckets only.

This method sidesteps the inverse point problem by generating accurately placed sample points on the surface, rather than by projecting points from a fixed grid. The variable spacing also ensures that the number of points used to represent the surface and workpiece is sufficient and not excessive. On the downside, the variable spacing makes material removal calculation more complicated. In addition, since Z aligned vectors are being used, error estimates are affected by the angle between the surface normal and Z direction.

Surface Normal Algorithms

Early on, Chappel used the point-vector approach to surface representation just described [6]. However, in Chappel's work, each vector uses the surface normal for orientation. Points are chosen on the part surface. At each point normals are extended both into the part until another part or face is reached, and out of the part until another part, face or the end of the stock is reached. The tool movement is simulated as a series of static tool positions. Cutting is done by finding the intersection of the vectors and cylinder positions. From this, interference and gouging are reported. This method allows 3-, 4-, and 5-axis machining to be simulated. Chappel doesn't discuss how surface points are chosen, nor how tool positions are selected.

Oliver and Goodman [25] take this approach further. They also use the surface normal at each point on the surface as the vector to be cut. The point selection is done visually by picking an orientation for the surface. The pixels are then projected onto the part surface, whereupon the normal is found and used as the vector. Then simulation proceeds as previously described. By using surface normals, five-axis machining is more easily simulated. In addition, the error at a point is now correct. However, by picking their points using a view-based method, they experience the problems due to the grid accuracy, the view direction chosen, and having to solve the inverse point problem.

Jerard and Drysdale [22] extended their point set approach to use normal vectors in order to allow five-axis machining. Points are still on the surface, spaced properly to guarantee the accuracy of the simulation, thereby avoiding the inverse point problem and ensuring that there are just enough points needed to simulate at the desired accuracy and no more.

In addition, they addressed the issue of localization for five-axis simulations. The problem here is that with normals pointing all over instead of in a single direction, two-dimensional bucketing no longer works correctly. A point may reside in one bucket, but its normal cuts through several. [22] suggests two methods of dealing with this.

The first method, called the short normal method, is to use a short vector at each point. Instead of having a long vector that extends to the ends of the stock material, the vector is limited to a small distance above and below the surface. This allows the points to be placed into buckets again. Then, when a tool envelope is compared to the bucket boundaries, the envelope is expanded by the short normal length to guarantee getting all points that may intersect the tool movement.

The second approach, average normal method, uses a set of preselected directions to serve as normals. Each direction has associated a bucket set. Each point uses the normal whose direction is closest to its own, and then is put into the bucket set for that direction. Cutting is then done for each direction. This takes advantage of having parallel vectors while reducing the errors that result because the real normals are close to the chosen direction. In theory, the direction vector is an average of the surface normals of the points it represents. Also, even though the cutting process is repeated several times, the time required doesn't increase because each point is only placed into one bucket set. The total number of intersections performed doesn't increase.

One final discrete modeling method was proposed by Ozair [9, 27]. He applies to NC verification the common technique of approximating surfaces by triangles. By doing this, many fewer primitives can be used in very flat regions. While surface points must be close enough together to prevent the tool from sticking down into the surface undetected, only three are needed to define a triangle to represent the same region. Triangle representation is very useful for flat regions, but suffers if the surface is highly curved in a lot of places, because the cost of performing intersections is higher. One deficiency of the method is that excess material can no longer be effectively tracked. In addition, a gouge is associated with a triangle, so if the corner of a large triangle is nicked, the whole region is flagged as gouged.

1.3 Modeling Five-Axis Tool Movements

All the surface and material representation methods mentioned above work well with three-axis tool movements. This is not surprising. Chapter 2 gives the mathematical description of a three-axis tool movement, but basically, it consists of two end cylinders, two side parallelograms, and a top and bottom tube swept out by the cylinder end discs. Finding the intersection of a line and this object, essentially what most of the reviewed verification methods do, is simple, straightforward and exact to the limits of the machine's precision. The volume is easy to work with because it is convex.

This means that the intersection step does not have to be taken into account when determining the accuracy of simulation and verification.

None of this is true anymore when five-axis tool movements are considered. The addition of rotation to the linear motion of the tool allows nonconvex cutting to occur. This explains why some researchers avoided five-axis cutting when designing their verification methods. It requires the ability to store multiple values in Z-buffer based algorithms to account for the possibility of a single cut leaving a portion of a line, not to mention multiple cuts leaving several disconnected pieces of the line.

The volume occupied by the tool as it moves is often referred to as a swept volume. The motion itself can be referred to as a sweep. Sweeps have a number of different applications to CAD, prompting research into good sweep representations.

1.3.1 Sweep Representation

Looking for a way to detect collisions between moving objects, Ganter and Uicker [13] computed intersections between the paths swept by the objects involved. The objects are taken to be polyhedra and then a polyhedral mesh is built to represent the sweeps. Their method places a solid at prespecified points along its path of motion and creates a "silhouette" at each one. The silhouette is the set of edges that make up the exterior of the projection of the solid in the direction tangent to the instantaneous tool path. These silhouettes are then joined to form a mesh representing the envelope. The resulting polyhedra can then be intersected to locate regions of possible contact.

Polygonization makes things easier to work with, but the actual object and surface information is lost when making the conversion. Other research has looked at representing sweeps without resorting to discretization. Weld and Leu [36] present a result showing that a swept volume is equivalent to the union of the swept volume of the surface of the solid being swept and an instance of that solid. This is shown for general n dimensions. They use this to generate swept surfaces for polyhedral sweeps. creating the swept volume by computing the swept volumes of the faces. The total volume is the union of the volumes swept by the faces.

Sambandan [31] employs envelope theory to generate sweep representations for polygonal objects swept in two dimensions and polyhedral objects in three dimensions. He takes a similar tack to Weld's work. but goes further. The faces sweep out the volume of the external portion of the sweep envelope. but they in turn can be broken down. The sweep of a polygonal face is created by two parts. The edges of the face sweep out most of the surface. while the interior sweeps out the remainder. creating what is known as a developable surface.

Instead of taking all of the possible contributions to the final surface, Sambandan uses the theory of envelopes to determine critical curves for the various components of the polyhedron, which happen to be line segments due to the linear nature of everything involved. The component is swept, and when there is a critical section on the component, it is included in the candidate set. From this set of sweeps. sweeps that actually lie on the surface are chosen and the remainder are removed. Then everything can be trimmed to produce the actual surface.

Key to the accuracy of Sambandan's method is choosing the instances at which to evaluate the critical curve boundaries used to decide valid sections of the component's sweep. However, he doesn't explore how to actually determine them.

Pegna [28] works with the theory of envelopes in a somewhat different manner. He takes an n-dimensional object and represents the sweep as an $n+1$-dimensional object. The envelope is obtained by projection of the sweep back into n dimensions. Representation in this manner allows compact generation code, but makes working with the sweep surface, which is needed for intersection calculation. difficult.

Blackmore and Leu [3] analyze sweeps in the context of differential equations.

They associate a sweep differential equation with each sweep which represents the trajectories of the points that make up the object. The association of a differential equation allows them to analyze further the swept volumes of certain classes of motions and objects. The type of sweep done in five-axis tool movements doesn't fall under their classification of easily analyzable sweeps.

1.3.2 Five-Axis Tool Representation

Directly tackling the problem of five-axis tool movement representation, Wang and Wang [35] derive a parametric description of the surface swept by the side of a cylinder using envelope theory. However, they do not say what method is actually used to represent the tool movement and how intersections are then found. Sambandan [30] uses the theory of envelopes to derive equations describing all surfaces generated by a cutting tool in a five-axis sweep. He also gives equations to describe fillet end and ball end cutters. The sweep is polygonalized and then scan line techniques are used to render the polygonalized sweep into a modified image buffer. Boolean subtraction is used to perform the simulation in image space.

Jerard and Drysdale [22] describe an approximation to five-axis tool movements consisting of numerous short three-axis tool movements. The interval of the sweep is chopped up into segments, each represented by a three-axis tool movement using the orientation of the cylinder axis at the start of the segment. Intersection is accomplished by intersecting the line with each three-axis volume in turn. This avoids numerical computation code at the expense of numerous simple intersections.

Chang and Goodman [5] offer a method for finding intersections not based on envelope theory. They derive the equations to determine when a surface point is inside the cutter volume as a function of t. This is accomplished by looking at the perpendicular from the point to the ruled surface swept out by the tool axis. Not revealed by the symbolic form of the equations listed. this perpendicular is a nonlinear equation involving trigonometric functions. The cutting test is done by checking whether a point satisfies two inequalities that depend on this perpendicular function. Chang and Goodman don't expand these inequalities to reveal their nonlinear nature. nor do they explain how these inequalities are solved. Cutting depth is then determined by localization followed by calculation of distance equations that also depend on the equation for the perpendicular. Chang and Goodman claim that their method is as exact as machine tolerance allows but make no explanation of how this is achieved.

Narvekar [24] also uses the theory of envelopes to generate surface equations for general APT tools. These are used to create systems of three nonlinear equations in three unknowns which are then solved with Newton-Raphson iteration to find solutions. However, Newton-Raphson iteration usually needs to be near a solution to be able to converge (see [29, Ch. 9]). Narvekar makes no attempt to try to localize the vicinity of intersection solutions, nor to distinguish between possible multiple solutions.

Narvekar also proposes a simplified method for finding intersections for a restricted class of tool movements. Based on the critical curve, his approach uses a one-dimensional root finder to locate the intersection point. Once again, nothing is said about guaranteeing finding an answer.

None of the research mentioned actually looks at the issues of guaranteeing that the intersection calculation be accurate to some degree. Some offer the equations describing the surfaces or other formulation of the intersection and then indiscriminately use a root finder to locate the intersection point. However, this cavalier application of a useful numerical tool overlooks its limits. Root finders usually need to know that a root is likely to be in the general area before they can successfully find it. Without this knowledge, using one is just a shot in the dark.

Another problem is that accurate location of intersections requires finding all roots in the interval or region of concern. While a root finder may converge on one of the roots, others may go undetected. So, the lax use of root finders amounts to applying local search techniques (finding a root) to a global search problem (finding all roots in the region).

In this thesis, we will present several different approaches to representing five-axis tool movements in a manner that allows intersections with lines to be computed. We contribute two completely new techniques. In addition, for each of the proposed methods, we give techniques by which these intersections may be found with guarantees on the accuracy of the answer. Finally, we look at the performance of implementations of each of the methods on test data and real data.

The remainder of this thesis is organized as follows. The mathematical model of three- and five-axis tool movements is laid out in Chapter 2 along with a definition of tolerance and other general information. Chapter 3 looks further at the method originally proposed by Jerard and Drysdale and derives error bounds. Chapter 4 develops a method that involves solving several one-dimensional equations in order, not requiring multidimensional techniques. Chapter 5 explores the construction of a polygonal approximation to the swept envelope with guarantees on the error of the approximation. Chapter 6 describes a new method in which the intersection is found by testing points on the line in order until a point is located that lies with the envelope of the tool movement. Finally, Chapter 7 looks at actual performance data generated from tests varying parameters and real data. We analyze the data and offer final conclusions, both theoretical and practical.

Chapter 2

Preliminary Math and Notation

Before describing the different methods for finding intersections between lines and five-axis tool movement sweeps, we need to describe the actual tool movements. In addition, we describe the meaning we attribute to error in this thesis.

2.1 Cylinder Definition

A cylinder will be specified in this thesis by a point c, a unit vector a, a radius R, and a length L. The point $c = (c_x, c_y, c_z)$ is the base point of the cylinder. The unit vector $a = (a_x, a_y, a_z)$ is the axis vector of the cylinder, describing its orientation. The cylinder axis extent is from c to La. The body of the cylinder will simply be a cylinder of radius R around the axis as defined.

2.2 Three-Axis Tool Movements

A three-axis tool movement in machining allows the cutting tool three degrees of freedom. These degrees of freedom correspond to translation along the three cartesian axes. One degree allows the tool to slide up and down. The other two provide translation to the side in two directions. In addition, the motion is defined in particular by a starting tool position c_i and a finishing tool position c_f. The axis vector a stays fixed for the duration of the motion. The tool linearly translates with time from one position to the other. Thus the equation of motion for the base of the cylinder is

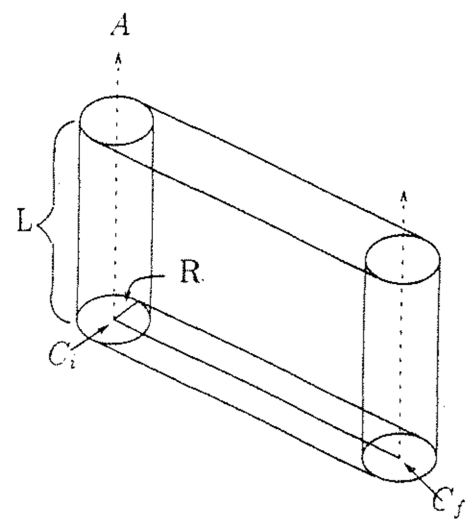


Figure 2.1: A three-axis tool movement

$c(t) = c_i + (c_f - c_i)t$ with t being the time parameter over the interval $0 \leq t \leq 1$.

2.3 Rigid Sweeps

Throughout this paper, we will be concerned with rigid sweeps of objects. Simply put, a rigid sweep is any transformation over time that does not change the shape of the object. only its location and orientation. In R^3, the only two transformations that preserve distance between points (thereby guaranteeing shape consistency) are rotations and translations. Any composition of translations and rotations can be put into either homogeneous form,

$$p' = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} p,$$

or nonhomogeneous form,

$$p' = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} + \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} p.$$

2.4 Five-Axis Tool Movements

A five-axis tool movement in this paper is the same as a three-axis tool movement except that two extra degrees of freedom are added to the allowed transformations. These two degrees of freedom correspond to rotation around two different axes in space, permitting arbitrary rotation. The center of rotation occurs at the base point of the cylinder.

The five-axis movement is defined by a starting cylinder position (c_i, a_i) and an ending position (c_f, a_f). The base of the cylinder translates linearly just as in the case of a three-axis motion. The axis of the cylinder changes orientation from a_i to a_f along a great circle. This means that the rotation occurs about the cross product of the two vectors $a_i \times a_f$. The angle of the cylinder axis to either a_i or a_f changes linearly. Therefore we have the following two equations to describe a five-axis tool movement over the interval $0 \le t \le 1$:

$$c(t) = c_i + (c_f - c_i)t \tag{2.1}$$

$$a(t) = A(t) \cdot a_i. \tag{2.2}$$

where A(t) is the rotation matrix [11, pg. 73]

$$\begin{pmatrix} n_x{}^2 + \cos\theta t(1 - n_x{}^2) & \begin{matrix} n_x n_y(1-\cos\theta t) \\ +n_z \sin\theta t \end{matrix} & \begin{matrix} n_x n_z(1-\cos\theta t) \\ +n_y \sin\theta t \end{matrix} \\ \begin{matrix} n_x n_y(1-\cos\theta t) \\ +n_z \sin\theta t \end{matrix} & n_y{}^2 + \cos\theta t(1 - n_y{}^2) & \begin{matrix} n_y n_z(1-\cos\theta t) \\ +n_x \sin\theta t \end{matrix} \\ \begin{matrix} n_x n_z(1-\cos\theta t) \\ +n_y \sin\theta t \end{matrix} & \begin{matrix} n_y n_z(1-\cos\theta t) \\ +n_x \sin\theta t \end{matrix} & n_z{}^2 + \cos\theta t(1 - n_z{}^2) \end{pmatrix}$$

about the normalized vector $n = (n_x, n_y, n_z) = \frac{a_i \times a_f}{|a_i \times a_f|}$ where $\theta = \arccos(a_i \cdot a_f)$ is the total angle of rotation of the tool movement.

The equations just outlined are for the general five-axis tool movements we will consider. For the purpose of computation, we can get much simpler equations by transforming the problem.

First, we can assume that $c_i = (0, 0, 0)$ by doing a simple translation of the problem. Since c_f is the only non-zero end of the linear portion of the tool movement, we will refer to it as $c = (c_x, c_y, c_z)$ from now on. The remainder of this dissertation will work in a left-handed coordinate system, illustrated in figure 2.2. Next, assume that a_i and a_f are constrained to the XY plane. This simply rotates the problem so that $n = (0, 0, 1)$. If we choose $a_i = (1, 0, 0)$, we can further simplify the problem and now only need an angle θ to determine the rotational component of the motion. Having made these choices, which amount to rigidly transforming the problem, we get the following equations to describe the cylinder position:

$$c(t) = \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} t \tag{2.3}$$

$$a(t) = \begin{pmatrix} \cos\theta t & -\sin\theta t & 0 \\ \sin\theta t & \cos\theta t & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \cos\theta t \\ \sin\theta t \\ 0 \end{pmatrix} \tag{2.4}$$

The points on the cylinder, as well as any other point p, will then undergo the transformation

$$p' = \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} t + \begin{pmatrix} \cos\theta t \\ \sin\theta t \\ 0 \end{pmatrix} p \tag{2.5}$$

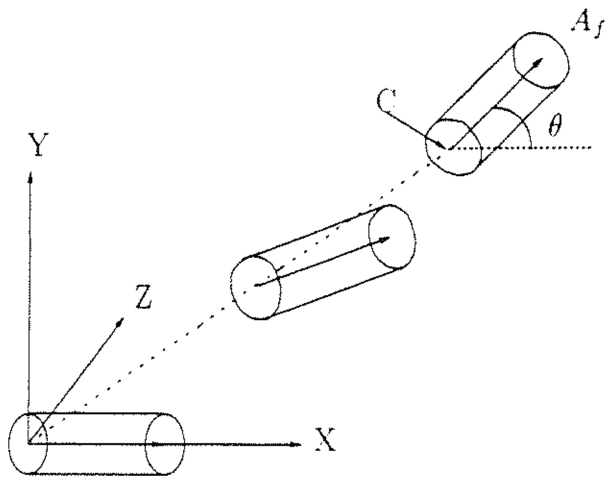


Figure 2.2: A five-axis tool movement

This produces a helix-like motion we could refer to as a "helicoid," where the linear translation is not necessarily perpendicular to the plane of rotation. The path of any point on the cylinder axis will look like a segment from the coil of a "Slinky" toy that has been squashed to one side. Note that if $c_x = c_y = 0$, the motion produced will indeed follow a helix.

2.5 The Meaning of Error

Throughout this document, we will refer to error when trying to determine how good an approximation of five-axis tool and line intersection really is. One possible measurement of error is the distance of the returned intersection point from the true intersection point. This has two obvious problems. One, if the intersection routine being used fails to find an intersection at all, the error to return is undefined or infinite, neither of which is a very satisfying answer. Second, when the line and cylinder axis are nearly parallel, small displacements result in huge shifts in the location of the intersection. Thus the error depends highly on the relative orientation of the tool and line.

We would prefer a measure of error that better reflects the geometric situation. For instance, if the routine cannot find an intersection, we would hope that the line does not run right through the middle of the tool. Also, when the line and tool are almost parallel, small changes in the relative orientation and position only cause small changes in the depth of the line beneath the tool's surface.

As long as our approximation model is geometrically near the true tool movement envelope, these concerns will be addressed. Therefore, throughout the rest of this dissertation, tolerance will be measured in terms of the distance between the true surface and the approximation surface. As long as we can move the approximation some distance less than the tolerance and obtain the correct cut depth, we will say that the approximation is a good representation.

2.6 Useful Formula

$a \cos \theta t + b \sin \theta t = c \cos(\theta t + \delta)$ where $c = \sqrt{a^2 + b^2}$ and $\delta = \arctan(-\frac{b}{a})$. This is obtained by first converting to exponential form

$$\begin{aligned}
a\frac{e^{i\theta t} + e^{-i\theta t}}{2} + b\frac{e^{i\theta t} - e^{-i\theta t}}{2i} &= \frac{1}{2}\left((a - ib)e^{i\theta t} + (a + ib)\epsilon^{-i\theta t}\right) \\
&= \frac{1}{2}\left(\sqrt{a^2 + b^2}\epsilon^{i \arctan - \frac{a}{b}} e^{i\theta t} + \sqrt{a^2 + b^2}\epsilon^{i \arctan \frac{a}{b}} \epsilon^{-i\theta t}\right) \\
&= \frac{1}{2}\left(\frac{\sqrt{a^2 + b^2}}{2}\epsilon^{i(\theta t \arctan - \frac{a}{b})} + \sqrt{a^2 + b^2}\epsilon^{-i(\theta t - \arctan \frac{a}{b})}\right) \\
&= \sqrt{a^2 + b^2} \cos(\theta t - \arctan \frac{a}{b})
\end{aligned}$$

Chapter 3

Bounding Three-Axis Subdivision of a Five-Axis Sweep.

3.1 Method

One way to approximate a five-axis sweep of a finite cylinder is by stringing together a series of small three-axis motions that guarantee that the accuracy of the approximation is within a given tolerance T (See Figure 3.1). This method was first suggested by Jerard et al. in [22]. We explain the method and give an error bound to allow determining the amount of subdivision needed for a desired level of accuracy.

The technique for approximation is straightforward. Recalling equations 2.3 and 2.4, we have a rotation of θ and a linear translation of $|c|$. We would like to break the motion up into a set of n segments. Since the interval for the whole motion is $0 \leq t \leq 1$, each segment will occupy a period of time $\Delta t = \frac{1}{n}$. For segment 1, the position of the cylinder is

$$\begin{aligned} c(t) &= (\frac{c_x t}{n}, \frac{c_y t}{n}, \frac{c_z t}{n}) && (3.1) \\ a(t) &= (\cos\frac{\theta t}{n}, \sin\frac{\theta t}{n}, 0) && (3.2) \end{aligned}$$

Note that it doesn't matter if the interval is $0 \leq t \leq 1$, angle is θ/n, and end point is $(\frac{c_x}{n}, \frac{c_y}{n}, \frac{c_z}{n})$, or the interval is $0 \leq t \leq \frac{1}{n}$ and angle and end point are θ and (c_x, c_y, c_z). Therefore any segment can be looked at as a complete movement by itself. Other segments can be transformed to the same coordinate system using the appropriate

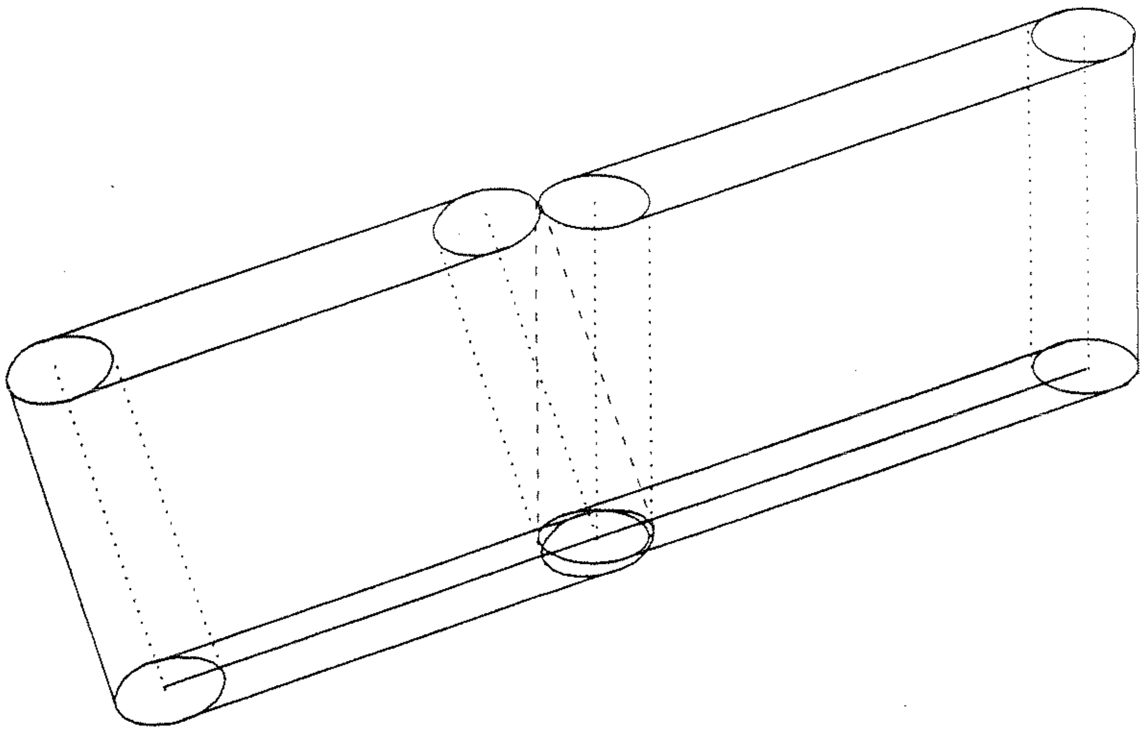


Figure 3.1: Consecutive three-axis movements

translation and rotation. If we laid each of these segments end to end, we would rebuild the entire motion (ignoring the overlapping ends).

Creating the approximation is trivial. Each five-axis segment is replaced by the corresponding three-axis movement in which the axis stays fixed at its starting value. For segment i, $a_i(t) = a(\frac{i-1}{n})$. The linear translation remains the same.

Finally, to approximate the intersection of a line with the five-axis sweep, we successively find the intersections of the line with each of the three-axis segments previously created. The closest intersection point found is returned as the answer we seek.

This leaves the question of how to guarantee that our approximation is accurate to within tolerance T.

3.2 Bounding three-axis Approximation Error

We proceed by looking at the vector field of points undergoing the motion in question starting at $t = 0$. We only need to consider motions starting at $t = 0$ since we can transform any motion to the alignment we desire. Any point (x, y, z) subject to this rigid motion will be transformed by the following

$$p_5(t) = \begin{pmatrix} x_5(t) \\ y_5(t) \\ z_5(t) \end{pmatrix} = \begin{pmatrix} c_x t + x\cos(\theta t) - y\sin(\theta t) \\ c_y t + x\sin(\theta t) + y\cos(\theta t) \\ c_z t + z \end{pmatrix}$$

A three-axis movement with the same translation component motion gives us the following transformation

$$p_3(t) = \begin{pmatrix} x_3(t) \\ y_3(t) \\ z_3(t) \end{pmatrix} = \begin{pmatrix} c_x t + x \\ c_y t + y \\ c_z t + z \end{pmatrix}$$

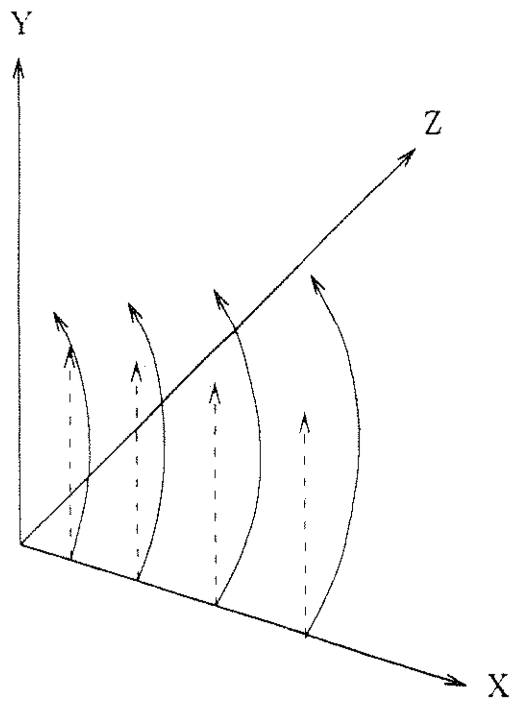


Figure 3.2: Paths of points undergoing sweep. Solid lines show actual sweep paths. dashed lines show linear approximation.

We can then be sure that the true sweep differs from the approximation by no more than

$$\begin{aligned} d &= |p_5 - p_3| \\ &= \sqrt{(x\cos\theta t - y\sin\theta t - x)^2 + (x\sin\theta t + y\cos\theta t - y)^2} \\ &= \sqrt{2x^2 + 2y^2 - 2x^2\cos\theta t - 2y^2\cos\theta t} \\ &= \sqrt{(2x^2 + 2y^2)(1 - \cos(\theta t))} \end{aligned}$$

We can consider this distance to be the tolerance of the approximation. It is obvious that a bound on this distance will bound the error in the approximation. because any point in the approximation has a corresponding point in the real sweep no further away than this distance and vice versa.

If we choose $\theta \leq 90°$, $t = 1$ will generate the largest distance over the interval $0 \leq t \leq 1$ and represent the maximum error of the approximation. Setting $t = 1$, we have

$$d = \sqrt{(2x^2 + 2y^2)(1 - \cos(\theta))}.$$

We would like to determine step size based on a user specified tolerance T. If we solve the above equation for θ, we will get θ as a function of d. This will allow us to determine appropriate values of θ for given tolerances. The new function is

$$\theta = \cos^{-1}\left(1 - d^2/2(x^2 + y^2)\right).$$

If we choose d to be the user specified tolerance T, this will mean that θ is controlled by x and y. Obviously, the worst case occurs when the point (x, y, z) is as far from the origin as possible. If we substitute in the worst case values (for a cylinder positioned as stated previously, these would be $x = L$ and $y = R$), we get for θ

$$\theta = \cos^{-1}(1 - T^2/2(L^2 + R^2)).$$

For example, if L=4, R=.5 and T=.002 then

$$\theta = .028°.$$

This means we need about 35 linear steps per degree of rotation to achieve this accuracy for the given cylinder size.

We can cut this requirement in half by taking advantage of the symmetry of the tool movement. The tool has the same type of motion in the $-t$ direction that it does in the t direction. To see that the $-t$ direction is subject to the same bounds, we can start by looking at $p_5(t)$ at $-t$. We find that

$$p_5(t) = \begin{pmatrix} -c_x t + x\cos(\theta t) + y\sin(\theta t) \\ -c_y t - x\sin(\theta t) + y\cos(\theta t) \\ -c_z t + z \end{pmatrix}$$

A three-axis movement at $-t$ gives

$$p_3(t) = \begin{pmatrix} -c_x t + x \\ -c_y t + y \\ -c_z t + z \end{pmatrix}$$

Repeating the distance calculation, we get

$$\begin{aligned} d &= |p_5 - p_3| \\ &= \sqrt{(x\cos\theta t + y\sin\theta t - x)^2 + (-x\sin\theta t + y\cos\theta t - y)^2} \\ &= \sqrt{2x^2 + 2y^2 - 2x^2\cos\theta t - 2y^2\cos\theta t} \\ &= \sqrt{(2x^2 + 2y^2)(1 - \cos(\theta t))} \end{aligned}$$

As we can see, the same expression occurs in the negative direction as in the positive.

Therefore, a sweep in the negative t direction can be approximated by a three-axis tool movement with the same cylinder orientation and starting point as the positive t direction. Since the two three-axis tool movements would join seamlessly, we can in fact double the size of five-axis tool movement that can be approximated by a three-axis tool movement by simply choosing the sampled tool position to be in the middle of the sweep instead of at one end as before.

This method of bounding three-axis approximation can apply to any tool shape desired. If the tool shape is projected onto the XY plane, the furthest point of the projection from the origin will be the worst case. This point can then be used in the above equation to determine the maximum rotation allowable for the given tolerance.

The method has at least two obvious disadvantanges. One, as the above example shows, a large number of steps will be needed to achieve good accuracy. Two, the error bound depends on the amount of rotation θ, but has no dependence on the linear portion of the tool movement. In the most extreme case, if $c_z = 0$, the sides of the envelope are flat and no errors are introduced there by the approximation. This sort of motion is sometimes referred to as a four-axis tool movement. The only errors are created at the two ends of the cylinder. In these cases, the error is probably less than the bound would indicate. In addition, a better approximation can be had by using one plane to represent the sides of all the tool movement segments.

3.3 Optimization

This model has the advantage of being uncomplicated. It is quite simple, both to describe and to program. However, it is apparent that, as described in [22], it isn't very efficient. Every sub-movement has to be intersected with the line to get the final answer.

One solution is to apply a space subdivision scheme to this model. One general approach is to partition the volume of space containing the set of three-axis submovements into disjoint boxes. Each box contains pointers to the three-axis submovements that overlap the box's volume of space. Then, we find the intersection by determining the first box that the line hits, if any. Each submovement contained in that box is intersected with the line. If no intersection is found, the next box in the line's path is explored, and so on until an intersection is found, or all boxes in the line's path have been processed.

There are several variants of this style of space subdivision available. The more common ones are octrees, binary space partition (BSP) trees, and uniform grids [15]. In an octree or BSP tree, each volume node of the tree is subdivided until a reasonable number of objects lie in the cube it represents. The uniform grid simply divides space into a set number of cells in each dimension.

Octree-style subdivision normally works very well when objects are spread out, but in the case of the model we are dealing with, we have many objects that closely overlap each other. Each three-axis movement is likely to have a short linear movement, meaning that the end cylinder is almost on top of the starting cylinder. It also means that the starting cylinder of two successive movements lie nearly on top of each other. It is easily possible for hundreds of cylinders to overlap each other. This means that each cube will contain lots of cylinders because subdivision cannot separate the overlap. Even in cubes where it can, it will require a lot of subdivision to cut down the list sizes, leading to potential memory problems as well as performance loss.

It is clear that just chopping up space into partitions won't work very well. However, the model does stand to benefit from separation into sections. There is no need to have the line intersect every submovement when it only runs near a few of them.

Another approach to subdividing space is to build a hierarchy of bounding volumes. Each object is surrounded by a bounding volume, then groups of bounding volumes are surrounded by bigger volumes, repeatedly until all the objects are encased in one large bounding volume. The bounding volumes are allowed to overlap. This structure has the advantage of being able to separate overlapping objects, at the price of possibly having to explore a more complicated path through the hierarchy. An intersection in one node is not guaranteed to occur before the start of another node's volume.

In our case, this is a very convenient way to structure things. Each three-axis movement is already convex. Two consecutive movements can be approximated by a tool movement that uses the average of the two movements' cylinder axes. Therefore we can create a tool movement using the average axis and make it just large enough to contain both tool movements. This gives a fairly tight bounding volume for the two tool movements, which is desirable to avoid too many unnecessary intersections. The closer the two axes are, the better the bounding volume. The hierarchy is built by placing pairs of successive tool movements into three-axis bounding volumes, creating a binary tree.

This tree is used to facilitate the search for the intersection point. The algorithm to find the intersection is a modified binary search, as follows: First the line is intersected with the top volume. If a hit is found, the search continues. Each of the children are intersected. If either one is hit, the closer child is searched recursively. An actual intersection value is returned when the bottom of the tree is reached. If this value is closer than the second child, the search may be cut off and the value returned to the next upper level. Otherwise the second child is searched similarly.

3.4 Analysis

The performance of intersection-finding using the three-axis approximation depends on a few variables. First, it depends on the angle of rotation. Next, it depends on the furthest point of the tool from the origin, $L^2 + R^2$. And finally, it depends on the tolerance desired.

It should be obvious that the dependence on the angle of rotation is linear. The total number of tool movements is simply the total rotation divided by the maximum approximated rotation. Then, every tool movement is intersected with the line. So, a doubling of the angle doubles the number of tool movements used and doubles the number of intersections performed. This means that both preprocessing and intersection times should exhibit linear response to rotation angle.

The dependence on the tool size and tolerance is not so obvious. They can really be treated as one factor due to their simple relationship in the angle equation. Double the tolerance and double the maximum distance tool point, and the same situation results. Therefore we can consider the expression $T^2/(L^2 + R^2)$ as a single variable referred to as relative tolerance. Relative tolerance is just tolerance in terms of the tool size.

It turns out that the equation is not quite linear as a function of relative tolerance. However, it is close enough to linear to work with. We can see this from the graph of the angle function over relative tolerances from 0 to the size of the tool (see figure 3.3).

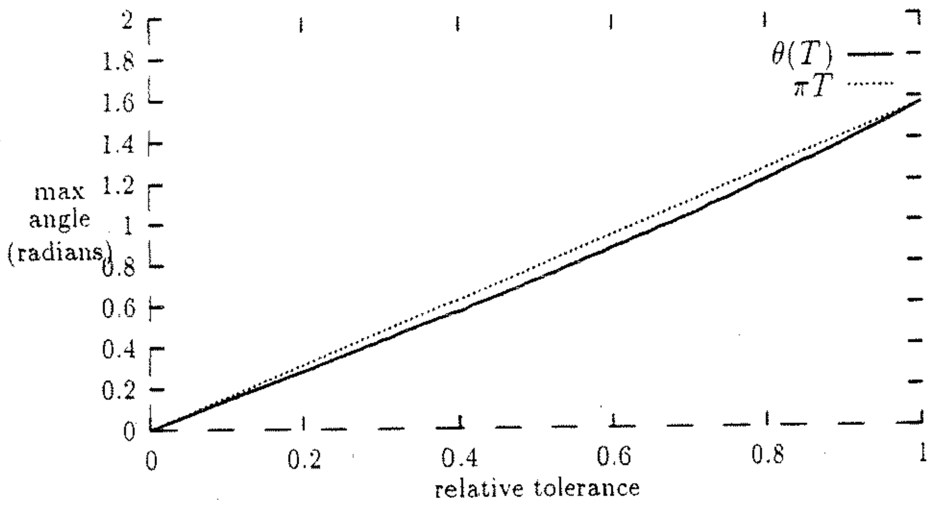


Figure 3.3: Graph of θ vs. relative tolerance

The case of the hierarchical bounding volume optimization performance is slightly more complicated. Preprocessing will be $O(n)$ with respect to tool movement rotation angle and tolerance since a binary tree structure to represent the bounding volume hierarchy is built from the bottom and will contain twice the number of submovements needed to represent the tool movement. Intersection time isn't guaranteed to be $O(\log n)$. Since intersections may require traveling down both branches of any given node, performance could be anything all the way up to $O(n)$. There are certain cases where this can happen. For instance, if we have a very short tool movement with very little rotation, a very large tool, and high tolerance, all the submovements will overlap highly. A line coming straight up from the bottom will hit every submovement, and thus may require travel down most of the tree. As we shall see in Chapter 7 though, this optimization works quite well. Most of the time, intersections only require traversing a limited path in the hierarchy.

3.5 Practical Considerations

From a practical point of view, the three-axis approximation method with the hierarchy is excellent. The method is simple and easy to implement. The performance as shown in Chapter 7 is outstanding. In addition, the framework of the method is valid for any tool shape for which a three-axis intersection routine exists.

The one concern is memory usage. In testing, memory usage was not a problem, but with high enough accuracy demands, it will become one. If need be, any node of the hierarchy tree can be calculated on the fly. The regularity of the movements in the tree results in needing only a constant amount of information.

Each bottom leaf submovement is easily calculated in constant time. The only concern is calculation time of upper nodes, since they are based on the complete subtree beneath them. However, since the movement is chopped up by powers of two, everything is regularly spaced, meaning that any node's bounding box can actually be determined by looking only at the two extreme leaf submovements, rather than the complete subtree. Only a constant amount of calculation is required to generate any node in the tree. Therefore the program can be refitted to run with minimal memory requirements at the expense of extra calculations while running.

Another approach to dealing with the problem is to divide all movements into smaller movements beforehand. Since the accuracy is known at the start, the maximum rotation is known and the maximum tree size is known. Therefore, any tool movement that requires too large a tree may be broken up into smaller pieces and placed back on the list. Then all movements will be guaranteed to avoid memory overruns.

A third way to handle this is to allow for more than two children in each node. This will reduce memory requirements and may even speed up the intersection process. Some combination of these methods may yield the best way to handle memory problems.

Chapter 4
A Numerical Approach

A simple method of approximating the intersection of a five-axis tool movement and a fixed line would be to choose a set of times over the interval [0.1] and find the intersection of the line with the cylinder at each of these times. The first intersection point along the line can then be returned as the envelope intersection. If we keep increasing the number of times chosen, we will approach a continuous function. This leads to the possibility of finding the minimum of the intersection of line and cylinder as a function of t and returning that as the intersection point. With enough accuracy in locating the global minimum, the method could be as exact as the computer's floating point limits allow.

4.1 Method

The basic idea behind this numerical approach is to take the equation of the intersection of the line $P = P_o + uV$ and the cylinder with base point C and axis direction vector A, and substitute the equations for the moving base and rotating axis in place of the stationary ones. Doing this makes the equation a function of t, and lets us find the minimum of this equation over the range $0 \leq t \leq 1$ to determine the closest point of intersection.

The stationary intersection equation [22] is:

$$u = \frac{(bd - c) \pm \sqrt{(bd - c)^2 - (1 - d^2)(a^2 - b^2 - R^2)}}{(1 - d^2)} \tag{4.1}$$

where

$$\begin{aligned} a &= |\mathbf{P}_0 - \mathbf{C}| \\ b &= (\mathbf{P}_0 - \mathbf{C}) \cdot \mathbf{A} \\ c &= (\mathbf{P}_0 - \mathbf{C}) \cdot \mathbf{V} \\ d &= \mathbf{V} \cdot \mathbf{A} \end{aligned}$$

The intersection point is given as a parameter along the line. We now make C and A functions of t. If we assume that the base of the cylinder starts at the origin (accomplished by a translation), then we have:

$$C(t) = C_f t \tag{4.2}$$

where C_f is the end point of the motion of the cylinder base point. We can also choose the axis of rotation to be the z axis and to have the cylinder axis lie along the positive x axis at the start of the motion. Then, the axis position is described by:

$$A(t) = \begin{pmatrix} \cos\theta t & \sin\theta t & 0 \\ -\sin\theta t & \cos\theta t & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \tag{4.3}$$

where θ is the angle of $A(1)$ to the x axis.

If we use these equations for A and C, then $u(t)$ gives the point of intersection along the line as t varies. Now all we need to do is find the point furthest in the $-V$ direction when $0 \leq t \leq 1$. For this, we only use the negative radical of $u(t)$.

$$\text{umin}(t) = \frac{(bd - c) - \sqrt{(bd - c)^2 - (1 - d^2)(a^2 - b^2 - R^2)}}{(1 - d^2)} \tag{4.4}$$

since the positive radical will give an intersection point further along the line in the positive V direction.

Ideally, we need only find the minimum of this equation. However, the equation is real-valued only when the fixed line intersects the cylinder. It is complex whenever the radical

$$\text{rad}(t) = (bd - c)^2 - (1 - d^2)(a^2 - b^2 - R^2)$$

becomes negative. This makes the job more complicated. We now need to identify the boundaries of regions of t where the line P actually contacts the cylinder. To do this, we find the roots of the radical as a function of t. Non-negative regions of the curve represent values of t where the line and cylinder intersect.

Another way to find the values of t when the line and cylinder intersect is to take the distance between $A(t)$ and the line P. This would give a signed distance as a function of t:

$$\text{dist}(t) = \frac{(P_0 - C(t)) \cdot (A(t) \times V)}{\sqrt{(A(t) \times V) \cdot (A(t) \times V)}} \tag{4.5}$$

with $\cdot$ indicating inner product and $\times$ denoting outer, or cross, product. With this equation, when $-R \leq \text{dist}(t) \leq R$, the cylinder and the line intersect. This will give us the same regions of t as the other approach.

We now have located regions of t where the line intersects the cylinder. Unfortunately, this only tells us that the line hits the infinite cylinder and we actually want to know that the line hits the finite length cylinder. To determine when this occurs, we use the following equation [22]:

$$\text{minpar}(t) = \frac{(b - dc) - d\sqrt{(bd - c)^2 - (1 - d^2)(a^2 - b^2 - R^2)}}{(1 - d^2)} \tag{4.6}$$

where a, b, c, and d are the same as for $u(t)$. This gives the position along the length of the cylinder at which the intersection occurred. Within the intervals found in the first step (guaranteeing the line hits the infinite cylinder), we find the intervals where $0 < \mathrm{minpar}(t) < L$. These intervals are regions where the line strikes the finite cylinder. Outside them (but inside the first set of intervals), the line misses. Once we have identified these regions, we find the minimum values of umin(t) over each region and take the smallest value.

The job isn't completely done yet. It is possible that the closest intersection may be with an endcap rather than with the cylinder body. For the line to hit the endcap at $l = 0$, it must strike the infinite cylinder beyond the extent of the cutting tool, meaning $l < 0$. Then it can hit the endcap. Finally, it must exit the cylinder with $l > 0$. The intervals that contain intersections of the cylinder and line that are excluded in the second step are the ones in which an endcap strike can occur. To locate regions where this happen, we look at the exit position of the line. If it exits the infinite cylinder on the opposite side of the endcap, the line must hit the endcap. The equation for the exit point of the line is:

$$\mathrm{maxpar}(t) = \frac{(b - dc) + d\sqrt{(bd - c)^2 - (1 - d^2)(a^2 - b^2 - R^2)}}{(1 - d^2)}$$

where a, b, c, and d are the same as for $u(t)$. To find the regions where the line hits the endcap, we must locate intervals of t with $\mathrm{maxpar}(t) > 0$ for the endcap at $l = 0$, and $\mathrm{maxpar}(t) < 0$ for the endcap at $l = L$. Again, these searches only occur within the regions of t included by the first search, but excluded by the second one.

If the cylinder axis becomes parallel to the line in the interval, minpar(t) will head to $\pm\infty$ and maxpar(t) will head in the opposite ($\mp\infty$) direction. Right around these infinities, an endcap is being intersected.

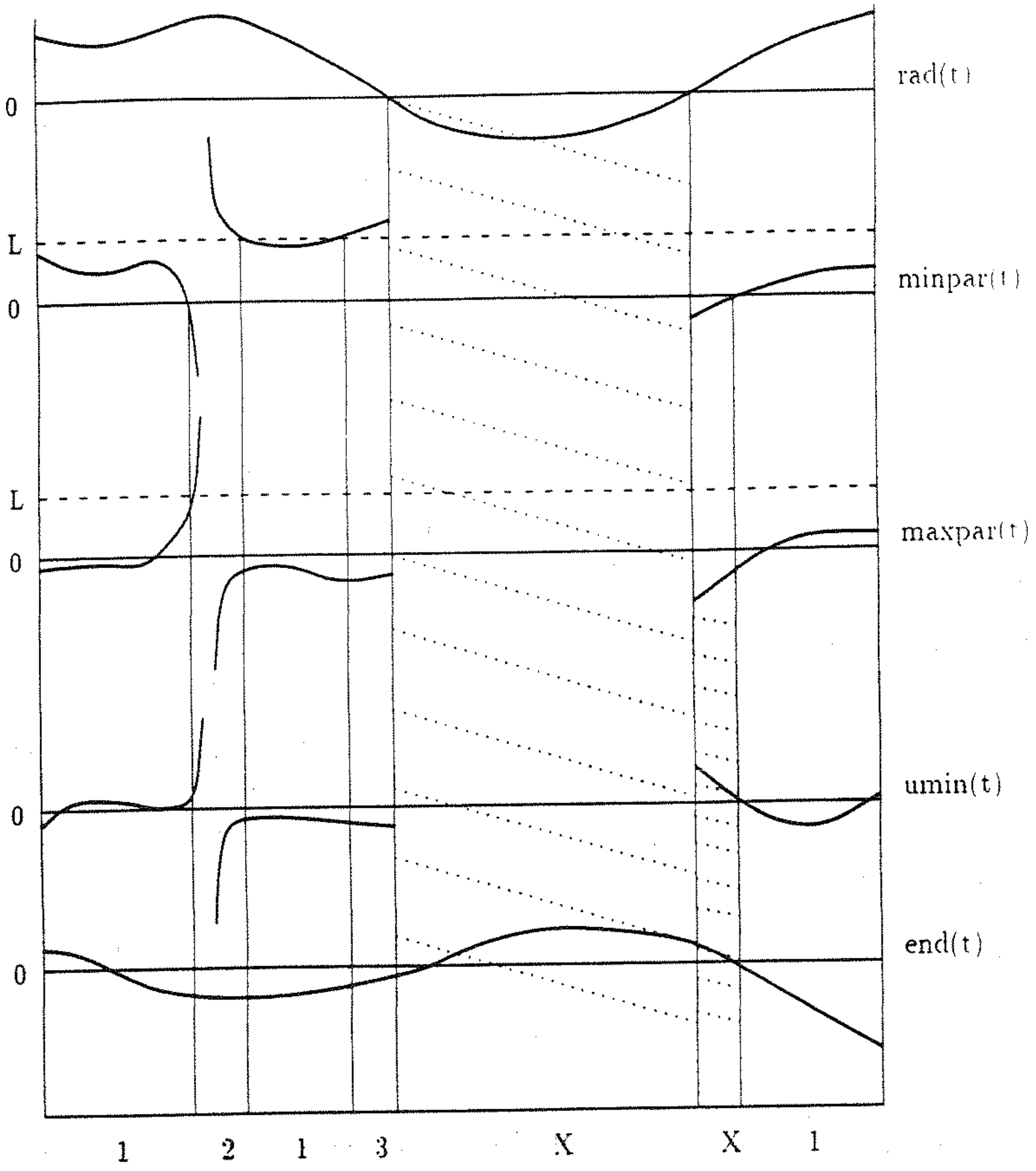
All the important intervals of t have now been identified. Intervals where the line does not intersect the infinite cylinder have been excluded by the first search. The second search determines intervals in which the line intersects the side of the finite cylinder. And the third search locates intervals in which the line intersects an endcap of the cylinder. With these intervals in hand, we have excluded all regions where the line does not intersect the finite cylinder and included the ones where it does.

The final step of the algorithm is to find the minimum intersection point in each valid interval and return the global minimum intersection as our answer. We have already seen the equation to do that when the line intersects the side of the cylinder. The endcap is simply a piece of the plane. Since the previous searches have guaranteed that the intersection is within the actual circle of the endcap, we only need to figure out the intersection of the line with the appropriate plane. For both endcaps, the plane normal is $A(t)$. The location of a point on the plane is $C(t)$ for the $l = 0$ cap and $C(t) + LA(t)$ for the other. The parameters of intersection of the line with these planes are [14]:

$$\begin{aligned}
\text{uend}(t) &= \frac{(C(t) - P_0) \cdot A(t)}{V \cdot A(t)} \quad \text{at } l = 0 \\
\text{uend}(t) &= \frac{(C(t) + L \times A(t) - P_0) \cdot A(t)}{V \cdot A(t)} \quad \text{at } l = L
\end{aligned}$$

Note that if the line is parallel to the cylinder axis, the denominator of umin(t) and umax(t) is 0. It is then trivial to trap this case and do an endcap intersection.

Figure 4.1 gives an example of the method. Where the topmost curve, rad(t), is less than 0, no intersection with the cylinder exists. This region is marked with the center X. In the remaining intervals, minpar(t) is evaluated. This gives 3 regions where minpar is greater than 0 and less than L. In these segments, umin(t) will be used to locate the intersection. These segments are labeled with 1 at the bottom of the figure. In the remaining intervals, maxpar(t) is evaluated. The region marked 2 has maxpar(t) on the opposite side of 0 from minpar(t). This region will be minimized for intersection using end(t). The region marked 3 is similar, with the two functions



rad(t)
0
L
minpar(t)
0
L
maxpar(t)
0
umin(t)
0
end(t)
0
1
2
1
3
X
X
1

opposed about L. EndL(t) is the minimizing function here. The remaining segment has both minpar and maxpar on the same side below 0 and does not intersect the cylinder. Finally, the minimum of the minimal value of umin(t) in regions 1, the minimal value of end(t) in region 2, and the minimal value of endL(t) in region 3, is returned as the intersection point.

4.2 Implementation and Bounds

The method we describe above is quite simple, conceptually, but carrying it out with a guarantee of accuracy is something else. To successfully implement the method, we need to find all the roots of rad(t), minpar(t), and maxpar(t) in the interval $[0, 1]$ or a subinterval. Then we must be able to find the **global** minimum of umin(t) and uend(t) over an interval.

Finding the roots of these equations is nontrivial. They each behave in a complex fashion, making an analytical approach to locating the roots difficult to devise. However, if we can find an upper and lower bound on the values of these functions over any given interval, a simple algorithm may be used to locate the roots.

To find the roots, all we need to do is look at the bounds of the function on the interval we desire. If both bounds are on the same side of zero, no root can exist in the interval and we're done. Otherwise, subdivide the interval into two halves, and check each half in the same manner. Once the interval is smaller than the machine precision limits allow, either return a root if the values have opposite signs, or no root if they have the same sign. It is possible to have a discontinuity in minpar and maxpar, but this can be detected by the fact that the values at the two ends of the interval will have a large magnitude.

The global minimum of a function may be found in a similar manner. The lower bound is used to determine if the interval can contain the minimum. If the lower bound is less than the current minimum, the interval is subdivided and each segment is searched. Otherwise, the minimum is elsewhere. Once the interval is too narrow, take the lesser of the two end values as the minimum. If that is less than the current global minimum value, the new value becomes the current global minimum. This process is continued until the complete interval has been evaluated.

4.2.1 Bounding the Equations

Bounds for rad(t)

The rad function used to determine intervals of intersection with the infinite cylinder is relatively simple. It consists only of factors and terms involving t, $\cos\theta t$, and $\sin\theta t$. With a bit of rearranging, it looks like:

$$\mathrm{rad}(t) = A + Bt + Ct^2 + (D + Et + Ft^2)\cos 2\theta t + (G + Ht + Jt^2)\sin 2\theta t \qquad (4.7)$$

with the constants being

$$\begin{aligned}
A &= \frac{1}{2}(-p_x^2 - p_y^2 + (p_x^2 + p_y^2 + p_z^2)(v_x^2 + v_y^2) - R^2(v_x^2 + v_y^2)) + R^2 - p_z^2 \\
&\quad + p_x p_z v_x v_z + p_y p_z v_y v_z + p_z^2 v_z^2 \\
B &= c_x p_x + c_y p_y + 2c_z p_z - c_x p_x v_x^2 - c_y p_y v_x^2 - c_z p_z v_x^2 - c_x p_x v_y^2 - c_y p_y v_y^2 - c_z p_z v_y^2 \\
&\quad - c_z p_x v_x v_z - c_x p_z v_x v_z - c_z p_y v_y v_z - c_y p_z v_y v_z - 2c_z p_z v_z^2 \\
C &= \frac{1}{2}(-c_x^2 - c_y^2 + (c_x^2 + c_y^2 + c_z^2)(v_x^2 + v_y^2)) - c_z^2 + c_x c_z v_x v_z + c_y c_z v_y v_z + c_z^2 v_z^2 \\
D &= \frac{1}{2}(p_x^2 - p_y^2 + (-p_x^2 + p_y^2 + p_z^2)v_x^2 + R^2(v_y^2 - v_x^2) + (-p_x^2 + p_y^2 - p_z^2)v_y^2) \\
&\quad - p_x p_z v_x v_z + p_y p_z v_y v_z \\
E &= -c_x p_x + c_y p_y + c_x p_x v_x^2 - c_y p_y v_x^2 - c_z p_z v_x^2 + c_x p_x v_y^2 - c_y p_y v_y^2 + c_z p_z v_y^2 \\
&\quad + c_z p_x v_x v_z + c_x p_z v_x v_z - c_z p_y v_y v_z - c_y p_z v_y v_z
\end{aligned}$$

$$
\begin{aligned}
F &= \frac{1}{2}(c_x^2 - c_y^2 + (-c_x^2 + c_y^2 + c_z^2)v_x^2 + (-c_x^2 + c_y^2 - c_z^2)v_y^2) - c_x c_z v_x v_z + c_y c_z v_y v_z \\
G &= p_x p_y - p_x p_y v_x^2 + p_z^2 v_x v_y - R^2 v_x v_y - p_x p_y v_y^2 - p_y p_z v_x v_z - p_x p_z v_y v_z \\
H &= -c_y p_x - c_x p_y + c_y p_x v_x^2 + c_x p_y v_x^2 - 2c_z p_z v_x v_y + c_y p_x v_y^2 + c_x p_y v_y^2 + c_z p_y v_x v_z \\
&\quad + c_y p_z v_x v_z + c_z p_x v_y v_z + c_x p_z v_y v_z \\
J &= c_x c_y - c_x c_y v_x^2 + c_z^2 v_x v_y - c_x c_y v_y^2 - c_y c_z v_x v_z - c_x c_z v_y v_z
\end{aligned}
$$

We can build up a bound for this function easily. The parabolic terms are simple to bound over any interval — if the peak is in the interval, it is the max (or min) and one of the two endpoint values is the min (or max). Otherwise the two endpoint values are the max and min. The sine functions are similarly easy to bound. If a peak or valley occurs in the interval, it forms an extreme. Otherwise the end values form the extremes. To find the bound on the terms composed of a parabola multiplied by a sine wave, the bound values for the parabola are multiplied by those of the sine wave, giving four results. The largest and smallest of these represent a bound on the overall term. Since any value of the two factors lies between their respective bounds, any product must similarly lie between the products of the bounds.

The upper bound on the entire function is simply the upper bound of the three terms added together, while the lower bound is the sum of the three individual lower bounds.

Bounds for minpar(t), maxpar(t), and umin(t)

We can apply the same technique to bounding minpar(t), maxpar(t), and umin(t). The three functions are similar in form. If umin(t) is rearranged a bit, we have

$$\text{umin}(t) = \frac{A + Bt + (C + Dt)\cos 2\theta t + (E + Ft)\sin 2\theta t - \sqrt{\text{rad}(t)}}{1 - (v_x \cos\theta t + v_y \sin\theta t)} \qquad (4.8)$$

and minpar(t) and maxpar(t) are slightly more complicated:

$$\begin{array}{l} \text{minpar}(t) \\ (\text{maxpar}(t)) \end{array} = (p_x - c_x t)\cos\theta t + (p_y - c_y t)\sin\theta t + \frac{\begin{array}{c}(A + Bt)\cos\theta t + (C + Dt)\sin\theta t + (E + Ft)\cos 3\theta t \\ +(G + Ht)\sin 3\theta t - (+)(v_x\cos\theta t + v_y\sin\theta t)\sqrt{\text{rad}(t)}\end{array}}{1 - (v_x\cos\theta t + v_y\sin\theta t)^2}$$

Constants for umin(t) are

$$\begin{aligned}
A &= -\frac{p_x v_x}{2} - \frac{p_y v_y}{2} - p_z v_z \\
B &= \frac{c_x v_x}{2} + \frac{c_y v_y}{2} + c_z v_z \\
C &= \frac{p_x v_x}{2} - \frac{p_y v_y}{2} \\
D &= -\frac{c_x v_x}{2} + \frac{c_y v_y}{2} \\
E &= \frac{p_y v_x}{2} + \frac{p_x v_y}{2} \\
F &= -\frac{c_y v_x}{2} - \frac{c_x v_y}{2}
\end{aligned}$$

while the constants for minpar(t) and maxpar(t) are

$$\begin{aligned}
A &= -\frac{p_x v_x^2}{4} - \frac{p_y v_x v_y}{2} + \frac{p_x v_y^2}{4} - p_z v_x v_z \\
B &= \frac{c_x v_x^2}{4} + \frac{c_y v_x v_y}{2} - \frac{c_x v_y^2}{4} + c_z v_x v_z \\
C &= \frac{p_y v_x^2}{4} - \frac{p_x v_x v_y}{2} - \frac{p_y v_y^2}{4} - p_z v_y v_z \\
D &= -\frac{c_y v_x^2}{4} + \frac{c_x v_x v_y}{2} + \frac{c_y v_y^2}{4} + c_z v_x v_z \\
E &= \frac{p_x v_x^2}{4} - \frac{p_y v_x v_y}{2} - \frac{p_x v_y^2}{4} \\
F &= -\frac{c_x v_x^2}{4} + \frac{c_y v_x v_y}{2} + \frac{c_x v_y^2}{4} \\
G &= \frac{p_y v_x^2}{4} + \frac{p_x v_x v_y}{2} - \frac{p_y v_y^2}{4} \\
H &= -\frac{c_y v_x^2}{4} - \frac{c_x v_x v_y}{2} + \frac{c_y v_y^2}{4}
\end{aligned}$$

To bound these three functions, we can start by breaking the fractional terms into the numerator and denominator. The numerators all have linear factors multiplied by sine waves. These may be bounded as described above, except that the linear terms always take their bounds from the two end points. The remaining term of the numerators is $\sqrt{\text{rad}(t)}$. All we need to do is take the square root of the upper and lower bounds. Since we are searching in an interval that was the result of finding positive segments of rad(t), the smallest value can be 0, so if the lower bound is negative, we substitute 0 in its place.

The denominator is a simple function. It is a constant minus the square of a sine wave. If the sine wave has a root in the interval, the minimum becomes zero while the maximum is the larger of the two endpoint values. Then we subtract the squared sine wave bounds from 1, giving bounds for the denominator.

Bounds for the complete fraction are now achieved by dividing the numerator's bounds by the denominator's bounds. The largest and smallest become the bounds for the fraction term. Two of the divides can be avoided with appropriate sign and magnitude testing.

Finally, in the case of minpar(t) and maxpar(t), there are additional terms involving a linear factor and a sinusoid factor. These can be bounded as we have already described, and then all the bounds added together to get bounds on the complete functions.

It should be noted that if a discontinuity exists in the interval, due to the line becoming parallel to the cylinder axis, the bounds will blow up the closer to the discontinuity the search procedure gets. The upper value will approach ∞ and the lower bound will approach $-\infty$. However, if a subinterval is created that no longer contains the discontinuity, both upper and lower bounds should become large (or small), eliminating it from the search. Once found, a discontinuity can be differentiated from a root by checking the value.

Bounds for end(t)

Bounding end(t) is simpler than the functions using rad(t). Expanded, end(t) is

$$\text{end}(t) = \frac{(c_x t - p_x)\cos\theta t + (c_y t - p_y)\sin\theta t}{v_x \cos\theta t + v_y \sin\theta t} \tag{4.9}$$

The numerator consists of two terms of a line multiplied by a sine wave. The denominator is just a sine wave. The bounds can be built up as before. Because of the nature of the algorithm, a division by 0 won't occur because the line would then be intersecting the side surface or not intersecting the cylinder at all.

4.2.2 Tolerance and the Numerical Approach

Unless we want to keep subdividing down to 0 interval width, we need to decide when an interval is too narrow. In fact, since this has to run on existing computers with limited precision, we have no choice but to have a finite limit on the interval width. This immediately brings up the fact that we can't actually find every root that the function has. If the function dips below zero and comes back up between two successive floating point numbers, there is no way that this can be detected.

So, we see that the machine floating point precision sets a lower bound on the accuracy obtainable. Numerical inaccuracies also set a limit on obtainable accuracy. Due to roundoff errors, a function often isn't smooth on a small scale. For instance, look at an example of rad(t) (figure 4.2). As we narrow the interval down around the near-zero region, we see that the curve is smooth (figure 4.3). When the interval gets narrow enough, the curve starts showing an unpredictable behavior as the effects of the discrete representation of floating point numbers become apparent from point to point(figure 4.4). This unstable small-scale activity sets a further limit on the

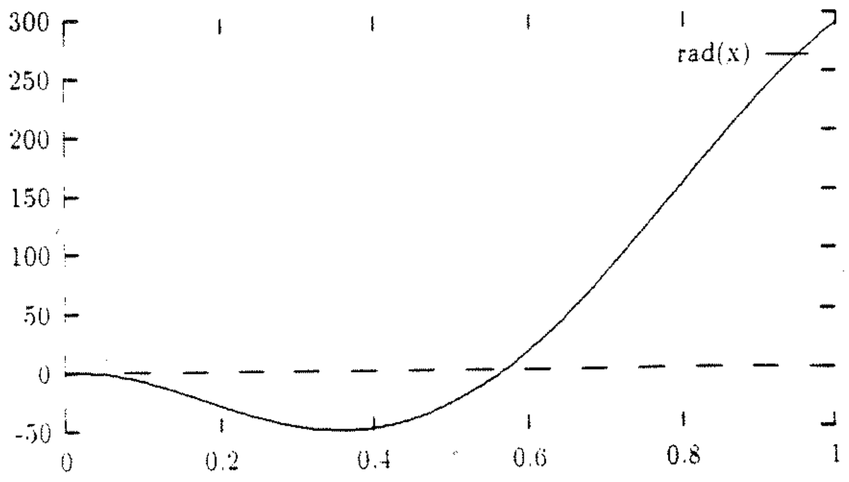


Figure 4.2: rad(t) from 0 to 1

resolution of our answer. Empirically, the width of the uncertainty interval for rad(t) is about 1e-9. This is for IEEE floating point hardware which gives about 1e-16 precision in the range from 0 to 1.

The numerical approach therefore has a tolerance. However, its correlation to a geometrical measure of error is not obvious. A loose sampling in the search of rad(t) could allow small contact regions to be missed. In addition, the ends of the intervals may be placed further in than before. As a result, the later function evaluations don't look at as much as they should. A deep gouge that occurs over a very narrow interval of time could occur in the missed contact region. A similar miss of the extent of cutting could occur at the ends of intervals that are found. The geometrical meaning of tolerance in this algorithm is distance between successive cylinder placements, since each value of t corresponds to a distinct cylinder placement. However, the error relationship is not anywhere near as simple.

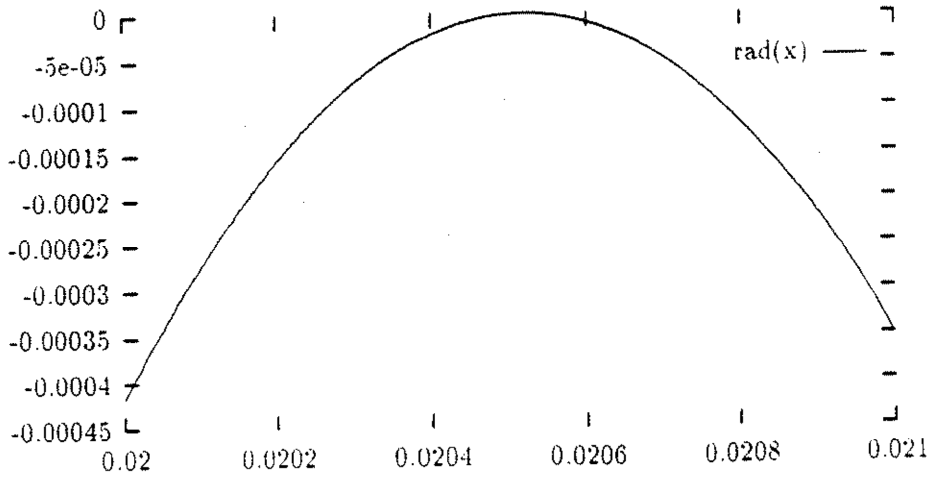


Figure 4.3: rad(t) from 0.02 to 0.021

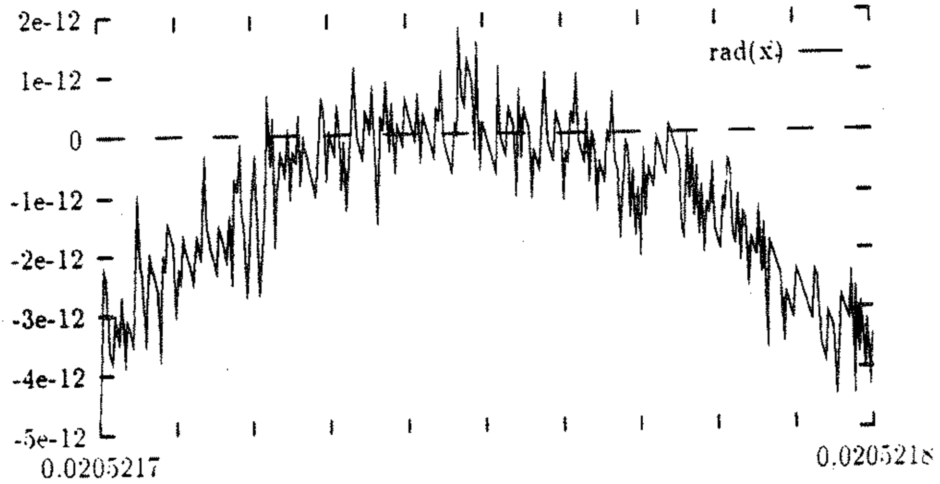


Figure 4.4: rad(t) from 0.0205217 to 0.0205218

For this reason, the numerical approach is not amenable to an adjustable tolerance. Another practical issue that gets in the way of adjustable tolerances is the need for backtracking if they are allowed. If the search of rad(t) misses an interval of negative values, later evaluations of minpar, maxpar and umin may try to take the square root of these negative numbers. This can't be allowed, requiring a reevaluation of rad(t) in this region so that it can be removed from consideration.

The graphs also bring up another important concern with safety of evaluations. In this graph, the function just touches near 0. The static in the values there obscure the number of actual roots present, whether 0, 1, 2, or even more. Due to the chaotic bouncing exhibited by the function, if this interval were to be classified as positive, later functions may try to take the square root of a negative number, with poor results. Therefore, instead of actually finding zeros, what has to be done is to set the effective zero at the positive epsilon value. This implies the necessity of a "safe" direction to root finding.

Chapter 5

Polyhedral Approximation of a 5-Axis Sweep

5.1 Introduction

A common way to represent objects for many graphics purposes is to make a polyhedron that approximates the surface of the object. Polyhedra have some desirable properties that make them a useful representation for graphical applications. A polyhedron consists of points, straight line segments and flat polygon faces. This means that intersections and other operations with polyhedra are simple to compute and therefore, make for fast computations. Polyhedra are easy to display since scan conversion of a flat surface essentially involves intersection of scan lines and polygons. It is also relatively easy to find the intersection of polyhedra with other points, lines, polygons, and polyhedra, since all the equations involved are linear. Polyhedra are also useful as objects for ray tracing since the algorithm consists of repeatedly finding intersections of rays and objects.

The biggest advantage of using polyhedra is also its biggest disadvantage: because lines and surfaces are flat and straight, it usually takes numerous polygons to represent curved surfaces. When a polyhedron is displayed, its contours appear segmented. In addition, the constant normals of the faces do not model the normals of curved surfaces very effectively. In ray tracing especially, this fact affects the shading process significantly and serves to highlight the segmented nature of the object. These effects are very noticeable unless many very small polygons are used, or in the case of shading, a smoothing algorithm is applied.

Since we are only concerned with the accuracy of the point of intersection, these disadvantages are of no concern to us. As long as the faces of the polyhedron are within the specified tolerance of the real swept envelope, we can take advantage of the simplicity of the intersection of lines and polygons. Finding the intersection of a ray and a polyhedron efficiently has been solved in ray-tracing research [15].

How, then, do we construct a polyhedral representation for five-axis tool movements?

5.2 Building a Polyhedron

5.2.1 The Swept Envelope

To build the polyhedron, we first must consider what constitutes the envelope of the sweep. The cylinder can generate the envelope by cutting with either the side or the ends. We can ignore the top end as it will in reality be attached to the rest of the milling machine, and contact should not occur between the cylinder top and the surface being cut. This is a major error and easily identified.

To attack the problem, we will describe how to build the individual pieces. First, we create the side meshes, then the surface swept by the end disc. The two ends are represented by the cylinders themselves at $t = 0$ and $t = 1$. We often only need to use the cylinder at the starting position, because the ending cylinder position of one tool movement is often the starting cylinder position of the next one. Intersecting a cylinder will be much faster than intersecting the many polygons that would result by polygonalizing the cylinder's surface. Finally, the pieces need to be stitched together to form a complete polyhedron. Because we won't build a surface for the sweep of the top end of the cylinder, the result will be an open shell, rather than a closed surface.

5.2.2 Constructing the Side Meshes

Parameterized Equations for the Side Surfaces

To build the polygonization for the sides, we turn to differential geometry, which gives us a means to find the envelope of a one-parameter family of surfaces [10]. To obtain the envelope, we take the implicit equation $f(x, y, z) = 0$ describing an instance of the surface, and then substitute in the equations describing the motion of the object, giving us $f(x, y, z, t) = 0$. Then, the intersection of $f = 0$ and the derivative with respect to the motion parameter t, $f'(x, y, z, t) = 0$, gives us a representation of the swept envelope created by the surface. By eliminating t from the two equations, we get an implicit description of the swept envelope as the result.

The effect of this computation, essentially, is to give a general formula for the intersection of the surface at time t and the surface at $t + \epsilon$ as ϵ goes to 0. We can see this by looking at a simple example. In two dimensions, a circle slid along a line will have two lines parallel to the line of motion as an envelope. The intersection points of two of the overlapping circles lie on a line perpendicular to the envelope lines. The closer these two circles are, the closer the points move towards the envelope lines. In the limit, they lie on the envelope lines. The points of contact of the surface and the envelope are referred to collectively as the critical curve.

For a cylinder, $f(x, y, z) = 0$ is

$$0 = (x - X)^2 + (y - Y)^2 + (z - Z)^2 - R^2 - ((x - X)A + (y - Y)B + (z - Z)C)^2$$

where (X, Y, Z) is the base point of the cylinder, and (A, B, C) is the axis vector. The base point and axis of the cylinder undergo the transformations in equations 2.3 and 2.4:

$$C(t) \quad = \quad (c_x t, c_y t, c_z t)$$

$$A(t) = (\cos\theta t, \sin\theta t, 0).$$

Now, the equation for the family of cylinders as a function of t, $f(x, y, z, t) = 0$, is

$$0 = (x - c_x t)^2 + (y - c_y t)^2 + (z - c_z t)^2 - R^2 - ((x - c_x t)\cos\theta t + (y - c_y t)\sin\theta t)^2. \quad (5.1)$$

The derivative $f'(x, y, z, t) = 0$ is

$$\begin{aligned} 0 &= 2(c_x(c_x t - x) + c_y(c_y t - y) + c_z(c_z t - z) - (\theta(y - c_y t)\cos\theta t + \\ &\quad \theta(c_x t - x)\sin\theta t - c_y \sin\theta t - c_x \cos\theta t)((x - c_x t)\cos\theta t + (y - c_y t)\sin\theta t)). \quad (5.2) \end{aligned}$$

It is very likely impossible to eliminate t from these two equations and get an implicit function for the surface.

However, we can turn these equations into a parametric surface representation $f(t, l)$. With a parametric function of the surface, we can generate points by giving t and l values and knit them together into a mesh. To build the parametric function, take 5.1 and 5.2 at $t = 0$:

$$0 = y^2 + z^2 - R^2 \quad (5.3)$$

$$0 = y(c_y + x\theta) + zc_z \quad (5.4)$$

Since the cylinder lies on the x axis starting at the origin here, we can replace x by a parameter l corresponding to the position along the length of the cylinder. We now have

$$0 = y^2 + z^2 - R^2 \quad (5.5)$$

$$0 = yc_y + ly\theta + zc_z. \quad (5.6)$$

Now we can solve the first equation for either y or z, substitute into the second equation and arrive at a parameterization of both y and z. Solving for z and plugging in to get y gives

$$\begin{aligned} y(c_y + l\theta) &= -c_z\sqrt{R^2 - y^2} \\ y^2(c_y + l\theta)^2 &= c_z^2(R^2 - y^2) \\ y^2 &= R^2\frac{c_z^2}{c_z^2 + (c_y + l\theta)^2} \\ y &= \pm R\frac{c_z}{(c_z^2 + (c_y + l\theta)^2)^{1/2}}. \end{aligned} \tag{5.7}$$

and solving for y and plugging in to get z gives

$$\begin{aligned} zc_z &= -\sqrt{R^2 - z^2}(c_y + l\theta) \\ z^2c_z^2 &= (R^2 - z^2)(c_y + l\theta)^2 \\ z^2(c_z^2 + (c_y + l\theta)^2) &= R^2(c_y + l\theta)^2 \\ z &= \pm R\frac{c_y + l\theta}{(c_z^2 + (c_y + l\theta)^2)^{1/2}}. \end{aligned} \tag{5.8}$$

The signs of y and z allow for four possibilities. Only two of these are actually valid solutions. Using equations 5.6, 5.7, and 5.8, if y and z have the same sign, we have

$$\begin{aligned} \pm R\frac{c_z}{(c_z^2 + (c_y + l\theta)^2)^{1/2}}(c_y + l\theta) &= \mp R\frac{c_y + l\theta}{(c_z^2 + (c_y + l\theta)^2)^{1/2}}c_z \\ \pm 1 &= \mp 1 \end{aligned}$$

which is impossible. When y and z have opposite signs, we get

$$\begin{aligned} \pm R\frac{c_z}{(c_z^2 + (c_y + l\theta)^2)^{1/2}}(c_y + l\theta) &= \pm R\frac{c_y + l\theta}{(c_z^2 + (c_y + l\theta)^2)^{1/2}}c_z \\ \pm 1 &= \pm 1 \end{aligned}$$

which is obviously true. Therefore, we only have a solution when y and z are of opposite signs.

From all this we get a complete set of parametric equations in l for the side surfaces at $t = 0$:

$$x(l) = l \tag{5.9}$$

$$y(l) = \pm R\frac{c_z}{(c_z^2 + (c_y + l\theta)^2)^{1/2}} \tag{5.10}$$
$$z(l) = \mp R\frac{c_y + l\theta}{(c_z^2 + (c_y + l\theta)^2)^{1/2}}. \tag{5.11}$$

This is a snapshot of the critical curve on the body of the cylinder at one time. We need to make these equations functions of t to describe the critical curve on the body of the cylinder at any given time. c_x, c_y, c_z, and θ change with time. (c_x, c_y, c_z) undergoes the reverse transformation, while θ decreases linearly. So,

$$\begin{aligned}
c_x(t) &= c_x(1-t)\cos\theta t + c_y(1-t)\sin\theta t \\
c_y(t) &= -c_x(1-t)\sin\theta t + c_y(1-t)\cos\theta t \\
c_z(t) &= c_z(1-t) \\
\theta(t) &= (1-t)\theta
\end{aligned}$$

and substituting this into our parametric equations results in

$$\begin{aligned}
x(t,l) &= l \\
y(t,l) &= \pm R\frac{c_z(1-t)}{(c_z^2(1-t)^2 + (l\theta(1-t) + c_y(1-t)\cos\theta t - c_x(1-t)\sin\theta t)^2)^{1/2}} \\
z(t,l) &= \mp R\frac{l\theta(1-t) + c_y(1-t)\cos\theta t - c_x(1-t)\sin\theta t}{(c_z^2(1-t)^2 + (l\theta(1-t) + c_y(1-t)\cos\theta t - c_x(1-t)\sin\theta t)^2)^{1/2}}
\end{aligned}$$

$$\begin{aligned}
x(t,l) &= l \\
y(t,l) &= \pm R\frac{c_z(1-t)}{(1-t)(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}} \\
z(t,l) &= \mp R\frac{(1-t)(l\theta + c_y\cos\theta t - c_x\sin\theta t)}{(1-t)(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}}
\end{aligned}$$

$$x(t,l) = l \tag{5.12}$$
$$y(t,l) = \pm R\frac{c_z}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}} \tag{5.13}$$
$$z(t,l) = \mp R\frac{l\theta + c_y\cos\theta t - c_x\sin\theta t}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}}. \tag{5.14}$$

We now have a description of the critical curve on the cylinder body in time. To obtain this curve in the fixed line's frame of reference, we transform the points of the critical curve using the transformation 2.5. To do this, the equations themselves are transformed, giving

$$\begin{aligned} x(t,l) &= c_x t + l\cos\theta t \mp R\frac{c_z \sin\theta t}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}} && (5.15)\\ y(t,l) &= c_y t + l\sin\theta t \pm R\frac{c_z \cos\theta t}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}} && (5.16)\\ z(t,l) &= c_z t \mp R\frac{l\theta + c_y\cos\theta t - c_x\sin\theta t}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}}. && (5.17) \end{aligned}$$

With these equations, we have a parameterization of the side surfaces of the swept envelope of a five-axis tool movement.

Another way to look at the functions geometrically is to notice that the first two terms of $x(t,l)$ and $y(t,l)$ and the first term of $z(t,l)$ give points along the axis of the cylinder at any time t. The remaining term in each function is the component of the vector that, added to the center axis point, is on the swept surface. Thus every cylinder contributes to the swept surface, and any point on the surface has a corresponding point on the other surface directly opposite through the cylinder axis. The line connecting them is normal to both surfaces.

It is interesting to note that the surfaces swept out by the cylinder side are not ruled. In other words, while it may seem that the cylinder at any time meets the surface in a line, this is not the case. We will show this in more detail in Section 5.3.1.

Assembling the Side Surface Mesh

Now that we have a set of parametric equations for the side surfaces, we can go about building a mesh for the side surfaces. Selecting either the first or second set of equations (+ or − of a ± pair), we pick a grid of pairs (t,l) with t ranging between 0 and 1 and l going from 0 to L. The equations are evaluated for each pair, returning a point in space on the side surface.

We then pick a triangulation in parameter space of the set of pairs (t, l). The corresponding points are then connected into triangles in real space, which gives us a triangulation of the side surface with all the points lying on the surface itself.

5.2.3 Building the Surface Swept by the Bottom Disc

To build the polygonal mesh for the bottom disc, we will use a similar approach as above. Parametric equations will be used to generate envelope points from two-dimensional coordinates that will describe the whole surface. The difference here is that the envelope has to be built in parts.

The disc, unlike the cylinder body, has an edge to worry about. Both the edge and the interior of the disc may sweep parts of the envelope. The interior, being planar, will sweep a developable surface, as we will show. This simply means that the surface is ruled (a line sweeps out the surface), and that it may be laid out flat on a plane. The developable surface may not actually be part of the envelope if its critical curve falls outside the circular boundary of the disc throughout the sweep.

The Developable Surface Swept by the Disc

The bottom disc is just a circular piece of the plane that is perpendicular to the cylinder axis and contains the axis point at $l = 0$. The family of surfaces is just the continuous set of planes generated as the cylinder sweeps along. Recall from Section 5.2.2 that the critical curve of a member of the family of surfaces is the intersection of the surface at time t with the surface at time $t + \epsilon$. In the case of planes, the intersection will be a line. In our motion, the plane always contains a line parallel to the z axis, so the intersection necessarily is parallel to the z axis as well. These two facts tell us that the plane of the bottom disc will sweep out a developable surface.

The normal to the plane as a function of time is just the cylinder axis vector $(\cos\theta t, \sin\theta t, 0)$. The plane always contains the center point of the base $(c_x t, c_y t, c_z t)$. We now can get the equation for the plane as a function of time:

$$x\cos\theta t + y\sin\theta t = c_x t\cos\theta t + c_y t\sin\theta t.$$

If we now take the derivative with respect to t and divide by θ, we have

$$-x\sin\theta t + y\cos\theta t = (\frac{c_x}{\theta} + c_y t)\cos\theta t + (\frac{c_y}{\theta} - c_x t)\sin\theta t.$$

Each equation describes a plane parallel to the z axis.

Solving the first equation for y and then x gives

$$\begin{aligned} y &= c_x t\frac{\cos\theta t}{\sin\theta t} + c_y t - x\frac{\cos\theta t}{\sin\theta t} \\ x &= c_x t + c_y t\frac{\sin\theta t}{\cos\theta t} - y\frac{\sin\theta t}{\cos\theta t}. \end{aligned}$$

These are plugged into the second equation to get $x(t)$ and $y(t)$ respectively:

$$x(t) = c_x t - \sin\theta t(\frac{c_x}{\theta}\cos\theta t + \frac{c_y}{\theta}\sin\theta t) \quad (5.18)$$

$$y(t) = c_y t + \cos\theta t(\frac{c_x}{\theta}\cos\theta t + \frac{c_y}{\theta}\sin\theta t). \quad (5.19)$$

This line, then, is the critical curve at time t for the plane of the cylinder base.

However, the bottom of the cylinder is actually a disc and not just an infinite plane. The line may or may not be within the disc's boundaries at a given time. If we find the intersection of this line with the cylinder, we will get the end points of the line segment in the disc itself. Plugging $x(t)$ and $y(t)$ into equation 5.1 and solving for z gives

$$z(t) = c_z t \pm \sqrt{R^2 - \frac{1}{\theta^2}\begin{pmatrix} c_x^2(\cos^4\theta t + \cos^2\theta t\sin^2\theta t) + c_y^2(\sin^4\theta t + \cos^2\theta t\sin^2\theta t) \\ +2c_x c_y(\cos^3\theta t\sin\theta t + \cos\theta t\sin^3\theta t) \end{pmatrix}}$$

$$= c_z t \pm \sqrt{R^2 - \frac{1}{\theta^2}\left(c_x^2 \cos^2 \theta t + c_y^2 \sin^2 \theta t + 2c_x c_y \cos \theta t \sin \theta t\right)}$$
$$= c_z t \pm \sqrt{R^2 - \frac{1}{\theta^2}\left(c_x \cos \theta t + c_y \sin \theta t\right)^2}.$$

From $z(t)$ we can derive the time intervals over which the critical line lies within the circle boundary. Since the line segment end points only exist when the expression under the square root is positive, we need

$$\begin{aligned} R &\geq \frac{1}{\theta}(c_x \cos \theta t + c_y \sin \theta t) \\ &\geq \frac{1}{\theta}\sqrt{c_x^2 + c_y^2}\cos(\theta t - \arctan \frac{c_y}{c_x}). \end{aligned} \tag{5.20}$$

Solving for t when 5.20 is exactly equal gives the interval end points

$$t = \pm\frac{1}{\theta} \arccos\left(\frac{R\theta}{\sqrt{c_x^2 + c_y^2}}\right) + \frac{1}{\theta}\arctan\frac{c_y}{c_x} + k\frac{\pi}{\theta}$$

where k is an integer chosen so that $0 \leq t \leq 1$ if possible. If it is not possible, then there is no developable component to the swept surface for the motion.

With these equations we have everything necessary to generate a developable surface mesh. Note that we essentially have a two parameter description of the surface, but the second parameter is just z. To build the surface, pick sample times over the interval in which the critical line lies in the circle boundary. The line segments are then joined at the ends to make connected quadrilaterals, giving a completed mesh.

Note that, in general, the points on a cylinder where the developable critical curve line meets the disc edge are not the same points that the side critical curves meet the disc edge. The developable surface line oscillates in the plane of the disc in time in a direction perpendicular to the line. The points due to the side curves lie on either side of the developable surface line when it is in the center of the disc. They oscillate back and forth in an arc with time (see figure 5.1).

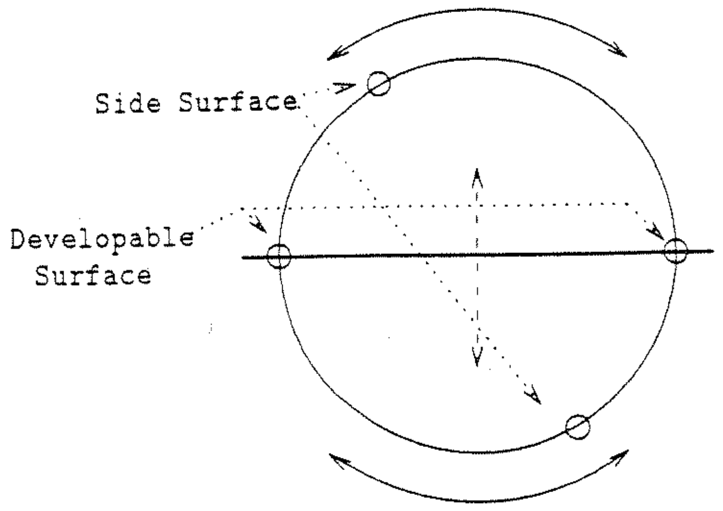


Figure 5.1: Movement of side and developable points on disc edge

The Surface Swept by the Disc Edge

The contribution to the swept envelope by the disc's edge is more complex than that of the developable surface. It is possible to determine the actual arcs that contact the envelope at any given time, but there can be more than one arc, making the mesh construction more difficult. Instead, we will generate a mesh for the whole disc. Any polygons that don't represent the surface will be internal to the sweep and simply slow down the intersection routines somewhat. If a good space subdivision algorithm is used to segment the polyhedron, the extra internal polygons should have almost no effect because the are unlikely to be in the boxes containing surface polygons. Therefore, intersections will never be performed with them.

To build the parametric description of the sweep of the disc, we need to introduce a parameter α to describe position around the disc's edge. We'll define α such that at $t = 0$, $\alpha = 0$ corresponds to $(0, 0, R)$ and that $(0, \sin\alpha, \cos\alpha)$ defines a point on the disc edge. Then, plugging this position formula into the motion equation 2.5, we get parametric representations for the sweep of the disc:

$$x(t,\alpha) = c_x t - R\sin\alpha\sin\theta t \tag{5.21}$$

$$y(t,\alpha) = c_y t + R\sin\alpha\cos\theta t \tag{5.22}$$

$$z(t,\alpha) = c_z t + R\cos\alpha. \tag{5.23}$$

For $0 \leq t \leq 1$ and $0 \leq \alpha \leq \pi$, all points of the swept surface will be generated.

5.2.4 Merging the Side and Bottom Meshes

We now have a way to build a mesh for each of the component surfaces of the tool movement envelope we want. All we need to do now is join them together smoothly and with no cracks. Cracks potentially would allow a fixed line to go right through the middle of the sweep and not produce an intersection.

The strategy to eliminate cracks in the completed polyhedron is to make sure that where two surfaces meet, they share the same boundary. This means that all the edges of the boundary of one surface should be the same as the edges of the other surface over the length of the border between them.

The bottom edge of each side surface has $l = 0$, meaning all points on these edges are at $(t, 0)$. Each point has a corresponding value (t, α), where α is the angular location around the body of the cylinder. This value also matches values of α on the disc surface. Therefore, each point on the bottom edges of the side surfaces should lie on the disc surface. However, each surface is created independently, so we have to force the disc surface to contain the same set of points as the bottom edge of each side surface.

To do this, we simply add each point on the side surface edge to the disc surface mesh. When a point is added in a disc surface triangle, the triangle is subdivided using the new point. The edge from the side surface point to the previous one becomes part of the disc mesh. If two successive side surface edge points lie in different triangles, the side surface edge must be split. A new point is added at the intersection between the side surface edge and the disc surface triangle edge. The new point is added to the side surface, splitting the side surface triangle it lies in. Then, the new point can be used to continue the process of laying in the side surface edge. This process is repeated until the complete set of side surface edges lies in the disc surface.

Once all of the points from the bottom of the sides are included in the disc mesh, the two surfaces will meet up at a smooth border since they will share connecting edges and vertices. Note that it may be possible to have more than two triangles sharing an edge on the border. If an octree type of approach to limiting intersections to exterior triangles isn't satisfactory, a data structure can be built to allow trimming these internal triangles later.

Joining the developable surface to the disc surface mesh can be done in a similar manner. If we first include the time t when the developable surface starts and ends, the two meshes will join up correctly at the ends of the developable surface. The rest of the developable surface consists of quadrilaterals with two opposing edges in the interior and the other two along the border of the surface. Triangles can be formed if desired. The ends of the developable line segments have corresponding α values. If we include these points in the disc surface mesh as described for the side surface edges, the developable and disc surfaces will then share vertices and edges, resulting in a continuous shared border with no gaps.

Edges due to the side meshes and the developable surface may cross in the disc surface mesh. To solve this problem, links must be maintained from the developable surface to the side surface in those triangles that share an edge. Then, if a developable edge crosses a triangle edge shared by the side surface, both meshes can be appropriately updated.

Once all this has been done, we have a closed polyhedron with extra triangles inside.

5.3 Bounding Polyhedron Tolerance

5.3.1 Bounding the Side Surface Meshes

Now that we have a way to choose points on the side surfaces, we have to decide how many are needed and where. To guarantee our approximation, we need to show that all points on the surface are within tolerance T of the approximation. In addition, all points on the approximation must be within T of the true surface, otherwise surfaces not even remotely near the surface could be included and considered valid.

For a given tolerance T and a point on the surface $(x(t_0, l_0), y(t_0, l_0), z(t_0, l_0))$, we can find a polygon tangent to the surface at that point which is within T of the true surface. By using a continuous mesh of these polygons, we guarantee that every point on the surface will be within T of the approximation. Since these are the only polygons that will be used to approximate the surface, the approximation will always be within T of the true surface as well. Starting from the parametric equations for the sides, we find the standard linear approximation [32, page 925] of each coordinate. These are:

$$\begin{aligned} x(t,l) &= x(t_0, l_0) + \frac{\partial x}{\partial t}(t_0, l_0)(t - t_0) + \frac{\partial x}{\partial l}(t_0, l_0)(l - l_0), \\ y(t,l) &= y(t_0, l_0) + \frac{\partial y}{\partial t}(t_0, l_0)(t - t_0) + \frac{\partial y}{\partial l}(t_0, l_0)(l - l_0), \\ z(t,l) &= z(t_0, l_0) + \frac{\partial z}{\partial t}(t_0, l_0)(t - t_0) + \frac{\partial z}{\partial l}(t_0, l_0)(l - l_0). \end{aligned}$$

This is just the parametric description of the tangent plane at (t_0, l_0). For small changes in l and t, this will give an acceptable approximation.

How acceptable is this? Using standard calculus, we see that the error in the standard linear approximation can be bounded in a region around (t_0, l_0) [32, page 927]. The error is

$$|E_x(t,l)| \leq \frac{1}{2}M(|t-t_0| + |l-l_0|)^2,$$

where M is an upper bound on $|\frac{\partial^2 x}{\partial t^2}|$, $|\frac{\partial^2 x}{\partial t \partial l}|$, and $|\frac{\partial^2 x}{\partial l^2}|$ over a range of values around (t_0, l_0). This tells how much at most the x coordinate is off by in this region, and we can similarly find it for y and z. Then, the maximum error for (t,l) is no more than $\sqrt{E_x^2(t,l) + E_y^2(t,l) + E_z^2(t,l)}$. At this point we have to select a region of t and l that keep the error less that T.

Using equations 5.15 with a +, 5.16 with a −, and 5.17 with a +, the first partial derivatives for each component are:

$$\begin{aligned}
\frac{\partial x}{\partial t} &= c_x - l\theta \sin\theta t + \frac{Rc_z\theta \sin\theta t(c_x\cos\theta t + c_y \sin\theta t)(l\theta + c_y\cos\theta t - c_x \sin\theta t)}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{3/2}} \\
&\quad + \frac{Rc_z\theta\cos\theta t}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}} \\
\frac{\partial y}{\partial t} &= c_y + l\theta\cos\theta t - \frac{Rc_z\theta\cos\theta t(c_x\cos\theta t + c_y\sin\theta t)(l\theta + c_y\cos\theta t - c_x\sin\theta t)}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{3/2}} \\
&\quad + \frac{Rc_z\theta\sin\theta t}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}} \\
\frac{\partial z}{\partial t} &= c_z + \frac{R\theta(c_x\cos\theta t + c_y\sin\theta t)(l\theta + c_y\cos\theta t - c_x\sin\theta t)^2}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{3/2}} \\
&\quad - \frac{R\theta(c_x\cos\theta t + c_y\sin\theta t)}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}} \\
\frac{\partial x}{\partial l} &= \cos\theta t - \frac{Rc_z\theta\sin\theta t(l\theta + c_y\cos\theta t - c_x\sin\theta t)}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{3/2}} \\
\frac{\partial y}{\partial l} &= \sin\theta t + \frac{Rc_z\theta\cos\theta t(l\theta + c_y\cos\theta t - c_x\sin\theta t)}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{3/2}} \\
\frac{\partial z}{\partial l} &= -\frac{R\theta(l\theta + c_y\cos\theta t - c_x\sin\theta t)^2}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{3/2}} \\
&\quad + \frac{R\theta}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}}.
\end{aligned}$$

Taking partial derivatives again gives us:

$$\frac{\partial^2 x}{\partial t^2} = 3\frac{Rc_z\theta^2\sin\theta t(c_x\cos\theta t + c_y\sin\theta t)^2(l\theta + c_y\cos\theta t - c_x\sin\theta t)^2}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{5/2}}$$

$$
\begin{aligned}
& +2\frac{Rc_z\theta^2\cos\theta t(c_x\cos\theta t+c_y\sin\theta t)(l\theta+c_y\cos\theta t-c_x\sin\theta t)}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}} \\
& +\frac{Rc_z\theta^2\sin\theta t(c_y\cos\theta t-c_x\sin\theta t)(l\theta+c_y\cos\theta t-c_x\sin\theta t)}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}} \\
& -\frac{Rc_z\theta^2\sin\theta t(c_x\cos\theta t+c_y\sin\theta t)^2}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}} \\
& -\frac{Rc_z\theta^2\sin\theta t}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{1/2}}-l\theta^2\cos\theta t
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial^2 x}{\partial l\partial t} &= -3\frac{Rc_z\theta^2\sin\theta t(c_x\cos\theta t+c_y\sin\theta t)(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{5/2}} \\
& -\frac{Rc_z\theta^2\cos\theta t(l\theta+c_y\cos\theta t-c_x\sin\theta t)}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}} \\
& +\frac{Rc_z\theta^2\sin\theta t(c_x\cos\theta t+c_y\sin\theta t)}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}}-\theta\sin\theta t \\
\frac{\partial^2 x}{\partial l^2} &= 3\frac{Rc_z\theta^2\sin\theta t(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{5/2}} \\
& -\frac{Rc_z\theta^2\sin\theta t}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}} \\
\frac{\partial^2 y}{\partial t^2} &= -3\frac{Rc_z\theta^2\cos\theta t(c_x\cos\theta t+c_y\sin\theta t)^2(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{5/2}} \\
& +2\frac{Rc_z\theta^2\sin\theta t(c_x\cos\theta t+c_y\sin\theta t)(l\theta+c_y\cos\theta t-c_x\sin\theta t)}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}} \\
& -\frac{Rc_z\theta^2\cos\theta t(c_y\cos\theta t-c_x\sin\theta t)(l\theta+c_y\cos\theta t-c_x\sin\theta t)}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}} \\
& +\frac{Rc_z\theta^2\cos\theta t(c_x\cos\theta t+c_y\sin\theta t)^2}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}} \\
& +\frac{Rc_z\theta^2\cos\theta t}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{1/2}}-l\theta^2\sin\theta t \\
\frac{\partial^2 y}{\partial l\partial t} &= \frac{3Rc_z\theta^2\cos\theta t(c_x\cos\theta t+c_y\sin\theta t)(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{5/2}} \\
& -\frac{Rc_z\theta^2\sin\theta t(l\theta+c_y\cos\theta t-c_x\sin\theta t)}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}} \\
& -\frac{Rc_z\theta^2\cos\theta t(c_x\cos\theta t+c_y\sin\theta t)}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}}+\theta\cos\theta t \\
\frac{\partial^2 y}{\partial l^2} &= -3\frac{Rc_z\theta^2\cos\theta t(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{5/2}}
\end{aligned}
$$

$$
\begin{aligned}
& + \frac{Rc_z\theta^2\cos\theta t}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}} \\
\frac{\partial^2 z}{\partial t^2} &= 3\frac{R\theta^2(c_x\cos\theta t+c_y\sin\theta t)^2(l\theta+c_y\cos\theta t-c_x\sin\theta t)^3}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{5/2}} \\
&\quad - 3\frac{R\theta^2(c_x\cos\theta t+c_y\sin\theta t)^2(l\theta+c_y\cos\theta t-c_x\sin\theta t)}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}} \\
&\quad + \frac{R\theta^2(c_y\cos\theta t-c_x\sin\theta t)(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}} \\
&\quad - \frac{R\theta^2(c_y\cos\theta t-c_x\sin\theta t)}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{1/2}} \\
\frac{\partial^2 z}{\partial l\partial t} &= -3\frac{R\theta^2(c_x\cos\theta t+c_y\sin\theta t)(l\theta+c_y\cos\theta t-c_x\sin\theta t)^3}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{5/2}} \\
&\quad + 3\frac{R\theta^2(l\theta+c_y\cos\theta t-c_x\sin\theta t)(c_x\cos\theta t+c_y\sin\theta t)}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}} \\
\frac{\partial^2 z}{\partial l^2} &= 3\frac{R\theta^2(l\theta+c_y\cos\theta t-c_x\sin\theta t)^3}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{5/2}} \\
&\quad - 3\frac{R\theta^2(l\theta+c_y\cos\theta t-c_x\sin\theta t)}{(c_z^2+(l\theta+c_y\cos\theta t-c_x\sin\theta t)^2)^{3/2}}
\end{aligned}
$$

As we can see, these equations are rather complex. But there are some facts that will help us obtain bounds. First, most of the factors are either $c_y\cos\theta t - c_x\sin\theta t$ or $c_x\cos\theta t + c_y\sin\theta t$, which can be rewritten as $\sqrt{c_x^2+c_y^2}\cos(\theta t + \arctan\frac{c_x}{c_y})$ and $\sqrt{c_x^2+c_y^2}\cos(\theta t - \arctan\frac{c_y}{c_x})$ respectively. The only other factors that depend on t are $\sin\theta t$ and $\cos\theta t$. Ignoring exponents for the moment, all factors have a linear dependence on l, if any. Therefore, by choosing starting intervals of t so that the trigonometric factors are monotonic over the interval, any factor of a term will be monotonic over regions smaller than or equal to the starting ones in the t direction. This means that we will only have to consider the values at the ends of an interval of l and of t to determine bounds.

Since we need a bound on the absolute value of the second derivatives, we'll need to keep both the largest and smallest value each term can have over a region. When factors are multiplied, the largest value of the product will be the product of the largest value of each factor. The one exception is if a factor taken to an even power equals zero at some point in a region of t and l. In this case, the minimum value is zero and the maximum value of the factor is the larger of the absolute value of the two ends. Division is similar; the smallest value of the quotient is produced by dividing the largest denominator value into the smallest numerator value, and the largest quotient is the smallest denominator dividing the largest numerator. None of the denominators can be negative[1], so this method holds when the numerator is negative as well. Once all the maximum and minimum values for the terms have been computed, they can be added and subtracted to get a maximum and minimum for the entire function. Then the bound is just the larger of the absolute value of the maximum and minimum.

This will get us a bound, but we can take some steps to improve it. θ can be arbitrarily limited by choice, since the motion can be broken up into smaller segments, as seen in Chapter 3. Therefore we can arbitrarily guarantee that $\theta \leq 1$. This in turn tells us that

$$\begin{gathered} 0 \leq \cos\theta t \leq 1, \quad \text{and} \\ 0 \leq \sin\theta t \leq 1. \end{gathered}$$

In addition, since the denominator of the complicated terms of the functions is essentially a normalization factor, we have

$$\begin{aligned}
\left|\frac{c_z \sin\theta t}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}}\right| &\leq 1, \\
\left|\frac{c_z \cos\theta t}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}}\right| &\leq 1, \\
\left|\frac{(l\theta + c_y\cos\theta t - c_x\sin\theta t)}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}}\right| &\leq 1.
\end{aligned}$$

[1]The denominators *can* become 0, however. This is the source of problems later on.

These inequalities can be used to limit the magnitude of these factors. Unfortunately this technique cannot be applied to all factors.

In general, the bounds presented will work well. However, under the right conditions, the denominators can get very small. The term in the denominator $(l\theta + c_y \cos\theta t - c_x \sin\theta t)$ can easily become zero. In this case, c_z is the only value underneath. When $c_z = 0$, the whole denominator is zero and the equations blow up. Fortunately, when this happens, the plane of rotation contains the line of linear movement, meaning the sides of the swept volume are planes and could be easily handled.

Otherwise $c_z^2 > 0$. If the linear motion lies near the xy plane, c_z will be small, causing the bound to become immense. This occurs because $(l\theta + c_y \cos\theta t - c_x \sin\theta t)$ approaches zero. To see why this happens, we can look at the critical curve in the cylinder's frame of reference (eqs. 5.12, 5.13, 5.14):

$$\begin{aligned} x(t,l) &= l \\ y(t,l) &= \pm R\frac{c_z}{(c_z^2 + (l\theta + c_y \cos\theta t - c_x \sin\theta t)^2)^{1/2}} \\ z(t,l) &= \mp R\frac{l\theta + c_y \cos\theta t - c_x \sin\theta t}{(c_z^2 + (l\theta + c_y \cos\theta t - c_x \sin\theta t)^2)^{1/2}}. \end{aligned}$$

This doesn't help answer the question very well. It is obvious that $y^2 + z^2 = R^2$ and that the two curves lie on opposite sides of the cylinder at any particular l. To get a better idea of what the curve looks like, we can parameterize it in terms of the angle around the cylinder. We start by assuming that $\alpha = 0$ corresponds to $(x, 0, R)$ and positive rotation is toward the y axis, implying that $\alpha = \pi/2$ is $(x, R, 0)$. Then,

$$\begin{aligned} y(t,\alpha) &= R\sin\alpha \\ z(t,\alpha) &= R\cos\alpha. \end{aligned}$$

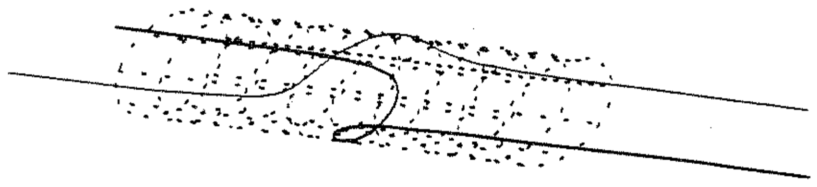


Figure 5.2: cotangent wrapping around cylinder

To compute l in terms of α, we set $y(t,l) = y(t,\alpha)$ and the same for z:

$$\begin{aligned} R\sin\alpha &= R\frac{c_z}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}} \\ R\cos\alpha &= -R\frac{l\theta + c_y\cos\theta t - c_x\sin\theta t}{(c_z^2 + (l\theta + c_y\cos\theta t - c_x\sin\theta t)^2)^{1/2}}. \end{aligned}$$

Combining the two leads to

$$\begin{aligned} \frac{c_z}{\sin\alpha} &= -\frac{l\theta + c_y\cos\theta t - c_x\sin\theta t}{\cos\alpha} \\ \frac{c_z\cos\alpha}{\sin\alpha} &= -(l\theta + c_y\cos\theta t - c_x\sin\theta t) \\ x = l &= -\frac{c_z\cos\alpha}{\theta\sin\alpha} + \frac{c_x\sin\theta t - c_y\cos\theta t}{\theta}. \end{aligned}$$

This form makes clearer what is happening. The shape of the curve at any time t is just $-\frac{c_z}{\theta}\cot\alpha$ wrapped around half the cylinder body. A matching curve is wrapped around the opposite side of the cylinder. As c_z decreases, the center of the cotangent curve flattens out, and the edges rise and fall more sharply over a smaller range of α (see figure 5.2). So, when c_z is near zero, the critical curve descends from infinity in almost a straight line over a small interval of α, then over a very short span of l the curve whips around the cylinder body to the other side, and finally heads out to negative infinity over the small interval of α at the other end. The term depending on t slides the curve up and down the cylinder body in a sinusoid manner.

The l parameterization does a good job of describing the curve from the two infinite ends to the band in which the curve crosses over the cylinder body. The α parameterization describes the crossover portion of the curve well. By using both forms when c_z is small, we will be able to get a better description of the surface and consequently should get better error bounds. Note that when $c_z = 0$, the l equations define two lines down the side of the cylinder body, while the α form describes a circle around the cylinder body. The curve has degenerated so that the turn from line to semicircle is a right angle.

The derivative equations for the α form are much simpler than the l counterparts. When the α equations are transformed into the fixed line frame of reference, we have

$$\begin{aligned}
x(t,\alpha) &= c_x t - \frac{c_z \cos\alpha \cos\theta t}{\theta \sin\alpha} + \cos\theta t \frac{c_x \sin\theta t - c_y \cos\theta t}{\theta} - R\sin\alpha \sin\theta t \\
y(t,\alpha) &= c_y t - \frac{c_z \cos\alpha \sin\theta t}{\theta \sin\alpha} + \sin\theta t \frac{c_x \sin\theta t - c_y \cos\theta t}{\theta} + R\sin\alpha \cos\theta t \\
z(t,\alpha) &= c_z t + R\cos\alpha.
\end{aligned}$$

The first derivatives are

$$\begin{aligned}
\frac{\partial x}{\partial t} &= 2c_x \cos^2\theta t + 2c_y \cos\theta t \sin\theta t + c_z \frac{\cos\alpha}{\sin\alpha} \sin\theta t - R\theta \sin\alpha \cos\theta t \\
\frac{\partial y}{\partial t} &= 2c_y \sin^2\theta t + 2c_x \sin\theta t \cos\theta t - c_z \frac{\cos\alpha}{\sin\alpha} \cos\theta t - R\theta \sin\alpha \sin\theta t \\
\frac{\partial z}{\partial t} &= c_z
\end{aligned}$$

$$\begin{aligned}
\frac{\partial x}{\partial \alpha} &= \frac{c_z \cos\theta t}{\theta \sin^2\alpha} - R\cos\alpha \sin\theta t \\
\frac{\partial y}{\partial \alpha} &= \frac{c_z \sin\theta t}{\theta \sin^2\alpha} + R\cos\alpha \cos\theta t
\end{aligned}$$

$$\frac{\partial z}{\partial \alpha} = -R\sin\alpha$$

giving second derivatives of

$$\begin{aligned}
\frac{\partial^2 x}{\partial t^2} &= -4c_x\theta\cos\theta t\sin\theta t + 2c_y\theta\cos 2\theta t \\
&\quad + c_z\theta\frac{\cos\alpha}{\sin\alpha}\cos\theta t + R\theta^2\sin\alpha\sin\theta t \\
\frac{\partial^2 y}{\partial t^2} &= 4c_y\theta\cos\theta t\sin\theta t + 2c_x\theta\cos 2\theta t \\
&\quad + c_z\theta\frac{\cos\alpha}{\sin\alpha}\sin\theta t + R\theta^2\sin\alpha\cos\theta t \\
\frac{\partial^2 z}{\partial t^2} &= 0
\end{aligned}$$

$$\begin{aligned}
\frac{\partial^2 x}{\partial t\partial \alpha} &= -c_z\frac{\sin\theta t}{\sin^2\alpha} - R\theta\cos\alpha\cos\theta t \\
\frac{\partial^2 y}{\partial t\partial \alpha} &= c_z\frac{\cos\theta t}{\sin^2\alpha} - R\theta\cos\alpha\sin\theta t \\
\frac{\partial^2 z}{\partial t\partial \alpha} &= 0
\end{aligned}$$

$$\begin{aligned}
\frac{\partial^2 x}{\partial \alpha^2} &= -2\frac{c_z\cos\alpha}{\theta\sin^3\alpha}\cos\theta t + R\sin\alpha\sin\theta t \\
\frac{\partial^2 y}{\partial \alpha^2} &= -2\frac{c_z\cos\alpha}{\theta\sin^3\alpha}\sin\theta t - R\sin\alpha\cos\theta t \\
\frac{\partial^2 z}{\partial \alpha^2} &= -R\cos\alpha
\end{aligned}$$

All of the factors in the second derivatives are monotonic over easily determined intervals. This will allow finding maximum and minimum values over regions in a similar manner to that used for bounding the other parameterization.

The only remaining point to using this approach is to decide where to cross from l to α and back. A reasonable crossover point to choose is the point at which the slope of $l(\alpha) = \pm 1$. At this point the slope of $\alpha(l) = \pm 1$ as well. On either side of this point, the functions in l or α are not changing too quickly, which means that the bounds shouldn't grow too large. Doing this ensures that no divisions by zero will occur as those points are in the other parameter's domain. For example, when $l\theta = c_x \sin\theta t - c_y \cos\theta t$, then $\alpha = \pi/2$. The crossover point is straightforward to find as

$$\begin{aligned} \frac{dl}{d\alpha} &= \frac{c_z}{\theta \sin^2\alpha} = 1 \\ \alpha &= \arcsin\sqrt{\frac{c_z}{\theta}}. \end{aligned}$$

Despite dual parameterization, if the tool movement becomes flat enough without becoming completely flat, an alternative method for finding the intersection will become necessary or a different means of bounding triangle accuracy will have to be found. Otherwise, the number of triangles required will rise uncontrollably due to the degeneracy at the corners joining the nearly linear portions of the contact curve to the middle section wrapping around the cylinder body.

5.3.2 Bounding the Disc Edge Surface

Finding the bounds for regions of the disc edge surface is done in the same way as described above. A standard linear approximation is created and the error bound depends on the maximum absolute value of the second derivatives over the region of approximation.

The disc edge function is

$$\begin{aligned} x(t,\alpha) &= c_x t - R\sin\alpha\sin\theta t \\ y(t,\alpha) &= c_y t + R\sin\alpha\cos\theta t \\ z(t,\alpha) &= c_z t + R\cos\alpha. \end{aligned}$$

The first partial derivatives are

$$\frac{\partial x}{\partial t} = c_x - R\theta\sin\alpha\cos\theta t$$

$$\begin{aligned}
\frac{\partial y}{\partial t} &= c_y - R\theta \sin\alpha \sin\theta t \\
\frac{\partial z}{\partial t} &= c_z
\end{aligned}$$

$$\begin{aligned}
\frac{\partial x}{\partial \alpha} &= -R\cos\alpha \sin\theta t \\
\frac{\partial y}{\partial \alpha} &= R\cos\alpha \cos\theta t \\
\frac{\partial z}{\partial \alpha} &= -R\sin\alpha
\end{aligned}$$

giving second derivatives of

$$\begin{aligned}
\frac{\partial^2 x}{\partial t^2} &= R\theta^2 \sin\alpha \sin\theta t \\
\frac{\partial^2 y}{\partial t^2} &= -R\theta^2 \sin\alpha \cos\theta t \\
\frac{\partial^2 z}{\partial t^2} &= 0
\end{aligned}$$

$$\begin{aligned}
\frac{\partial^2 x}{\partial t \partial \alpha} &= -R\theta \cos\alpha \cos\theta t \\
\frac{\partial^2 y}{\partial t \partial \alpha} &= -R\theta \cos\alpha \sin\theta t \\
\frac{\partial^2 z}{\partial t \partial \alpha} &= 0
\end{aligned}$$

$$\begin{aligned}
\frac{\partial^2 x}{\partial \alpha^2} &= R\sin\alpha \sin\theta t \\
\frac{\partial^2 y}{\partial \alpha^2} &= -R\sin\alpha \cos\theta t \\
\frac{\partial^2 z}{\partial \alpha^2} &= -R\cos\alpha
\end{aligned}$$

All the factors in these equations are monotonic over quarter wave intervals. By using these intervals as starting subdivisions for regions, the bounds can be found in the manner described in the previous section.

5.3.3 Bounding the Developable Surface

The developable surface looks like

$$\begin{aligned} x(t) &= c_x t - \sin\theta t(\frac{c_x}{\theta}\cos\theta t + \frac{c_y}{\theta}\sin\theta t) \\ y(t) &= c_y t + \cos\theta t(\frac{c_x}{\theta}\cos\theta t + \frac{c_y}{\theta}\sin\theta t) \\ z(t) &= c_z t \pm \sqrt{R^2 - \frac{1}{\theta^2}(c_x\cos\theta t + c_y\sin\theta t)^2}. \end{aligned}$$

The first derivatives are

$$\begin{aligned} \frac{\partial x}{\partial t} &= c_x - c_x\cos 2\theta t - c_y\sin 2\theta t \\ \frac{\partial y}{\partial t} &= c_y + c_y\cos 2\theta t - c_x\sin 2\theta t \\ \frac{\partial z}{\partial t} &= c_z \mp \frac{(c_x\cos\theta t + c_y\sin\theta t)(c_y\cos\theta t - c_x\sin\theta t)}{\theta\sqrt{R^2 - \frac{1}{\theta^2}(c_x\cos\theta t + c_y\sin\theta t)^2}} \end{aligned}$$

giving second derivatives

$$\begin{aligned} \frac{\partial^2 x}{\partial t^2} &= -2c_y\theta\cos 2\theta t + 2c_x\theta\sin 2\theta t \\ \frac{\partial^2 y}{\partial t^2} &= -2c_y\theta\sin 2\theta t + 2c_x\theta\cos 2\theta t \\ \frac{\partial^2 z}{\partial t^2} &= \mp\Bigg(\frac{(c_x\cos\theta t + c_y\sin\theta t)^2(c_y\cos\theta t - c_x\sin\theta t)^2}{\theta^2(R^2 - \frac{1}{\theta^2}(c_x\cos\theta t + c_y\sin\theta t)^2)^{3/2}} \\ &\quad - \frac{(c_x\cos\theta t + c_y\sin\theta t)^2 - (c_y\cos\theta t - c_x\sin\theta t)^2}{\theta\sqrt{R^2 - \frac{1}{\theta^2}(c_x\cos\theta t + c_y\sin\theta t)^2}}\Bigg) \end{aligned}$$

Since the developable surface is flat in the z direction, we only need to worry about the accuracy in the x and y directions. We essentially need to know the maximum length in two dimensions of the curve defined by $x(t)$ and $y(t)$. The second derivatives are just sine waves, which are trivial to bound over an interval.

The border of the surface may be ignored, because it joins the surface swept out by the disc edge. Therefore, as long as the disc edge surface meets the tolerance requirement, so will the border of the developable surface.

5.3.4 Using Surface Points as Vertices

The previous sections tell us how to make an approximating plane for a point on the surfaces that make up the tool motion swept envelope. However, this is less than ideal as it stands. To use the method as described, points chosen must be the center of every polygon that makes up the surface approximation. In addition, vertices and edges must be determined by finding the intersections of neighboring polygons. This clearly is not desirable. Not only must the intersections be calculated, the polygons obtained will not be triangles, either requiring triangulation or somewhat less efficient line intersection code. It would be much preferable to choose points that lie on the surface and connect them into polygons, allowing us to know the vertices and edges from the start.

To allow us to do this, we need to know that the triangle or polygon we wish to use is within the region of tolerance for the points it represents. One way to do this is to find the regions of tolerance (box in which tangent plane accurately represents the surface) around the tangent plane of each vertex as described before. If the tolerance regions for the vertices connect without gaps, then those tangent planes will represent the surface within tolerance. However, if the connecting region has high enough curvature, the polygon won't be guaranteed to lie within the tolerance regions. We would have to test the polygon for inclusion in every region.

However, if we pick a point in the interior of the vertices and look at its region of tolerance, we can see if the polygon falls in it. If the whole polygon is within the region, the tangent plane is within tolerance of the polygon everywhere and the surface itself won't be more than twice the tolerance from the polygon. Therefore, we can adaptively build the meshes.

Using a starting grid broken into triangles based on the conditions outlined earlier, the middle point of the triangle is used to find the tangent plane for the tolerance test. If the triangle vertices lie in the tolerance region around the tangent plane for half the desired tolerance, the triangle is acceptable, otherwise it is subdivided and the process recursively followed.

The actual point used in the triangle doesn't matter in terms of the accuracy guarantee, since the whole triangle must lie inside tolerance region of the tangent plane to be considered small enough. However, if the plane of the triangle is nearly perpendicular to the chosen tangent plane, the triangle must be quite small to stay in the tolerance region. If the tangent plane is nearly parallel to the triangle, the triangle can be larger and still fit in the tolerance region. So, we pick the barycentric center of the triangle to obtain the tangent plane in the hopes that the tangent plane will be nearly parallel to the triangle plane. The barycentric center of a triangle is the intersection of the three lines drawn from each vertex to the center of the edge opposing that vertex.

5.4 Implementation and Analysis

5.4.1 Implementation Issues

Turning this method into a working program requires building the polygonal model and then finding intersections with it. To build the model, we must generate each of the described meshes and then merge them together. For intersection location, we must store this model to allow as efficient access as possible to different parts in space.

As we just suggested at the end of the previous section, each mesh is built adaptively. An arbitrary grid is used to begin the mesh generation. Each cell in the grid contains two triangles and all triangles are connected across edges to maintain the mesh. Then the triangles are placed in a queue. They are pulled off the queue and subdivided based on the test described above. Subdivided pieces are placed at the back of the queue and triangles that meet the tolerance requirement are placed on a finished list. This continues until the subdivision queue is empty.

Subdivision of a triangle is done by splitting it in half. The triangles start as right isosceles triangles in parameter space. The right angle is split by a line connecting to a new vertex in the middle of the opposing edge. This creates two new triangles of the same shape. The new vertex also affects the triangle on the other side of the edge being broken. This triangle is also subdivided. If the new vertex doesn't lie on the long edge of that triangle, it is recursively subdivided. The resulting triangle has the edge containing the new vertex as its longest. Then it can be subdivided as well.

This subdivision scheme does not comfortably allow for the multiple parameterization to be used. We decided to forgo the multiple parameterization due to the extra increase in code complexity. In the end, the alpha parameterization was not used since these equations contain division by the angle of rotation. In the tool movements found in our data, rotation angles are quite small, causing the program to run out of memory when trying to build the model.

The next step calls for joining the meshes together. One edge of each side mesh should lie completely on the surface of the disc mesh. To do this, we locate the side mesh corner on the disc mesh and break the disc triangle to contain the side vertex. Then, each vertex of the side mesh edge is added to the disc mesh until the end is reached. This process requires breaking disc triangles and side edge triangles to make sure that a seamless join is achieved. Using a power of two to select the starting grid densities helps the meshes line up better, resulting in fewer subdivisions.

In addition, one has to be careful when breaking edges. Because the join curve is not straight, a new vertex will not lie on a line between the surrounding ones. This can result in a change of the working triangle. One problem encountered was numerical inaccuracy when joining. A vertex may lie very near a triangle edge but not on it, causing problems such as being classified in the wrong triangle.

Once the polygonal model has been constructed, we need to store it for efficient intersection calculation. Several ray tracing methods exist for storing a collection of polygons and other primitives for efficient intersection calculation. For this program, a BSP tree was used. As used, this is a variant of octree in which a tree of half spaces formed by dividing planes is constructed. All objects are placed on one side of the plane or the other, or both if the object crosses the dividing plane. Each half space is recursively subdivided until the list of objects is small enough or the maximum tree depth is reached. This structure divides up the triangles so that a line need only test those that lie near and may ignore the others.

Memory usage turns out to be a significant problem. Tool movements that have a lot of rotation require lots of triangles. The system ran out of memory beyond about 320,000 triangles. Even this many left no room to build the BSP tree. Even for small angle movements, certain tool movements can easily generate enough triangles to use up all available memory. Tool movements in the tests often used 40,000 to 100,000 triangles, depending on the accuracy level and the tool movements involved. The starting mesh densities set a bottom limit of around 3000 triangles. This level was chosen since initial grid generation is more efficient than subsequent subdivision.

Our triangles required about 100 bytes of storage each. Pointers were used to link the individual mesh triangles to their neighbors, link to other meshes when assembling, form linked lists while processing (both in mesh generation and later in storing in BSP tree). Other fields used were indices allowing fast identification of matching vertex numbers in adjacent triangles, vertex indexes into vertex array, surface type identifier, and maximum and minimum coordinates of triangle vertices.

In addition, the BSP tree takes up a significant amount of space. Each node stores its maximum and minimum coordinates and pointers to children as well as a list of objects, if the node is a leaf. The list of objects is actually a list of pointers to objects to reduce storage costs. An object in multiple cells is only stored once and represented by the small pointer structures. However, the sheer number of triangles still adds up in tree storage costs.

We had access to a DEC Alpha workstation on which to perform tests. This machine had 96 megabytes of real memory, yet we still routinely ran out of memory.

One reason for memory problems stems from the fact that some tool movements exhibit a large rotation. For a given tolerance level, doubling the rotation will require roughly double the number of triangles, since the surface curvature should remain relatively consistent.

The memory concerns are also related to the problems the method has with nearly flat tool movements. The critical curve on the side of a cylinder becomes degenerate, resulting in the need for large numbers of triangles. This increases to infinity when the tool movement is completely flat (four-axis). As a result of this, the three-axis approximation method was used in cases where the tool movement was flatter than a fixed value. To be able to finish most of the test cases, the defaulting occurred almost half of the time.

This phenomenon is counter-intuitive and deserves some explanation. A four-axis tool movement (in which the rotation plane contains the linear motion vector) has flat sides, yet causes the bounds to blow up. As described before, the cylinder body contacts the tool envelope in a barber-pole shaped curve. This curve degenerates as the tool movement flattens, until it consists of a straight line down the side which turns a sharp corner, travels around the cylinder body 180 degrees, and continues down the other side. The method we present requires very dense sampling as the curvature increases at these corners. Hence the memory is exhausted on tool movements that are too flat.

Unlike the other programs, the complexity of the implementation is a factor. Our implementation ran 6600 lines of C code. Both coding and debugging were much more involved than with the other methods. Optimization would involve even more effort.

We tested a version of the polygonal method in which the bounds used during mesh generation were replaced by a simple and looser test. When a triangle is to be subdivided, the barycentric center of the triangle in parameter space is tested for distance in real space from the triangle plane. If this distance is less than the tolerance the triangle is accepted.

In this program, adaptive subdivision is still used. This means that the number of triangles can still balloon if the tool movement is too flat (the z component of the linear motion approaches 0). Therefore, the three-axis approximation was still used to handle these cases.

The unbounded program provides a more realistic view of the potential performance of the polygonal method outlined in this chapter. Model sizes usually ran from 1500 to 4000 triangles, starting from a bottom limit of around 200 triangles. This is 10–20 times fewer triangles than the bounded method generated. Although sampling just one point in a triangle is no guarantee of flatness and in certain cases provides a very bad approximation, it is likely to produce acceptable results. The surface is not highly curved in practice. On the other hand, the min-max bounds presented are much worse than they need to be.

5.4.2 Performance Analysis

Preprocessing time depends on two things—how long it takes to build the model, and the time to put the model into the tree structure, which depends largely on the model size. The number of triangles in the model is dependent on the curvature of the tool movement envelope, a very complex relationship. It is also related to the curvature in the two parameter directions. In the direction of cylinder length, this curvature can become infinite as the tool movement becomes flat, because the crossover regions, exhibited in figure 5.2, become sharp corners. However, in this case, the envelope surface curvature is not high at all. The large triangle demands of nearly flat tool movements is mainly due to the parameterization.

Finally, the point chosen to determine triangle accuracy affects the size of a triangle by changing the tangent plane, and the bounding of the equations can depend on the numbers involved, further complicating analysis.

The time needed to build the efficiency tree depends on several factors. The size of the model is most important. At each node, all the objects are placed in the child whose half-space they lie in, which requires going through the complete list of objects at that node. Objects split by the dividing plane end up in both halves, which causes the model size to grow. In addition, build time is affected by the recursion limit and the number of objects allowed in each node.

Intersection time depends on the number of cells the line must traverse and the number of objects in each cell to be tested for intersection before an intersection is found. Therefore, it depends on the size of the model as well. Intersection time also depends on how well objects are distributed in the tree. If many objects occupy a node as a result of not being split up, any line hitting that cell must test for intersection with all of them. The better the objects are distributed over the leaf nodes of the tree, the better the intersection performance will be.

The result of all this is a complicated relationship between all the components. Statistical analysis of what we should expect might be possible but would be very difficult. As we will see in Chapter 7, while intersection testing times were acceptable, preprocessing costs were incredibly high by comparison to other methods. due to the bounds available.

The surface representations we have presented are very natural, arising directly from the application of differential geometry to the problem. However. we have already seen that these representations don't work well in degenerate cases. These cases are in fact the ones we would expect to have the best results, not the worst. However, these representations account for the complete surface of a tool movement in a simple manner. Other surface representations may be much more useful for building meshes but will most likely be more complex.

Exploring further the theoretical performance of the polygonal approach as presented here becomes less interesting in light of the implementation complexity and the poor performance results. Triangle generation is at its worst when it should be at its best. Finding better ways of generating fewer triangles, *while achieving provably good accuracy*, is an important open problem.

Chapter 6

Discrete Stepping Along the Fixed Line

6.1 Introduction

Our goal in finding the intersection of the fixed line $M = P + uV$ with the swept surface created by the five-axis tool movement is to locate the point with the smallest value of u along the length of the line that is within or on the swept envelope boundary. We would like have the answer be accurate to some user specified tolerance.

One obvious approach would be to evaluate each point on the line for inclusion in the swept solid. i.e. is it inside, on, or outside the swept envelope. The point inside or on the envelope with the most negative u is returned as the intersection point.

Implementing this algorithm as stated has two problems: testing whether the point is inside or outside the envelope is difficult, and testing every point on the line is impossible. If we had an implicit equation for the swept surface of the finite cylinder, the first problem would not exist (except for floating point limits, which we assume are acceptable). We could just plug the coordinates of the point in question into the implicit equation and read off the value as a yes or no. Unfortunately, a closed form implicit equation seems difficult to generate even for an infinite cylinder, much less a finite one.

Assuming we had an accurate test to tell us when a point is inside or outside the envelope, we still have to contend with testing the points on the line. Since we can't test them all, we would need to create an equation in u that is 0 when the line intersects the envelope in order to get an exact answer. With this equation in hand, the smallest root would be the u value of the intersection we seek. This leads to equations like those at the heart of the 1-D approach described earlier. Again, we will run into equations that are not analytically solvable. Some error will have to be introduced in the numerical solution process.

We can solve the second problem by taking an inherently discrete approach. Start by choosing a point on the fixed line that is guaranteed not to be included in the swept solid. Then, stepping along the line by some Δu, check each point for inclusion. If a point is found to be contained by the envelope during the search, that point and the previous one bound the actual intersection of the line and the envelope. This answer can then be refined by a bisection routine.

A discrete approach to finding the intersection introduces a source of error. An intersection may go undetected between two points that do not intersect the sweep. The choice of step size will control the amount of error allowed because of this. The smaller the step size, the less the cylinder can gouge between the sample points.

We still need a way to test whether or not a point falls inside the swept envelope. Over the time interval of the tool movement, the cylinder cuts a path in space and may cut the point in the process. We can devise a test by transforming into the cylinder's coordinate system so that the cylinder holds still while the point moves (See Figure 6.1). The point will sweep out a space curve in the transformed problem. If this space curve intersects the stationary cylinder, the point hits the cylinder during the sweep and we have found a point that lies within (or on) the cylinder's swept envelope. We hope that this method will give us a simple set of equations to solve.

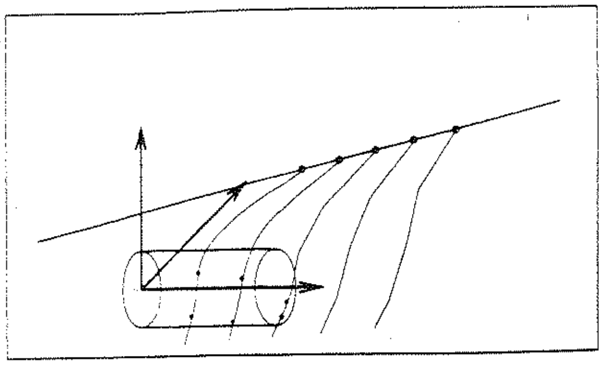


Figure 6.1: Trajectories of points on the line

6.2 Algorithm Description

We implement this method in several pieces. First, we find a point on the line guaranteed to be outside the envelope. This gives us a place to start our search, preventing wasted point tests that definitely will not hit the cylinder. Then we need to determine the interval size to step down the line by. Finally, we must have a test for deciding whether a point hits the cylinder or not.

These functions combine to give us the algorithm. Starting at the first point that may hit the envelope, we walk down the line, picking points a step size apart. Each point is tested. The first hit found and the previous point bracket the real intersection, which can be found by applying a bisection routine.

6.2.1 Starting Point

Finding a starting point can be accomplished by putting a bounding box around the whole tool movement and using the intersection of the line and bounding box as a starting point. The simplest bounding box would just be a box around the whole tool movement. This could be computed using techniques from chapter 3, but the box would usually be a loose fit. We suggest using a three-axis tool movement as a bounding box. Ignoring the linear portion of the tool movement, a virtual cylinder is selected that is large enough to contain the complete rotation of the tool. This virtual cylinder then undergoes the same translation as the tool. Since the virtual cylinder can contain the tool at all times during the movement, its envelope bounds the tool movement. In addition, we can find the intersection of a line and a three-axis movement easily. For efficiency purposes, we could break the tool movement into several pieces and bound each of them individually, as this reduces the amount of rotation per segment and therefore decreases the size of the bounding cylinder.

6.2.2 Step Increment

Next we must pick a step size. Call it Δu. This directly affects the tolerance of the algorithm. Here we address the error caused by the cylinder grazing the line segment between two points undetected. The scan procedure will start at a point u_0 that is outside the swept envelope of the tool movement. If the point at $u_0 + \Delta u$ is in or on the envelope, then we return it and call it the intersection. (We can use these two points as boundaries for doing a binary search to locate the intersection point more accurately.) If it misses (i.e., falls outside the envelope), then one of two things has happened. Either no points on the line segment $(P + u_0V, P + (u_0 + \Delta u)V)$ hit, in which case the procedure returns the point located (or refines to obtain the exact answer), or else some points in that segment hit the swept envelope.

If part of the line segment hits the cylinder, but both ends miss, it either hits the side of the cylinder or the edge around one of the ends (we'll assume that the step size is chosen so that the line segment isn't long enough to enter and exit through opposite ends). See Figure 6.2. For a side cut, the deepest penetration occurs if the segment is perpendicular to the cylinder axis. This can be seen by looking at a projection down the axis. The segment forms a chord on the circle. If the segment is not perpendicular, the projection will be shorter, meaning a lesser protrusion. The maximum protrusion depth is $d = R - \sqrt{R^2 - \frac{\Delta u^2}{4}}$ where R is the cylinder radius. Since we would like d to less than the user-specified tolerance T, we should set $\Delta u \leq 2\sqrt{2RT - T^2}$.

An end cut will have the deepest penetration when the line segment is in a plane with the cylinder axis and the two endpoints are equidistant from the edge of the end cap. This can be seen by looking at a projection perpendicular to the cylinder axis. The projection is a rectangle and the segment cuts the corner, forming a triangle. If the line segment is skew to the axis, the projection will be shorter, reducing the altitude of the triangle. The distance from the corner to the line segment is the maximum distance the segment gets from the surface of the cylinder. This distance is maximized when the triangle is isosceles. This results in a depth $d = 1/2\Delta u$. It also means that no point on the segment can intersect a cylinder of radius $R - \Delta u$ and of the same length as the one in which we are interested. Since we want d to be no greater than the tolerance T, we should set $\Delta u \leq 2T$.

We have two calculations to limit the size of Δu. They are the same when

$$\begin{aligned} T &= \sqrt{2RT - T^2} \\ T^2 &= 2RT - T^2 \\ T^2 &= RT. \end{aligned}$$

We can see from this that as long as R is greater than T, a reasonable requirement, we only need to use the bound $\Delta u \leq 2T$.

6.2.3 Inclusion Test

The remaining part of the algorithm is creating a test that determines if a point ever hits the cylinder over $0 \leq t \leq 1$. The cylinder (and any other point) undergoes the

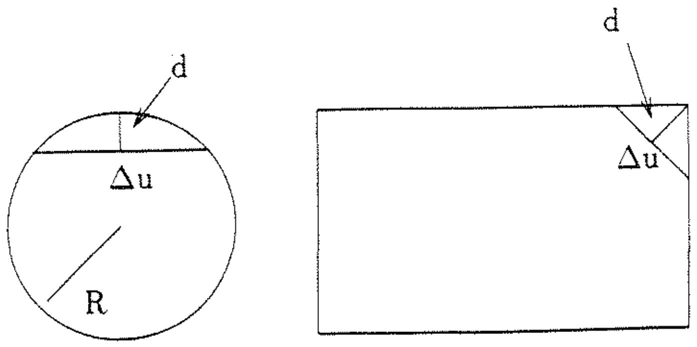


Figure 6.2: Side and end cuts by line segment

transformation

$$\begin{aligned} x(t) &= x_0 \cos\theta t - y_0 \sin\theta t + c_x t \\ y(t) &= x_0 \sin\theta t + y_0 \cos\theta t + c_y t \\ z(t) &= z_0 + c_z t \end{aligned}$$

where (x_0, y_0, z_0) is the point being transformed, θ is the angle of completed rotation, and (c_x, c_y, c_z) is the end point of the linear translation that starts at the origin (see section 2.4).

The inclusion test must determine if the point on the line is cut by the moving cylinder. Deciding if the point is inside the cylinder at time t is straightforward. We check the distance from the point to the cylinder axis at t and then see if the projection is within the end bounds. However, we want to do this as a function of t. If we keep the cylinder stationary and let the point move in the cylinder's frame of reference, the test becomes simpler. We need only check that $x(t)$ for the point lies between 0 and 1, and that $y(t)^2 + z(t)^2 < R^2$, as opposed to the more complicated form for arbitrary orientation.

Now we must determine the path of the tested point in the cylinder's frame of reference. The cylinder's coordinates have translated by $(c_x t, c_y t, c_z t)$ and rotated around the z direction by θt radians. Negating the translation centers the cylinder at the origin, and rotating by $-\theta t$ will place the cylinder axis back on the x-axis. The point's coordinates will be in the cylinder coordinate system if we apply this reverse transformation to it as well. The transformation applied to a point (x, y, z) is

$$p'(t) = \begin{pmatrix} x(t)' \\ y(t)' \\ z(t)' \end{pmatrix} = \begin{pmatrix} (x - c_x t)\cos\theta t + (y - c_y t)\sin\theta t \\ (y - c_y t)\cos\theta t - (x - c_x t)\sin\theta t \\ z - c_z t \end{pmatrix} \tag{6.1}$$

A point whose trajectory intersects the fixed cylinder lying on the x-axis will have:

$$\begin{aligned} 0 &\leq x(t) \leq L \\ R^2 &\geq y(t)^2 + z(t)^2 \end{aligned}$$

with L being the cylinder length. If we substitute in the reversed transformation, we get:

$$0 \leq (x_0 - c_x t)\cos\theta t + (y_0 - c_y t)\sin\theta t \leq L \tag{6.2}$$

$$R^2 \geq ((y_0 - c_y t)\cos\theta t - (x_0 - c_x t)\sin\theta t)^2 + (z_0 - c_z t)^2 \tag{6.3}$$

These two inequalities give us a test to determine if a point is in fact cut by the tool movement. First we locate subintervals of t in which 6.2 holds. The point cannot be cut outside these intervals as its x value is outside the cylinder's boundaries. The second step is to ascertain if 6.3 holds within the intervals just found. If it is true anywhere in any of the intervals, the cylinder cuts the point being evaluated. Note that we only need to know that it holds somewhere, meaning that we only need to find one occurrence of the condition and that we don't need to know where it happens.

Solving Condition 6.2

To find the intervals where equation 6.2 holds, we can locate the roots of

$$0 = (x_0 - c_x t)\cos\theta t + (y_0 - c_y t)\sin\theta t,$$

and

$$0 = (x_0 - c_x t)\cos\theta t + (y_0 - c_y t)\sin\theta t - L$$

over $0 \le t \le 1$. The solutions to these two equations will give us the bracket ends for the intervals we seek. It is a difficult proposition to find these roots directly. However, if we know the maxima and minima of $(x_0 - c_x t)\cos\theta t + (y_0 - c_y t)\sin\theta t$, we can then easily determine if 0 and/or L lies between any adjacent pair. Then it would be a simple matter to apply a standard root finder such as found in [29, Ch 9]. Since we know the root is in there, the routine will find it. We also know that the interval contains only one root for each of 0 and L, so we are sure of finding them all.

Finding the roots of the derivative is the same problem as finding the roots of $(x_0 - c_x t)\cos\theta t + (y_0 - c_y t)\sin\theta t = 0$, since the derivative is

$$\frac{d}{dt} = (c_x\theta t - x_0\theta - c_y)\sin\theta t + (y_0\theta - c_x - c_y\theta t)\cos\theta t.$$

The equation we want to solve, then, is of the form

$$0 = (A + Ct)\sin\theta t + (B + Dt)\cos\theta t.$$

Dividing by $\cos\theta t$ and rearranging, we have

$$\tan\theta t = -\frac{B + Dt}{A + Ct}.$$

The right hand side of this equation is a hyperbola with horizontal asymptote of $y = -\frac{D}{C}$ and vertical asymptote of $t = -\frac{A}{C}$. If we take the hyperbola equation and

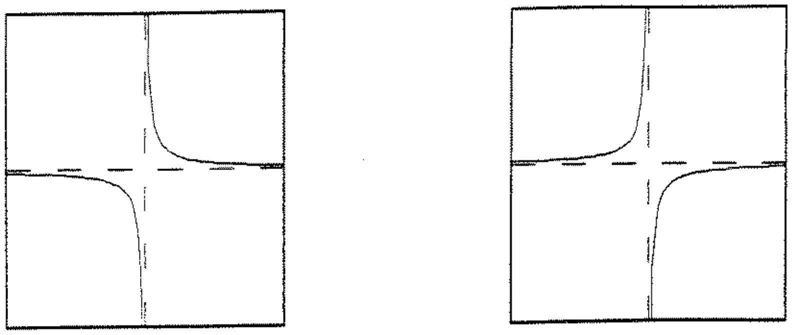


Figure 6.3: Hyperbolas when $bc < ad$, $bc > ad$

rearrange it, we can turn it into the form of a hyperbola:

$$\begin{aligned} y &= -\frac{B+Dt}{A+Ct} \\ Ay + Cty + Dt &= -B \\ \frac{A}{C}y + ty + \frac{D}{C}t &= -\frac{B}{C} \\ (t+\frac{A}{C})(y+\frac{D}{C}) &= -\frac{B}{C} + \frac{AD}{C^2} \end{aligned}$$

The constant $-\frac{B}{C} + \frac{AD}{C^2}$ controls the shape of the hyperbola. The magnitude controls the sharpness of the curve. Setting equal to 0 and rearranging, we can get the discriminant $BC - AD$. If $BC > AD$, the curve sits in the first and third quadrant offset by the asymptotes. If $BC < AD$, the curve sits in the second and fourth quadrant. If $BC = AD$, the curve is just the two asymptote lines.

Although it is possible for both portions of the hyperbola to intersect $\tan\theta t$ if the vertical asymptote lies in the interval $[0, 1]$, we can treat each portion individually. When $BC < AD$, the curve consists of the upper right portion and the lower left portion. In each case, the curve clearly intersects $\tan\theta t$ at most once because the tangent is increasing monotonically while the hyperbola pieces are both decreasing.

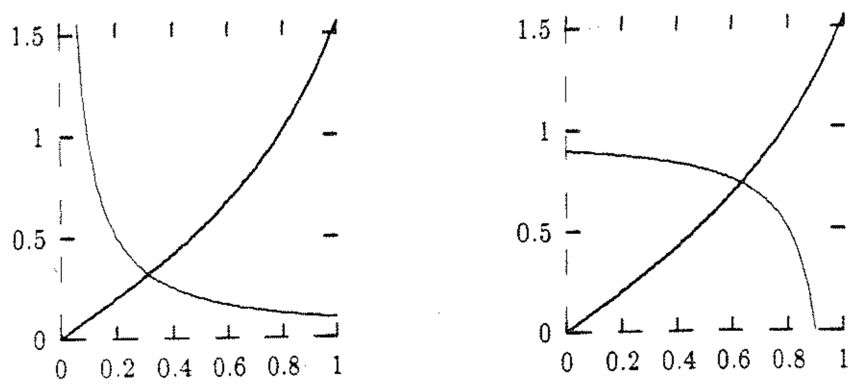


Figure 6.4: UR, LL hyperbola portions crossing tangent, respectively

If $BC = AD$, we just need $t = -A/C$ and $t = -\arctan D/C$.

When $BC > AD$, the situation is trickier. The lower right hyperbola can cross through $\tan\theta t$ twice, because it arcs away from the tangent curve. To find these intersections, we can look for the minimum of $\tan\theta t - \frac{B+Dt}{A+Ct}$ since we now know it will be U-shaped (or some piece thereof) over $[0, 1]$. Assuming the minimum is less than zero, the intersection points will be to either side.

The upper left hyperbola curves in the same direction as $\tan\theta t$. Therefore it isn't immediately obvious how many times the curves may cross. To settle the question we'll turn to the derivatives. If some curve $F(t_0) > G(t_0)$, then to have an intersection at t_1, we must have $\frac{dF}{dt} < \frac{dG}{dt}$ at t_1. If they again cross at t_2, then $\frac{dF}{dt} > \frac{dG}{d}t$ at t_2. Thus, to have n intersections between the two curves, the derivatives must cross at least $n-1$ times.

The derivative of the tangent curve is

$$\frac{d}{dt} = \frac{\theta}{\cos^2\theta t}$$

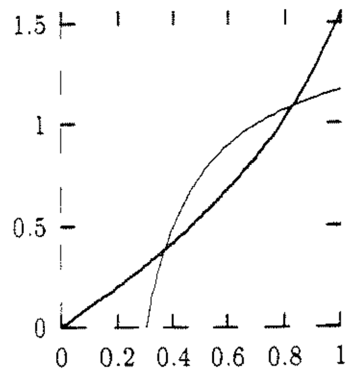


Figure 6.5: LR hyperbola portion crossing tangent twice

and for the hyperbola, we have

$$\frac{d}{dt} = -\frac{D(A+Ct) - C(B+Dt)}{(A+Ct)^2} = \frac{BC - AD}{(A+Ct)^2}.$$

This doesn't immediately tell us anything. However, if

$$\frac{\theta}{\cos^2 \theta t} \quad > (\text{or} <) \quad \frac{BC - AD}{(A+Ct)^2},$$

then,

$$\begin{aligned} \frac{\cos^2 \theta t}{\theta} \quad &< (\text{or} >) \quad \frac{(A+Ct)^2}{BC - AD} \\ \frac{|\cos \theta t|}{\sqrt{\theta}} \quad &< (\text{or} >) \quad \frac{|(A+Ct)|}{\sqrt{BC - AD}}. \end{aligned}$$

Since the interval of interest is $t \in [0,1]$ and $\theta < 1$, $|\cos \theta t| = \cos \theta t$. We are interested in the upper left hyperbola, so $|(A+Ct)| = \pm(A+Ct)$, whichever has negative slope. As shown in figure 6.6, at most two intersections are possible. As a result, at most three intersections may occur between the tangent and hyperbola.

To find the intersections, we first find the intersections of the cosine and line. If $\frac{1}{\sqrt{\theta}} > \pm\frac{A}{\sqrt{BC-AD}}$ and $\frac{\cos\theta}{\sqrt{\theta}} > \pm\frac{(A+C)}{\sqrt{BC-AD}}$, there may be two intersections, $\pm$ indicating

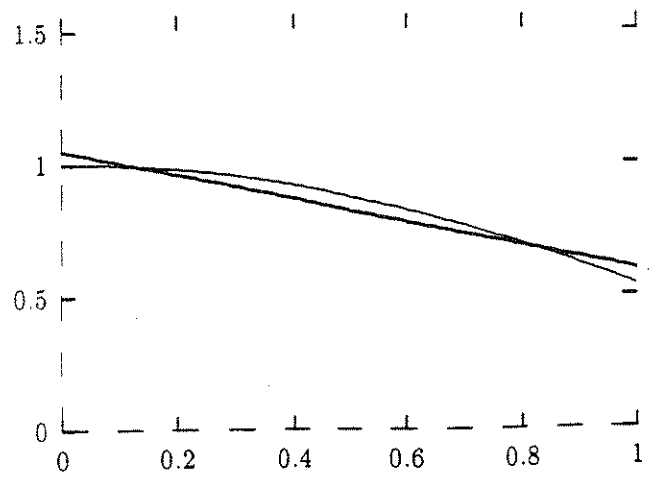


Figure 6.6: Cosine hitting line twice

the appropriate sign. In this case, we compute $t_s = \frac{1}{\theta} \arcsin \frac{\pm\sqrt{\theta C}}{\theta\sqrt{BC-AD}}$. This tells us where the slopes are the same. Assuming $t_s \in [0,1]$ and $\frac{\cos \theta t_s}{\sqrt{\theta}} > \pm \frac{(A+Ct_s)}{\sqrt{BC-AD}}$, the intersections are on either side and may be located with a root finder. If either end of the line is below the cosine at $t = 0$, only one intersection is possible, in which case the two endpoints bracket it and a root finder may be applied.

With the derivative intersections in hand, we now can locate the intersections of the tangent and hyperbola. Only one intersection is possible between each adjacent pair of derivative intersections and interval endpoints. Thus, we can apply a root finder to each interval in turn to locate them.

Since these points are the maxima and minima of the original equation, they are used as limits for finding the intervals that satisfy condition 6.2.

Solving Condition 6.3

We now have located the ends of the intervals where the point may be cut by the cylinder. Next we need to determine if condition 6.3 holds anywhere in those intervals. To do that we can determine if there are any roots to the equation

$$0 = ((y_0 - c_y t)\cos\theta t - (x_0 - c_x t)\sin\theta t)^2 + (z_0 - c_z t)^2 - R^2.$$

If any roots exist, the cylinder will cut the point in question. Remember that we only need to determine if a root exists to answer the test, not how many nor their location(s). This means we can stop as soon as we identify a root's existence or learn that none exist.

This function behaves in a complicated manner, but we will show how to find an upper and lower bound for its slope,

$$\begin{aligned}\frac{d}{dt} &= 2(c_x\sin\theta t - c_y\cos\theta t - (y_0 - c_y t)\theta\sin\theta t - (x_0 - c_x t)\theta\cos\theta t) \\ &\quad ((y_0 - c_y t)\cos\theta t - (x_0 - c_x t)\sin\theta t) - 2c_z(z_0 - c_z t),\end{aligned}$$

over any given interval. Given those values, we can determine if an interval might have a root. In the worst case, ignoring higher derivative information, the curve could descend along a line from the smaller end of the interval, turn a sharp corner, and climb up along a line to the larger end of the interval. The minimum and maximum slopes limit these two lines, respectively. The deepest point over the interval, the intersection of these two lines, either falls above 0, on 0, or below 0. In the first case, no root can exist in this interval and searching can stop here. Otherwise, we can divide the interval and recursively search the subintervals. This is continued until the search is stopped in all active intervals, a point less than 0 is found (indicating that a root does exist — again, we don't need to know where), or the limits of machine precision are reached.

Obtaining bounds on the slope over an interval is mostly straightforward. We can simplify the derivative equation as follows:

$$
\begin{aligned}
0 &= A + Bt + (C + Dt + Et^2)\cos 2\theta t + (F + Gt + Ht^2)\sin 2\theta t \\
& \quad \text{with} \\
A &= -xc_x - yc_y - 2zc_z, \\
B &= c_x^2 + c_y^2 + 2c_z^2, \\
C &= xc_x - yc_y - 2\theta xy \\
D &= c_y^2 - c_x^2 + 2\theta xc_y + 2\theta yc_x \\
E &= -2\theta c_x c_y \\
F &= xc_y + yc_x + \theta(x^2 - y^2) \\
G &= 2\theta(yc_y - xc_x) - 2c_x c_y, \text{ and} \\
H &= \theta(c_x^2 - c_y^2).
\end{aligned}
$$

If we can bound the individual terms, they can be assembled into bounds for the whole function. The linear term is easily bounded. Since $A + Bt$ is a line, the minimum and maximum values occur at the interval endpoints.

The other two terms are a parabola multiplied by a sine wave. If we choose sample points at the minimums of the parabolas, we limit intervals to monotonic portions of the parabolas. The sine waves can also be limited to monotonic sections. Choosing sample points at $t = \frac{n\pi}{2\theta}$ with n an integer limits $\cos 2\theta t$ to monotonic intervals, while samples at $t = \frac{\pi}{4\theta} + \frac{n\pi}{2\theta}$ likewise gives monotonic intervals for $\sin 2\theta t$. Therefore, sample points placed at $t = \frac{n\pi}{4\theta}$ will generate both sets of points and guarantee monotonic intervals for each function. In addition, both $\cos 2\theta t$ and $\sin 2\theta t$ will be completely positive or negative over each interval.

Now, since both the parabolic and the trigonometric factors of each term are monotonic over the interval, we can place bounds on the values they take. If the absolute value of both functions are moving in the same direction, the values at the endpoints will suffice. If they are opposed, however, the maximum value may occur somewhere in between. The simplest approach is to take the four products of the values at the ends of the linear and sine wave factors and choose the largest and smallest of these. This will guarantee that the actual slope falls between the chosen values.

Once we have maximum and minimum values over an interval for the individual terms, they can be combined to give overall maximum and minimum values for the interval. Notice that since the trigonometric terms are close to linear over small intervals, decreasing an interval will produce a reasonably corresponding reduction in the span of the maximum and minimum values. This gives us an assurance that the search will make progress as it divides the intervals.

6.3 Optimizations

There are several ways in which we can improve the performance of this algorithm. It would be useful to take larger steps down the line in regions where the point misses the cylinder. This especially true in cases where the line is never cut by the cylinder and we must walk down the entire line segment selected earlier. Another area that can be improved is the bounding box used to select the line segment to be tested. The better the box, the smaller the chosen segment, and thus fewer point tests will have to be performed.

6.3.1 Bounding Box Improvements

Right off, one simple culling test can be performed. When $R^2 - (z_0 - c_z t)^2 < 0$, no intersection can occur. Therefore we can make sure that all intervals in question fall between $t = \frac{z_0 \pm R}{c_z}$, possibly deleting some intervals that would otherwise be searched.

We could also test $y(t) = \pm R$ in the same manner as we did for $x(t)$. This would result in further interval limits, although it is not clear that the computation saved by doing this would not be counterbalanced by the time it takes to find the limited intervals.

The above two tests would in effect create a bounding surface that amounts to sweeping a box containing the cylinder. This is certainly tighter than using a three-axis tool movement to bound the tool envelope, although the computation is clearly more complex. The decision to do these tests has to be made by doing real-world comparisons.

As Chapter 7 shows, the z culling is a clear win. The cost of and complexity of the test is minimal while the gains can be substantial. Culling y was not explored since preliminary tests showed no gains at all, while increasing the program complexity substantially since it requires a nonlinear equation search.

6.3.2 Step Size Improvements

The step size is dictated by the tolerance and the size of the cylinder. If we use a larger cylinder of radius R' around the real cylinder, a larger step could be taken without the real cylinder gouging the line more than T.

The basic search procedure is modified to take advantage of a larger step size. Using a large step (but smaller than the cylinder length) and large radius, a step is taken and the point tested. If it misses, we can be sure that the real cylinder doesn't

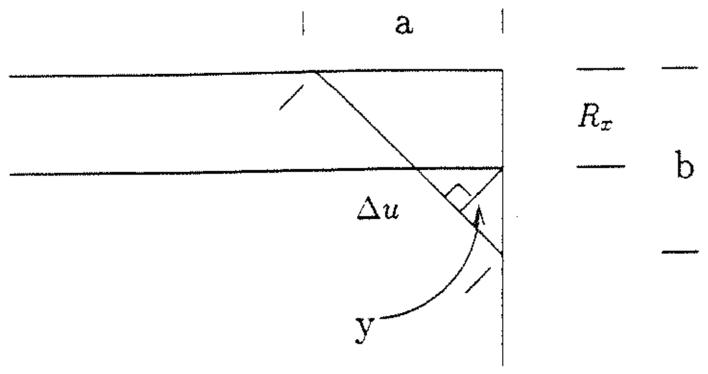


Figure 6.7: Deep cut for larger cylinder and step size

gouge by more than the tolerance. If it hits, it may still miss the real cylinder, so we need to reduce the step size and retest.

This recursion continues until a miss occurs, indicating that no contact with the real cylinder is found, or the test is done with the real cylinder. This has the benefit of speeding up cases where the line misses or doesn't hit until late along the line, while adding only a few tests when locating an early hit. In the worst case, however, the line may be very near the cylinder while not hitting, forcing resizes every step of the way but yielding no results.

Figure 6.7 represents the use of a larger cylinder and step size. We can choose Δu here to be some power of two times the Δu we have been using. This makes it easier to apply a binary search according to the step size. $R_x = R' - R$, representing the extra amount we have added to the cylinder. a and b are the two remaining sides of the big triangle. y is the distance from the corner of the real cylinder to the line segment.

Our goal is to find a way to select R_x given T and Δu. Two things are immediately apparent. We have

$$\Delta u^2 = a^2 + b^2, \text{ and}$$

$$\frac{\Delta u}{a} = \frac{b - R_x}{y}.$$

Making a substitution and squaring, we get

$$\Delta u^2 = \frac{\Delta u^2 - b^2}{y^2}(b - R_x)^2.$$

Rearranging to solve for y gives

$$y = \frac{\sqrt{\Delta u^2 - b^2}}{\Delta u}(b - R_x).$$

Since y is the deepest cut the line segment makes into the real cylinder, we need $y \leq T$. Therefore, if we set the maximum value of y to be T, we can guarantee that the larger step and cylinder don't cause us to miss an error larger than T for any b. Taking the derivative of y gives

$$\frac{dy}{db} = -\frac{b(b - R_x)}{\Delta u\sqrt{\Delta u^2 - b^2}} + \frac{\sqrt{\Delta u^2 - b^2}}{\Delta u}$$

which can then be set to 0 to find the maximum y value. If we do this and then solve for b, we have

$$b = \frac{R_x \pm \sqrt{R_x^2 + 8\Delta u^2}}{4}.$$

We can assume that $\Delta u > R_x$, otherwise the line segment couldn't hit the real cylinder at all. The additive solution is guaranteed to be positive, so we will work with that one. This is the value of b when y is at a maximum. If we limit y to be no larger than T, our error bound is assured. Therefore, plugging b into the equation for y, we have

$$y \leq T = \frac{(3R_x - \sqrt{R_x^2 + 8\Delta u^2})\sqrt{\Delta u^2 - \frac{(R_x + \sqrt{R_x^2 + 8\Delta u^2})^2}{16}}}{4\Delta u}.$$

We then would like to solve this for R_x. We assumed that $\Delta u = nT$, so given that, Mathematica (a symbolic math program) gives us six solutions for R_x. Only one is positive for $n > 2$, which is

$$R_x = \frac{T}{\sqrt{3}}\sqrt{1 + 3n^2 + \frac{2^{\frac{2}{3}} - 27\ 2^{\frac{5}{3}}n^2 + (2 + 270n^2 - 729n^4 + 3^{\frac{3}{2}}n(4 + 27n^2)^{\frac{3}{2}})^{\frac{2}{3}}}{2^{\frac{1}{3}}(2 + 270n^2 - 729n^4 + 3^{\frac{3}{2}}n(4 + 27n^2)^{\frac{3}{2}})^{\frac{1}{3}}}}.$$

This is messy, but easily computable.

As long as lines don't come so close to the cylinder that resizing occurs often, this should perform better than taking small, constant-sized steps. In fact, testing in Chapter 7 shows that it is a huge success. As long as the distance to the intersection is long enough to amortize the extra costs, the optimization pays for itself handsomely.

6.4 Implementation and Analysis

The implementation of the step programs is quite straightforward. The algorithm is simple; the only real complexity lies in the point testing, which has been described already. Tool movements are chopped into segments of less than 45 degrees. A three-axis tool envelope surrounds the tool movement to act as a bounding box. The line intersections with this envelope are used as starting and ending points for the search. The basic method steps from start to finish, testing each point as described until the end point is reached or an intersection is found. If an intersection is found, the actual point of first intersection lies between the located point and the previous one. Binary search is used to refine the answer to locate the actual envelope boundary.

The z culling optimization is trivial to implement. The large step optimization is more complicated. Basically, we keep a table of step size information such as current radius and maintain the alignment data necessary to decide when it is time to try taking a bigger step again, as well as how big a step to try.

The preprocessing done for this method is nearly nonexistent. Intersection time for the basic method should be inversely proportional to the step size, of course. This is also true of the z culling optimization, as the same number of point tests is performed, just at a faster rate. The large step algorithm is more complex, complicated by variable distances of the point trajectory to the cylinder. In the simplest model. the program would take a constant number of steps, getting very near the intersection, then drop the step size and take a constant number of steps again, drop the step size again, and so on until the minimal step size is reached and the intersection is found. In this case. the running time should be inversely proportional to the log of the step size, resulting in great savings. Real tests aren't quite as favorable, but unless the point trajectories spend a lot of time very near the cylinder but not touching, substantial savings are still realized.

A source of potential speedup is curve coherence. Since each point tested is very close to the one that follows, it is reasonable that the resulting curves generated will be similar. It may be advantageous to maintain the curve information as each point is tested and incrementally modify it to get resulting curves and roots for the next point test. The large step algorithm might make this sort of speedup more difficult to apply since it doesn't proceed as regularly down the line.

Chapter 7

Results and Conclusion

We have described several ways to find the intersection of a five-axis tool movement and a line. On that basis alone, there is no way to choose between them. Therefore, we have implemented each method and performed tests to look at how each method performs. We examined the effects of problem size in terms of number of tool movements and number of intersections per tool movements. We also looked at the effects of varying tool movement rotation angles and linear movement distances. And we looked at the performance of each method relative to changes in intersection tolerance. Last, we looked at the performance of the methods on real-world data, both with respect to tolerance and relative to each other. Finally, we wrap up the thesis and look at future directions for research.

7.1 Testing

7.1.1 Programs

We have implemented each of the methods described in this thesis in C in a UNIX environment. The timing runs were executed on a DEC Alpha workstation on loan from DEC. This machine uses the new Alpha RISC CPU running at 133 Mhz. With this machine, we were able to finish our tests in a fifth of the time it would have taken on other available machines. The machine had available 96 megabytes of memory. As we will see, it still wasn't enough for the polygonal program.

Each of the programs we created had several variants. The descriptions of each follows:

For the numerical approach, the initial implementation, **1d**, was done without the bounds. In this version, roots are located by taking a fixed number of sample points on each interval and looking for sign changes between two samples, or extrema in the direction of 0 involving three samples. Sign changes are fed to a hybrid bisection and secant root finder [29]. The points making up an extremum pointing towards 0 are input to a minimum finder that stops upon finding a root crossing. Then roots are located on either side of this point. This permits regions to be incorrectly classified, affecting the final answer. Good results can be obtained with a subdivision of 64 per interval.

In the other version of the numerical approach that we coded, referred to as **rad**, rad(t) was evaluated using the bounding method described in Chapter 4. As we will see, the running times often escalate dramatically. It is reasonable to assume that things will only become worse by using the bounds on the other equations, especially since they involve divisions that can give infinite values.

The numerical approach does all the work in the intersection routine. Almost no tool movement preprocessing is applicable as all the equations depend on individual line values.

For the three-axis approximation method, we implemented three versions. The first, **3x**, is the simple, straightforward approach. All submovements are generated and placed on a list during preprocessing. During intersection location, the line is intersected with each submovement in turn. The returned result is the minimum value found.

The second version, **3xtree**, implements the object hierarchy detailed earlier. The set of submovements is generated as above. Then they are paired off and wrapped with an enclosing three-axis movement. Each wrapping movement points to its enclosed submovements. Then the wrapping movements are paired off and the process is repeated recursively, building a complete tree. Intersection is performed by walking down the tree in a pseudo-binary search. At each node, the intersection is found between the line and the node's children. The closer child is explored first, pruning the second branch if the first branch generates an intersection outside the envelope of the second. Otherwise the second branch is followed as well. This is necessary due to the overlap of the envelopes.

The third variant, **3xtr1**, is the same as the second except that all values needed for each submovement's intersection are precomputed, if possible. Constants precomputed are ones that don't depend on the values of the line to be intersected.

The step method resulted in five different implementations. First the basic method as described was created, called **step**. The points on the line are tested one after the next, performing both tests, until an intersection with the cylinder is found, or the point is beyond the bounding box and no intersection can possibly take place.

The next version, **zstep**, adds in the z culling test. Each point test checks to see that the point's path overlaps the cylinder limits in the z direction before continuing with the rest of the test.

The third major variant, **fastep**, implements the large step approach. The method tries to double the step length, testing against an appropriately enlarged cylinder to guarantee accuracy, until an intersection is found. Then, using the largest non-intersecting step size, the algorithm takes a step and then tries to increase the step size again. This continues until the step size has been reduced to the unit length and an intersection is found, or until the end of the bounding box is reached.

All three programs included a refinement step in which the intersection location is improved by binary search between the found intersection point and the previous test point. This has the effect of greatly increasing the accuracy of intersection points when a hit is found, but miss results are only guaranteed to the accuracy specified. The remaining two programs are the z culling program, **zstepnr**, and the long step program, **fastepnr**, with the refinement step removed. These programs allow us to look at the costs of the method itself, separating out the near-constant cost of refinement.

The step programs have almost no preprocessing costs. A few values are set up beforehand but nothing computationally strenuous is done.

The polygonal approximation approach was implemented in two programs. The actual method as described made one program, named **poly**. A variant that serves as a lower bound was created as a second program called **poly nb**. Actual model size based on the surface representation presented must be at least as large as generated by this program. It provides no guarantees on accuracy but is probably closer to the needed level of subdivision for the representation developed here. In this version, subdivision occurred if the barycentric center of a triangle in parametric space is farther than the tolerance limit from the plane of the triangle. This is too loose, as the center point in real space may be far from the triangle boundaries while still being close to the plane. In addition, other points on the surface might lie too far away from the triangle plane. However, this program gives us a glimpse at potential performance for the approach, since the bounds as stated are highly conservative.

The polygonal approach is also heavily dependent on preprocessing. The actual intersection process is not nearly as intensive.

One other test program, **null**, is provided. This is simply the common test program without an intersection routine to allow subtracting testing overhead costs. It only depends on the number of tool movements and intersections to be performed.

7.1.2 Tests and Data

Testing had several goals. We wished to get comparisons on the relative performance of each method. We also wanted to verify that scaling of the problem set size is linear. We wished to look at the effects of changing tool movement shape. And finally, we desired to find out the effect of changing intersection tolerance on each method.

To accomplish this, we ran a test suite consisting of both randomly generated test cases and real cutting programs and surfaces supplied by the Ford Motor Company and automatic five-axis toolpath generation software [23] created by Xiaoxia Li at University of New Hampshire.

Our first group of tests looks at the performance of each program with changes in the number of tool movements processed and the number of intersections performed. For these tests, random tool movements and lines are generated by a test routine and fed to the intersection program. The tool size was set so the radius was less than 1/6th the length, which was set at 30 units. The rotation angle was randomly chosen but did not exceed 4 degrees, in order to approximate the type of tool movements found in the kinds of cutting programs in use today. The linear portion of the movements was allowed to vary up to 3 times the tool length. Intersection accuracy was set at .02 units.

Lines were chosen randomly by choosing a random direction vector and choosing a random base point lying in a bounding box surrounding the tool movement. This allowed us to get a reasonably large set of intersection hits as opposed to misses. Hits accounted for about 65% of all intersection tests performed.

With these parameters, we did a test run of the programs with 100 tool movements and 1000 tool movements. For 100 tool movements, we successively performed 10, 100, and 1000 random intersection tests per tool movement. For 1000 tool movements, 1 and 10 intersection tests per tool movement were carried out.

For each test run, we give the time spent preprocessing, time to find intersections, and total time spent in seconds. Total time spent was obtained from the UNIX **time** command. Intersection and preprocessing times were obtained from internal calls to **clock()**. The total time minus the time spent in the **NULL** program gives the time spent in preprocessing and intersection combined, for all data. The sum of preprocessing, intersection, and **NULL** program total is approximately the total time. The data for the first tests are in tables 7.1 and 7.2. Times around .5 second and less are not very reliable.

# int tests		10			100			1000	
Program	pre	int	tot	pre	int	tot	pre	int	tot
NULL	0.0	0.0	0.4	0.0	0.2	0.2	0.0	1.4	1.5
1d	0.0	2.0	2.3	0.0	25.3	25.5	0.0	257.8	259.2
rad	0.0	37.5	37.9	0.0	533.4	533.6	0.0	4360.7	4362.2
step	0.0	16.4	16.8	0.0	181.0	181.2	0.0	1936.1	1936.2
zstep	0.0	14.5	14.9	0.0	157.7	157.9	0.0	1569.6	1569.8
fastep	0.0	13.4	13.7	0.0	144.2	144.9	0.0	1552.8	1552.9
zstepnr	0.0	2.5	2.9	0.0	32.0	32.2	0.0	317.0	317.1
fastepnr	0.0	2.2	2.6	0.0	27.2	27.5	0.0	285.3	285.4
3x	0.0	0.1	0.5	0.0	4.1	4.2	0.0	38.1	39.6
3xtree	0.1	0.2	0.3	0.1	1.7	1.8	0.1	15.5	17.1
3xtr1	0.3	0.2	0.5	0.3	2.1	2.7	0.2	16.9	18.4
poly	224.5	2.4	227.3	237.1	23.4	260.7	209.9	223.9	433.1
poly nb	13.7	1.4	15.4	14.2	15.1	29.4	10.2	142.9	154.5

Table 7.1: 100 Tool Movements

The second group of tests looked at variation in the tool movement parameters. We first looked at the rotation angle and then the length of linear movement. To investigate rotation angle effects, we tested the programs on tool movements with maximum rotations of 3, 6, 9, 15, 30, 60, 90, 120, 150, and 180 degrees. We did this for 100 tool movements and 10 and 100 intersection tests per tool movement. These results are in tables 7.3 and 7.4.

# int tests		1			10	
Program	pre	int	tot	pre	int	tot
NULL	0.0	0.1	0.2	0.0	0.3	0.3
1d	0.0	2.4	2.7	0.0	24.4	25.0
rad	0.1	58.6	58.9	0.0	216.4	217.0
step	0.1	17.4	17.8	0.0	172.5	172.8
zstep	0.1	15.6	16.0	0.2	142.4	142.9
fastep	0.2	14.1	14.7	0.1	141.0	141.4
zstepnr	0.1	3.0	3.3	0.1	30.0	30.2
fastepnr	0.2	2.7	3.0	0.2	27.1	27.6
3x	0.3	0.5	0.9	0.3	3.8	4.4
3xtree	0.8	0.1	1.0	0.6	1.7	2.5
3xtrl	2.3	0.3	2.8	2.1	2.1	4.5
poly	2185.9	2.5	2188.6	2190.9	25.2	2216.1
poly nb	25.1	2.0	27.6	24.5	22.0	47.6

Table 7.2: 1000 Tool Movements

To test the reaction to change in the linear portion of tool movements, we ran tests with the linear distance limited to 3, 2, 1, .5, .25, .125, and .0625 times the tool length, which was still set to 30. These tests were done with rotation angle maximums of 3 and 90 degrees for 100 tool movements and 10 and 100 intersection tests per tool movement. These results are in tables 7.5, 7.6, 7.7, and 7.8. The program **3xtrl** was not tested since it became clear in the first test set that it was always inferior to **3xtree**. The **poly** program was not tested because of the memory problems encountered with large angle tool movements. The only step programs tested were **zstepnr** and **fastepnr** since the refinement just adds constant costs and the **step** program is always slower than **zstep**.

Next, we looked at the effects of changes in accuracy on each of the approaches. We did tests with varying accuracies of .2, .02, and .002. The programs **1d** and **rad** don't have adjustable tolerances, so a single running time is given. These tests were done for 100 tool movements with 10 and 100 intersections per tool movement. These

	3xtree			3x			fastepnr			zstepnr		
Angle	pre	int	tot	pre	int	tot	pre	int	tot	pre	int	tot
3	0.0	0.1	0.2	0.0	0.3	0.3	0.0	2.3	2.4	0.0	2.1	2.1
6	0.1	0.1	0.3	0.0	0.6	0.6	0.0	2.8	2.9	0.0	3.5	3.7
9	0.1	0.2	0.3	0.1	0.8	0.9	0.0	3.0	3.2	0.0	5.3	5.4
15	0.2	0.3	0.5	0.0	1.4	1.5	0.0	4.0	4.2	0.0	9.5	9.7
30	0.4	0.4	0.8	0.2	2.6	2.9	0.0	5.3	5.5	0.0	22.0	22.2
60	0.8	0.5	1.3	0.4	5.3	5.7	0.0	6.5	6.7	0.0	32.3	32.5
90	0.9	0.7	1.7	0.6	7.7	8.4	0.0	7.3	7.5	0.0	35.9	36.0
120	1.5	0.8	2.5	0.7	10.6	11.4	0.0	6.9	7.0	0.1	36.5	36.7
150	1.7	0.9	2.6	0.8	13.2	14.1	0.0	7.4	7.5	0.0	36.6	36.6
180	2.1	0.9	3.1	1.2	15.8	17.2	0.0	7.2	7.3	0.0	35.2	35.3

	poly nb			1d			rad		
Angle	pre	int	tot	pre	int	tot	pre	int	tot
3	8.3	1.5	10.0	0.0	3.1	3.2	0.0	1213.0	1212.1
6	8.4	1.6	10.0	0.0	3.2	3.2	0.0	1261.2	1260.7
9	8.6	1.4	10.8	0.0	3.2	3.3	0.0	1279.8	1278.7
15	8.7	1.5	10.2	0.0	3.3	3.3	0.0	1478.0	1472.0
30	8.9	1.6	10.5	0.0	3.4	3.4	0.0	1265.5	1263.4
60	12.2	1.9	14.4	0.0	3.5	3.6	0.0	1365.8	1365.4
90	17.6	2.3	19.8	0.0	3.5	3.6	0.0	1389.4	1389.4
120	23.2	2.6	25.5	0.0	3.7	3.8	0.0	1378.5	1377.1
150	26.9	2.8	29.5	0.0	3.9	4.0	0.0	1304.2	1304.6
180	37.0	2.5	39.7	0.0	4.1	4.2	0.0	1201.1	1201.4

Table 7.3: Rotation Data 100 Tool Movements, 10 Intersections Per

	3xtree			3x			fastepnr			zstepnr		
Angle	pre	int	tot	pre	int	tot	pre	int	tot	pre	int	tot
3	0.0	1.4	1.5	0.0	2.9	3.0	0.0	27.1	27.3	0.0	24.3	24.5
6	0.1	1.8	2.0	0.0	5.8	5.9	0.0	33.5	33.7	0.0	46.0	46.2
9	0.0	2.2	2.3	0.0	8.4	8.5	0.0	39.0	39.2	0.0	71.9	72.1
15	0.3	2.7	3.5	0.2	13.8	14.6	0.0	47.9	48.1	0.0	125.2	125.5
30	0.4	4.1	4.5	0.2	27.6	28.0	0.0	54.4	54.6	0.0	214.5	214.8
60	0.9	5.9	6.8	0.4	55.1	55.5	0.0	62.6	62.9	0.0	313.7	313.9
90	1.1	7.5	8.7	0.6	82.0	82.6	0.0	66.1	66.3	0.0	343.6	343.9
120	1.5	9.2	10.9	0.8	108.8	109.5	0.0	71.8	72.1	0.0	353.2	353.4
150	1.7	9.8	11.6	1.0	136.6	137.6	0.1	75.2	75.4	0.0	358.1	358.3
180	2.5	11.2	13.7	1.2	164.8	166.0	0.0	77.5	77.7	0.0	367.0	367.2

	poly nb			1d			rad		
Angle	pre	int	tot	pre	int	tot	pre	int	tot
3	9.0	14.6	23.8	0.0	32.3	32.5	0.0	37801.9	37802.1
6	9.1	14.9	24.2	0.0	33.0	33.2	0.0	25420.6	25420.8
9	9.2	15.3	24.5	0.0	33.6	33.8	0.0	21668.0	21668.2
15	9.3	15.3	24.7	0.0	33.2	33.4	0.0	17885.5	17885.6
30	9.9	16.5	26.6	0.0	33.2	33.4	0.0	14740.0	14740.2
60	13.3	19.8	33.4	0.0	34.6	34.8	0.0	13285.8	13285.9
90	20.6	21.6	42.7	0.0	36.2	36.4	0.0	13633.1	13633.3
120	30.9	24.4	55.8	0.0	38.0	38.2	0.0	13467.3	13467.6
150	40.4	27.0	67.7	0.0	40.3	40.5	0.0	13693.0	13693.2
180	43.5	30.1	73.8	0.0	43.1	43.3	0.0	14765.2	14760.3

Table 7.4: Rotation Data 100 Tool Movements, 100 Intersections Per

length	3xtree			3x			fastepnr			zstepnr		
(k*L)	pre	int	tot	pre	int	tot	pre	int	tot	pre	int	tot
.0625	0.0	0.2	0.2	0.0	0.3	0.3	0.0	0.6	0.6	0.0	0.5	0.6
.125	0.1	0.1	0.2	0.0	0.3	0.3	0.0	0.7	0.7	0.0	0.7	0.7
.25	0.0	0.1	0.2	0.0	0.3	0.3	0.0	0.8	0.8	0.0	0.8	0.8
.5	0.0	0.2	0.2	0.0	0.3	0.3	0.0	1.0	1.1	0.0	0.9	0.9
1	0.0	0.1	0.2	0.0	0.2	0.3	0.0	1.5	1.5	0.0	1.2	1.3
2	0.1	0.1	0.2	0.0	0.3	0.3	0.0	2.1	2.7	0.0	1.8	1.8
3	0.0	0.1	0.2	0.0	0.3	0.3	0.0	2.3	2.4	0.0	2.1	2.1

Length	poly nb			1d			rad		
(k*L)	pre	int	tot	pre	int	tot	pre	int	tot
.0625	10.9	1.2	12.4	0.0	1.3	1.4	0.0	118.1	118.2
.125	10.6	1.2	12.0	0.0	1.4	1.5	0.0	172.1	172.1
.25	10.2	1.3	11.6	0.0	1.5	1.6	0.0	303.2	303.4
.5	9.7	1.3	11.1	0.0	1.9	1.9	0.0	458.5	458.7
1	9.1	1.3	10.5	0.0	2.3	2.4	0.0	795.5	795.7
2	8.5	1.3	10.0	0.0	2.8	2.9	0.0	1039.2	1039.4
3	9.3	1.5	10.9	0.0	3.1	3.2	0.0	1211.9	1212.1

Table 7.5: Linear Data 100 Tool Movements 10 Intersections Per, Max 3 Degrees

Length	3xtree			3x			fastepnr			zstepnr		
(k*L)	pre	int	tot	pre	int	tot	pre	int	tot	pre	int	tot
.0625	0.1	1.3	1.5	0.0	3.0	3.8	0.0	6.6	6.7	0.0	5.3	5.4
.125	0.1	1.3	1.5	0.0	3.0	3.8	0.0	7.8	7.9	0.0	6.4	6.5
.25	0.0	1.4	1.4	0.0	3.0	3.8	0.0	10.3	10.4	0.0	9.2	9.9
.5	0.1	1.3	1.4	0.0	3.0	3.6	0.0	13.6	13.8	0.0	12.3	12.5
1	0.0	1.4	1.5	0.0	2.9	3.3	0.0	18.1	18.3	0.0	16.6	16.5
2	0.1	1.3	1.5	0.0	2.9	3.0	0.0	23.1	23.2	0.0	21.2	21.2
3	0.0	1.4	1.5	0.0	2.9	3.0	0.0	27.2	27.3	0.0	24.5	24.5

Length	poly nb			1d			rad		
(k*L)	pre	int	tot	pre	int	tot	pre	int	tot
.0625	10.5	14.8	25.2	0.0	15.3	15.4	0.0	1088.0	1088.1
.125	10.3	14.6	24.9	0.0	16.2	16.3	0.0	1816.0	1816.2
.25	10.4	14.3	24.8	0.0	18.2	18.2	0.0	3009.3	3009.5
.5	10.0	14.3	24.4	0.0	21.0	21.8	0.0	5020.0	5020.2
1	9.6	14.6	24.3	0.0	25.2	25.1	0.0	9084.8	9085.0
2	9.4	15.1	24.5	0.0	29.9	29.8	0.0	16706.2	16706.4
3	9.0	14.5	23.5	0.0	32.6	32.5	0.0	27802.0	27802.1

Table 7.6: Linear Data 100 Tool Movements 100 Intersections Per, Max 3 Degrees

Length	3xtree			3x			fastepnr			zstepnr		
(k*L)	pre	int	tot	pre	int	tot	pre	int	tot	pre	int	tot
.0625	1.0	0.9	2.7	0.6	8.1	8.7	0.0	6.2	6.3	0.0	6.8	6.9
.125	1.0	1.0	2.6	0.4	8.1	8.7	0.0	6.2	6.3	0.0	8.3	8.3
.25	1.0	1.0	2.7	0.6	8.1	8.7	0.0	6.3	6.3	0.0	11.1	11.2
.5	1.0	1.0	2.6	0.4	8.1	8.7	0.0	6.5	6.6	0.0	16.2	16.4
1	1.1	0.8	1.9	0.5	8.0	8.6	0.0	6.5	6.6	0.0	22.4	22.5
2	1.0	0.8	1.8	0.5	8.0	8.5	0.0	6.7	6.8	0.0	30.3	30.5
3	0.9	0.7	1.7	0.6	7.7	8.4	0.0	7.4	7.5	0.0	35.9	36.0

Length	poly nb			1d			rad		
(k*L)	pre	int	tot	pre	int	tot	pre	int	tot
.0625	105.1	2.3	96.4	0.0	2.8	2.8	0.0	985.4	985.6
.125	62.5	2.1	61.3	0.0	2.8	2.8	0.0	1000.0	1000.2
.25	47.6	2.2	46.5	0.0	2.9	2.9	0.0	1068.4	1068.6
.5	29.4	2.3	31.1	0.0	2.9	3.0	0.0	1093.9	1094.1
1	24.3	2.3	26.2	0.0	3.0	3.1	0.0	1115.5	1115.7
2	19.5	2.4	21.6	0.0	3.4	3.5	0.0	1303.9	1304.1
3	17.6	2.3	19.8	0.0	3.5	3.6	0.0	1389.2	1389.4

Table 7.7: Linear Data 100 Tool Movements 10 Intersections Per, Max 90 Degrees

Length	3xtree			3x			fastepnr			zstepnr		
(k*L)	pre	int	tot	pre	int	tot	pre	int	tot	pre	int	tot
.0625	1.0	10.4	11.5	0.6	84.1	84.8	0.0	63.1	63.2	0.0	74.6	74.9
.125	1.2	10.2	11.4	0.6	84.1	84.8	0.0	63.3	63.4	0.0	89.5	89.7
.25	1.2	9.7	11.3	0.6	83.6	84.3	0.0	64.3	64.3	0.0	119.5	119.6
.5	1.2	9.8	11.4	0.5	83.4	83.9	0.0	64.7	64.8	0.0	163.6	163.8
1	1.2	9.1	10.4	0.5	83.6	84.5	0.0	65.5	65.5	0.0	221.3	221.5
2	1.1	8.2	9.4	0.7	81.2	81.8	0.0	71.1	71.3	0.0	294.8	295.0
3	1.0	7.6	8.7	0.6	82.0	82.6	0.0	77.8	77.9	0.0	343.6	343.9

Length	poly nb			1d			rad		
(k*L)	pre	int	tot	pre	int	tot	pre	int	tot
.0625	99.4	19.1	118.8	0.0	27.9	28.0	0.0	8948.4	8948.6
.125	69.8	19.2	89.2	0.0	28.0	28.2	0.0	8918.5	8918.7
.25	53.4	19.6	73.3	0.0	28.5	28.7	0.0	8868.1	8868.3
.5	37.1	20.4	57.7	0.0	29.2	29.3	0.0	9309.7	9309.9
1	29.7	20.3	50.2	0.0	31.4	31.6	0.0	10440.1	10440.3
2	22.7	21.1	44.1	0.0	34.3	34.5	0.0	11777.8	11778.0
3	20.6	21.6	42.4	0.0	36.5	36.6	0.0	13633.1	13633.3

Table 7.8: Linear Data 100 Tool Movements 100 Intersections Per, Max 90 Degrees

results are given in tables 7.9 and 7.10.

Tolerance	.2			.02			.002		
	pre	int	tot	pre	int	tot	pre	int	tot
NULL				0.0	0.0	0.4			
1d				0.0	1.9	2.3			
rad				0.0	37.5	37.9			
step	0.0	13.0	13.4	0.0	16.4	16.8	0.0	53.8	54.2
zstep	0.0	11.0	11.4	0.0	14.5	14.9	0.0	35.4	35.8
fastep	0.0	12.7	12.8	0.0	13.3	13.7	0.0	15.8	16.1
zstepnr	0.0	0.1	0.5	0.0	2.5	2.9	0.0	26.1	26.5
fastepnr	0.0	1.0	1.5	0.0	2.2	2.6	0.0	4.5	5.0
3x	0.0	0.3	0.9	0.0	0.1	0.5	0.0	4.0	4.3
3xtree	0.0	0.1	0.1	0.1	0.2	0.3	0.5	0.3	0.9
3xtrl	0.0	0.1	0.1	0.3	0.2	0.5	2.2	0.4	2.9
poly	79.5	2.0	81.7	224.9	2.1	227.3	Out of Memory		
poly nb	2.3	1.3	3.8	13.7	1.4	15.4	157.0	1.8	158.9

Table 7.9: Tolerance Tests, 100 Tool Movements, 10 Intersections Per

Tolerance	.2			.02			.002		
	pre	int	tot	pre	int	tot	pre	int	tot
null				0.0	0.0	0.4			
1d				0.0	25.4	25.5			
rad				0.0	533.6	533.6			
step	0.0	136.0	136.2	0.0	181.0	181.2	0.0	601.0	601.0
zstep	0.0	118.4	118.8	0.0	157.8	157.9	0.0	399.3	399.8
fastep	0.0	128.7	128.9	0.0	144.2	144.9	0.0	160.0	160.9
zstepnr	0.0	4.6	4.6	0.0	32.0	32.2	0.0	299.2	299.7
fastepnr	0.0	11.1	11.2	0.0	27.2	27.5	0.0	52.6	52.6
3x	0.0	0.7	0.7	0.0	4.1	4.2	0.4	42.3	42.8
3xtree	0.0	0.8	0.8	0.1	1.7	1.8	0.5	3.2	3.9
3xtrl	0.1	0.8	0.9	0.3	1.8	2.7	2.5	4.9	7.5
poly	82.3	22.1	104.5	237.1	23.4	260.7	Out of Memory		
poly nb	2.4	14.3	16.8	14.4	15.1	29.4	171.2	18.3	189.6

Table 7.10: Tolerance Tests, 100 Tool Movements, 100 Intersections Per

Finally, tests were run on actual cutting tool data and surface point sets generated by NC, a software testbed created at UNH and Dartmouth College to permit testing of new approaches to toolpath generation and verification. The density of the point sets for each file is controlled by simulation accuracy. We generated some of the point sets at two different simulation accuracies to get varying surface point densities for a given cutting program. This has no effect on intersection tolerances.

In the real data tests, a bucketing scheme employing the short-vector localization method mentioned in [22] was used to limit the number of intersections performed. As a result, the number of intersection tests performed is not equal to the product of the number of tool movements and surface points. A 2-D grid is placed over the surface to be cut, and all points were placed in the corresponding cell. When intersection is performed, a bounding box for the tool movement is projected onto the grid plane and all cells overlapping it are selected. Then all points in those cells only are cut. The bounding box dimensions are expanded in each direction by the short normal length before cell selection. This is because points in nearby unselected cells may still be cut by the tool movement if their normals extend into the regions selected.

We had several real surface files and cutting programs on which to test the programs. Zip5 is a benchmark surface generated for verification testing. Door is the exterior skin of a car door. Door_small uses the same surface file, but tests only a small portion of the cutting program, thereby machining a piece of the surface. Saddle represents a hyperbolic saddle point. And z6324r is an automotive quarter panel. The surface files were all supplied by the Ford Motor Company along with the five-axis cutting program for door. The remainder of the cutting programs were generated at UNH.

The following tests were done: zip5 with surface points generated at default simulation accuracy (4.75mm), zip5 with surface points generated at 1mm simulation accuracy, door with surface points generated at default accuracy, saddle with points generated at 1.5mm simulation accuracy, door_small with surface points generated at default accuracy, door_small with surface points generated at 3mm simulation accuracy, z6324r with points generated at default accuracy, and z6324r with surface generated at 1.5mm accuracy. This provides some comparison data on real world problems. The intersections were found with a tolerance of .2mm. The data are in tables 7.11, 7.12, 7.13, 7.14, 7.15, 7.16, 7.17, and 7.18 respectively.

Then we tested some of the files with varying intersection tolerances to look further at the behavior of the various programs with tolerance changes. These files are door_small at default and 3mm simulation accuracy, and z6324r at default and 1.5mm simulation accuracy. These programs were run with tolerance values of .2, .02, and .002. Then, **fastepnr**, **3xtree**, and **poly nb** were also run at .5, .05, and .005 to get a better view of the curves they generated.

The numerical methods programs, **1d** and **rad**, have no accuracy input, so were only run once. The only step programs run were **zstepnr** and **fastepnr** for the reasons outlined in the second test. The **poly** program was limited by memory so we ran it at 2.0mm instead of .002mm tolerance. We ran the **poly nb** program at 2.0mm as well for comparison purposes. **poly** was unable to run at all on z6324r. **poly nb** ran out of memory as well on z6324r at tolerances of .005 and .002. Finally, **3x** was run at 2.0 instead of .002 on z6324r at 1.5mm since it was clear from theory and the previous tests that linear scaling would occur and the time required would be enormous.

Relative speed comparisons are shown in tables 7.19 and 7.20 in terms of intersections tests performed per second, and number of tool movements preprocessed per second. The data shown is for real tests only. Preprocessing tables only contain 3x, 3xtree, poly, and poly nb, because the others have only constant preprocessing costs.

File			zip5
Simulation accuracy			4.75mm (default)
Intersection accuracy			.2
Surface points			290
Tool movements			2153
Intersections performed			83445
Avg. # hits found			56243
	pre	int	tot
null	0.0	0.8	2.4
1d	0.0	274.9	277.3
rad	0.2	16508.7	16506.0
zstepnr	0.1	26.3	28.8
fastepnr	0.2	60.0	62.6
3x	0.1	4.3	6.8
3xtree	0.1	4.7	7.2
poly		Out of Memory	
poly nb	0.2	5.6	7.2

Table 7.11: Data for file zip5, default simulation accuracy

File			zip5
Simulation accuracy			1mm
Intersection Accuracy			.2
Surface points			3246
Tool Movements			2153
Intersections performed			972765
Avg. # hits found			641805
	pre	int	tot
null	0.0	1.9	4.8
1-d	0.1	2969.1	2973.0
1-d rad	0.1	110765.4	115011.6
step z cull nr	0.3	254.4	259.5
fast step nr	0.2	521.0	526.7
3x	0.2	51.4	56.4
3xtree	0.3	52.1	57.3
poly		Out of Memory	
poly nb	0.2	55.1	58.5

Table 7.12: Data for file zip5, 1mm simulation accuracy

File			door
Simulation accuracy			4.75mm (default)
Intersection Accuracy			.2
Surface points			3630
Tool Movements			38272
Intersections performed			500433
Avg. # hits found			358144
	pre	int	tot
null	0.5	1.6	25.8
1d	1.2	1672.3	1707.7
rad	1.7	5260.1	5295.3
zstepnr	2.0	857.1	885.3
fastepnr	1.8	726.9	758.9
3x	2.4	207.4	235.6
3xtree	3.6	55.5	84.5
poly	8136.8	535.9	8700.8
poly nb	291.4	545.0	868.2

Table 7.13: Data for file door, default simulation accuracy

File			saddle
Simulation accuracy			1.5mm
Intersection Accuracy			.2
Surface points			2320
Tool Movements			3226
Intersections performed			2843740
Avg. # hits found			1758473
	pre	int	tot
null	0.0	6.1	9.0
1d	0.3	6782.2	6783.5
rad	0.2	5091.5	5094.9
zstepnr	0.4	1193.2	1202.8
fastepnr	0.4	2313.3	2322.7
3x	0.6	620.1	630.2
3xtree	1.1	666.1	676.4
poly	Out of Memory		
poly nb	323.5	3949.1	4281.6

Table 7.14: Data for file saddle, 1.5mm simulation accuracy

File	door_small
Simulation accuracy	4.75mm (default)
Surface points	2674
Tool Movements	493
Intersections performed	6004
Avg. # hits found	4758

	null			1d			rad		
	pre	int	tot	pre	int	tot	pre	int	tot
	0.0	0.0	1.1	0.0	17.6	18.7	0.0	23.6	24.9

	zstepnr			3x			poly		
	pre	int	tot	pre	int	tot	pre	int	tot
2							47.9	2.2	51.7
.2	0.0	4.2	5.3	0.0	1.8	3.0	80.3	4.7	85.6
.02	0.0	34.9	36.2	0.2	18.3	19.6	616.2	35.3	603.9
.002	0.0	344.0	344.9	1.4	185.7	187.9			

	fastepnr			3xtree			poly nb		
	pre	int	tot	pre	int	tot	pre	int	tot
.5	0.0	3.6	4.7	0.0	0.4	1.5	1.3	3.8	6.3
.2	0.0	4.7	5.9	0.1	0.6	1.8	3.0	5.0	9.2
.05	0.0	7.6	8.8	0.2	1.1	2.4	10.6	14.8	26.6
.02	0.0	10.2	11.5	0.3	1.5	3.0	25.8	34.8	61.4
.005	0.1	13.8	15.4	1.1	2.3	4.5	121.4	135.1	256.0
.002	0.0	16.9	18.2	2.3	2.8	6.2	282.0	336.0	614.3

Table 7.15: Data for file door_small, default simulation accuracy

File	door_small
Simulation accuracy	3mm
Surface points	14754
Tool Movements	493
Intersections performed	31287
Avg. # hits found	23090

	null			1d			rad		
	pre	int	tot	pre	int	tot	pre	int	tot
	0.0	0.0	4.9	0.0	87.8	92.7	0.0	79.2	84.2

	zstepnr			3x			poly		
	pre	int	tot	pre	int	tot	pre	int	tot
2							47.9	12.2	64.6
.2	0.0	27.6	32.4	0.0	8.9	13.6	71.4	15.5	92.1
.02	0.0	246.0	250.9	0.1	91.8	96.7	615.5	179.0	751.3
.002	0.0	2437.7	2442.7	1.5	921.3	926.9			

	fastepnr			3xtree			poly nb		
	pre	int	tot	pre	int	tot	pre	int	tot
.5	0.0	22.2	27.2	0.1	1.8	6.7	1.2	19.7	25.6
.2	0.0	30.7	35.6	0.0	3.1	7.9	2.7	14.6	22.2
.05	0.0	47.3	52.3	0.2	5.6	10.5	10.5	76.0	91.1
.02	0.0	60.2	65.8	0.2	7.5	12.4	25.8	175.8	206.0
.005	0.0	78.6	83.6	1.1	10.7	16.5	121.5	681.0	805.3
.002	0.0	94.5	99.7	2.4	13.4	20.4	283.4	1691.7	1973.6

Table 7.16: Data for file door_small, 3mm simulation accuracy

File	z6324r
Simulation accuracy	default
Surface points	2675
Tool Movements	1458
Intersections performed	290564
Avg. # hits found	171853

	null			1d			rad		
	pre	int	tot	pre	int	tot	pre	int	tot
	0.1	1.2	2.3	0.2	739.6	741.4	0.2	765.4	767.4

	zstepnr			3x			poly		
	pre	int	tot	pre	int	tot	pre	int	tot
.2	0.2	71.0	73.5	0.2	32.4	34.9	Out of Memory		
.02	0.1	502.2	504.6	0.9	295.7	297.9	Out of Memory		
.002	0.1	4755.3	4757.7	7.5	3166.8	3176.7	Out of Memory		

	fastepnr			3xtree			poly nb		
	pre	int	tot	pre	int	tot	pre	int	tot
.5	0.2	108.2	110.7	0.2	17.5	19.0	39.9	639.6	681.7
.2	0.2	141.4	143.9	0.4	26.1	28.9	101.1	563.4	666.8
.05	0.2	213.1	215.4	0.7	50.8	53.8	297.4	409.2	708.9
.02	0.2	264.8	267.3	1.5	75.4	79.2	922.6	774.2	1699.1
.005	0.1	347.4	349.8	5.6	126.3	134.7	Out of Memory		
.002	0.2	415.7	418.3	13.6	173.1	189.0	Out of Memory		

Table 7.17: Data for file z6324r, default simulation accuracy

File	z6324r
Simulation accuracy	1.5mm
Surface points	2675
Tool Movements	1458
Intersections performed	1407907
Avg. # hits found	846840

	null			1d			rad		
	pre	int	tot	pre	int	tot	pre	int	tot
	0.0	2.6	4.4	0.1	3607.5	3608.5	0.2	3606.2	3608.6

	zstepnr			3x			poly		
	pre	int	tot	pre	int	tot	pre	int	tot
2				0.2	39.1	43.7			
.2	0.2	373.0	377.5	0.2	161.7	166.3	Out of Memory		
.02	0.2	2824.8	2829.4	0.6	1587.5	1592.5	Out of Memory		
.002	0.1	26010.2	26014.8				Out of Memory		

	fastepnr			3xtree			poly nb		
	pre	int	tot	pre	int	tot	pre	int	tot
2							28.8	2983.1	3016.3
.5	0.2	553.9	558.5	0.3	82.9	87.6	39.8	3064.5	3108.7
.2	0.2	727.7	732.3	0.4	133.2	138.6	100.4	2713.5	2818.3
.05	0.1	1098.7	1103.2	0.7	265.6	270.7	294.3	1962.6	2257.6
.02	0.2	1376.4	1381.0	1.7	394.6	400.7	901.9	3787.8	4694.1
.005	0.2	1805.6	1810.2	5.8	672.5	680.8			
.002	0.1	2158.8	2163.3	13.7	918.9	936.9			

Table 7.18: Data for file z6324r, 1.5mm simulation accuracy

Intersection Tests Per Second at Varying Accuracy

File	Program	1d	rad	zstepnr	fastepnr	3x	3xtree	poly	poly nb
.2 Accuracy									
zip5	default	304	5	3173	1391	19406	17754	—	14901
	1mm	328	9	3824	1867	18925	18671	—	17655
door		300	95	583	688	2412	9016	933	918
saddle		419	558	2383	1229	8884	4269	—	720
door_small	default	341	254	1429	1668	3336	10007	1277	1201
	3mm	356	395	1134	1409	3515	10092	2019	2143
z6324r	default	393	379	4092	2054	8968	11133	—	516
	1.5mm	390	390	3774	1935	8707	10570	—	519
.02 Accuracy									
door_small	default	304	5	172	589	328	4003	170	173
	3mm	328	9	127	520	341	4172	175	178
z6324r	default	300	95	498	1023	887	3568	—	372
	1.5mm	419	558	579	1097	982	3854	—	375
.002 Accuracy									
door_small	default	341	254	17	355	32	2144	—	—
	3mm	356	395	13	331	33	2335	—	—
z6324r	default	393	379	54	652	—	1532	—	—
	1.5mm	390	390	61	698	91	1679	—	—

Table 7.19: Relative intersection speed comparison charts

Preprocessing Tool Movements Per Second

	Program	3x	3xtree	poly	polynb
.2 Accuracy					
door		15947	10631	5	131
saddle		5377	2933	—	10
door_small	default	> 4930	4930	6	164
	3mm	> 4930	> 4930	7	183
z6324r	default	7290	3645	—	14
	1.5mm	7290	3645	—	15
.02 Accuracy					
door_small	default	2465	1643	1	19
	3mm	4930	2465	1	19
z6324r	default	1620	972	—	2
	1.5mm	2430	858	—	2
.002 Accuracy					
door_small	default	352	214	—	.2
	3mm	329	205	—	2
z6324r	default	194	107	—	—
	1.5mm	—	106	—	—

Table 7.20: Relative preprocessing speed comparison chart

7.2 Evaluation

7.2.1 Scaling

The results from the first test set in tables 7.1 and 7.2 show basically what we would expect: the running times increase linearly with the number of intersections performed and the number of tool movements used. In addition, we can see that the preprocessing times and intersection times separately increase linearly. Although it may appear that there is a downward trend in the times of **3xtree** and **3xtrl**, this comes about from randomness in the data and minor time variations that occur in different runs of the same program. The following table contains data from runs done on a DECstation 5000/200.

100 tool movements

# int tests	10			100		
	pre	int	tot	pre	int	tot
3xtree	0.1	1.1	1.3	0.2	11.7	11.8
3xtrl	0.7	1.2	2.1	0.8	12.9	13.7

1000 tool movements

# int tests	10			100		
	pre	int	tot	pre	int	tot
3xtree	1.4	1.0	3.4	1.3	10.5	13.2
3xtrl	7.4	1.2	9.9	7.3	12.2	21.2

From this data, it is pretty clear that these programs scale linearly with problem size.

Another odd feature is that the program 3xtrl, which precomputes some of the intersection values for each submovement in the tree, is actually slower than the program without precomputing these values. One side effect of computing all these values beforehand is that a much larger data structure is needed to store them all. This larger structure is in fact 3 times as large. We looked at the effect of memory traffic by running a version of 3xtree compiled with the large data structure but otherwise unchanged. On a DECstation 5000/200, we have the following:

100 tool movements						
# int tests	100			1000		
	pre	int	tot	pre	int	tot
3xtree	0.1	9.5	9.7	0.1	95.2	95.3
3xtree large	0.1	10.3	10.4	0.1	103.6	103.7
3xtrl	0.8	10.4	11.3	0.7	102.8	103.5

Two things are obvious from this. First, the larger structure slows down the intersection process. Most of this occurs in the tree descent. Second, the savings in actual intersection computation are not very large, and offset by the costs of implementation.

The next thing we can get from the first set of data is a look at the amount of time each method spends in preprocessing versus time spent in intersection calculations. For instance, as we stated above, the 1d programs and the step programs spend almost no time preprocessing. The polygonal programs spend a majority of their time and effort in preprocessing. What is very interesting is the lack of time spent in preprocessing in the 3-axis programs. They have to set things up but the amount of actual work done is relatively small.

The remaining conclusions to draw from the first test set are relative comparisons. We start to get a look at the comparative speeds of the methods, both between variants and between different approaches.

From this it was obvious that the basic step program is slower than the z culling variant. This also makes sense since the z culling program performs the same point tests, but can often make them quicker. The only way the basic step could be faster is if the z cull test failed so often that the time to perform the test (very minimal) outweighed the savings gained when it did succeed.

In addition, we get a look at the constant costs of the refinement once an intersection is located. Removing the refinement gives us a better look at the performance of the intersection routine itself while not reducing the guaranteed accuracy of the approach. Therefore we decided to restrict later testing to the z culling program and fastep program without refinement.

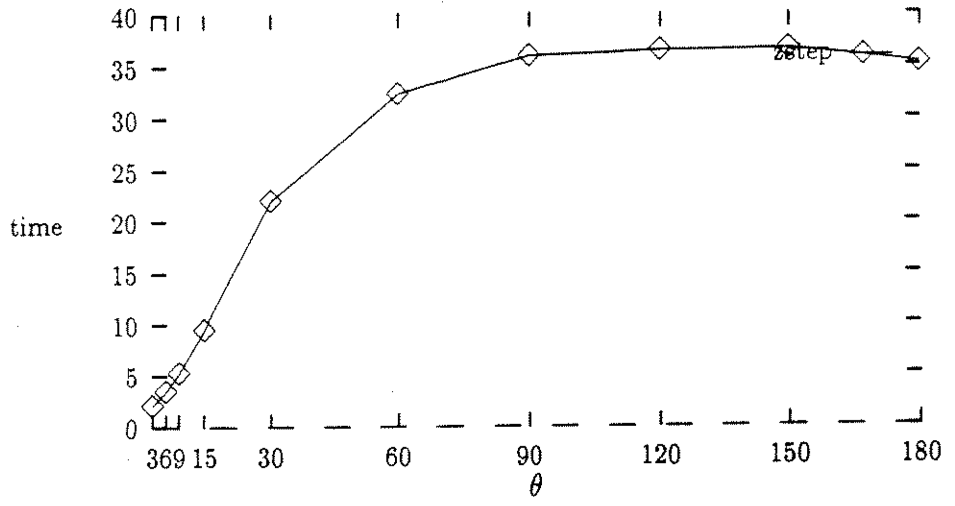
7.2.2 Rotation Angle

The rotation angle tests take a look the performance of each method with changes in rotation angle. Although the real data tends to have very limited rotations due to the difficulty of programming five-axis NC machines, they have the potential for very wide angular changes. Let us look at the data method by method.

step

We can see several things about the performance of the step programs from this data.

First, for small enough angles (and at this accuracy level), the zstep program is actually faster than the fastep program. This can be attributed to the fact that, in this case, the linear movement is still long and that as a result, the tool movement is pretty flat. That means that the three-axis bounding box is a close approximation to the actual envelope, and the starting point on the line is quite near the intersection in general. Therefore, there isn't enough distance or, subsequently, point tests to amortize the overhead associated with the fastep algorithm. As the angle increases, the three-axis envelope becomes a less adequate fit, resulting in longer average distances between the first test point and the intersection. In these cases, the fastep algorithm has enough time to amortize its startup costs and begin to be effective.



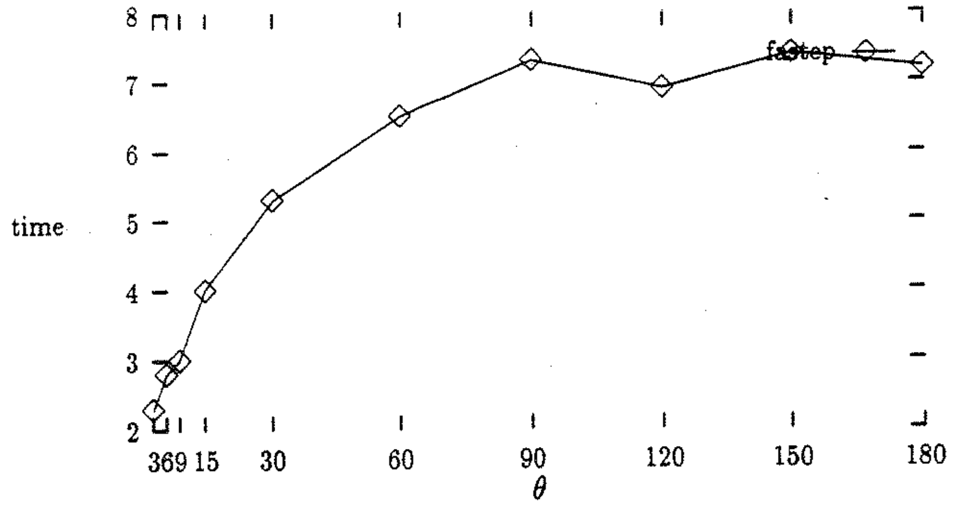
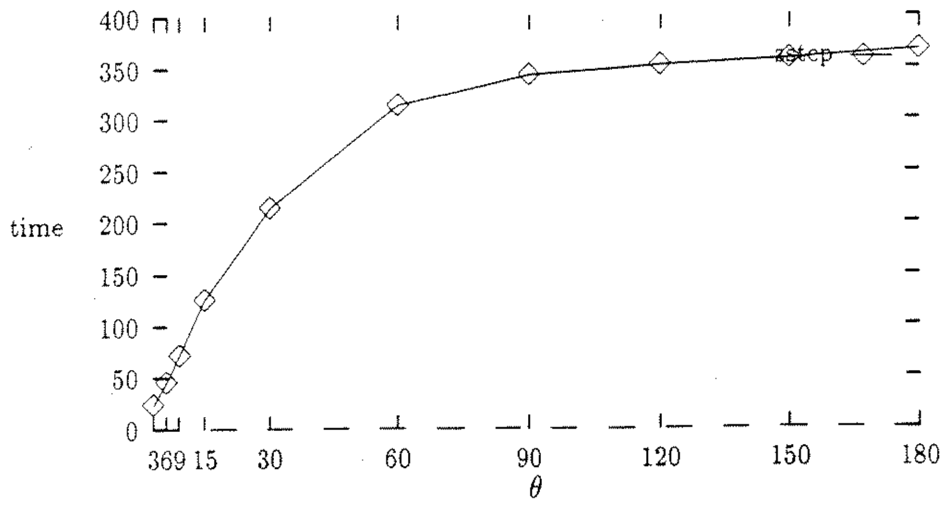


Figure 7.1: Rotation plots, 10 intersections per tool movement, zstep and fastep



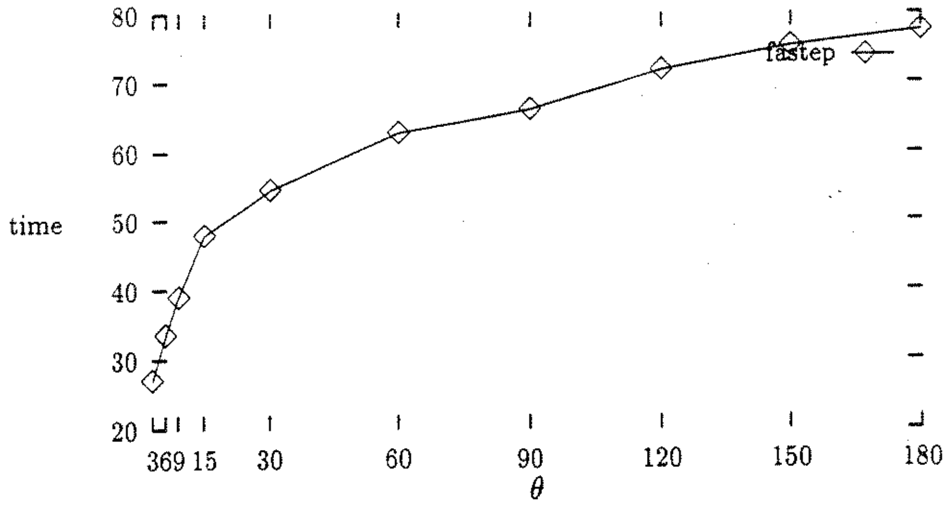


Figure 7.2: Rotation plots, 100 intersections per tool movement, zstep and fastep

The graphs of the rotation data for zstep and fastep are in figures 7.1, and 7.2. The trend for both programs is the same: increase times relatively rapidly at first, then level out some. The fact that both programs behave in the same manner means that both programs experience the same change. As the rotation angle increases, the distance from the bounding box to the actual intersection is likely to increase on average. This is because the three-axis bounding box becomes a less ideal fit as rotation angle goes up. We tested this by finding the average distance to the intersection point. The results are in figure 7.3. As we can see, this curve is very similar to the plots for the step programs, suggesting that this is indeed the explanation.

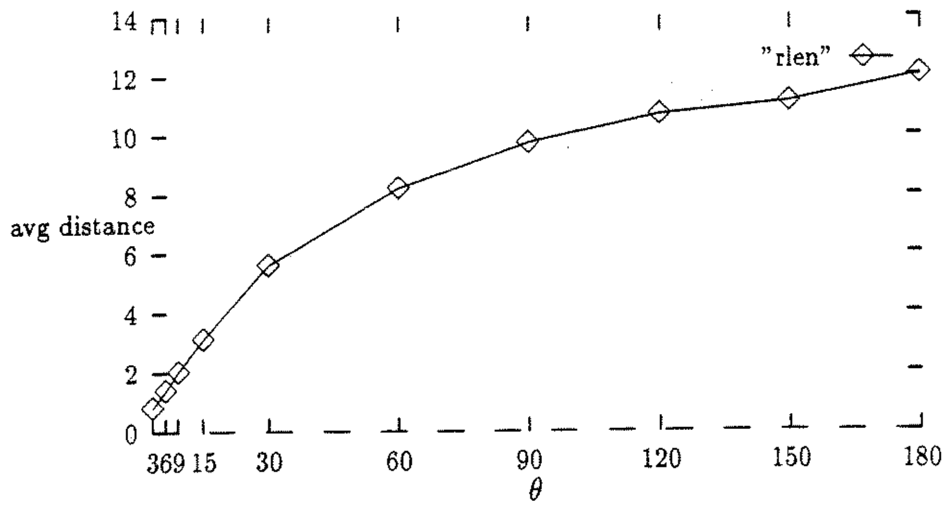


Figure 7.3: Average distance between bounding box and envelope intersection versus rotation angle

3x

The basic three-axis approximation does $O(n)$ preprocessing in building the object list, whose size is linear in the rotation angle. This trend is visible in graph 7.4, which

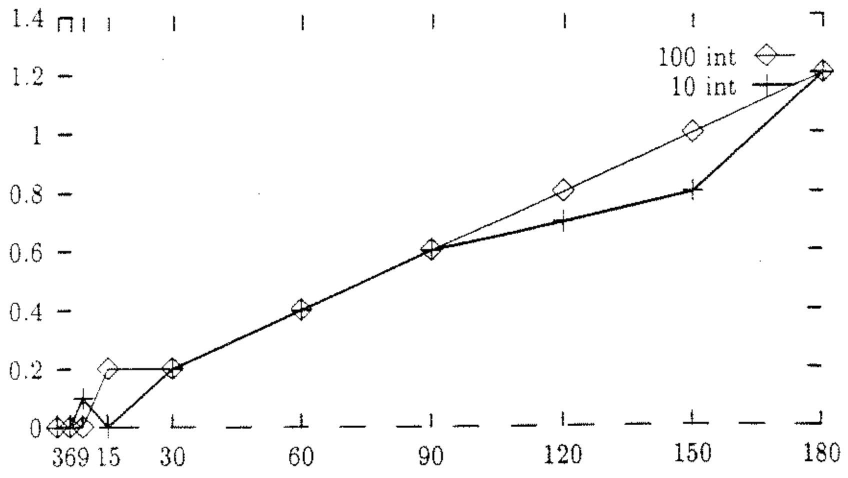
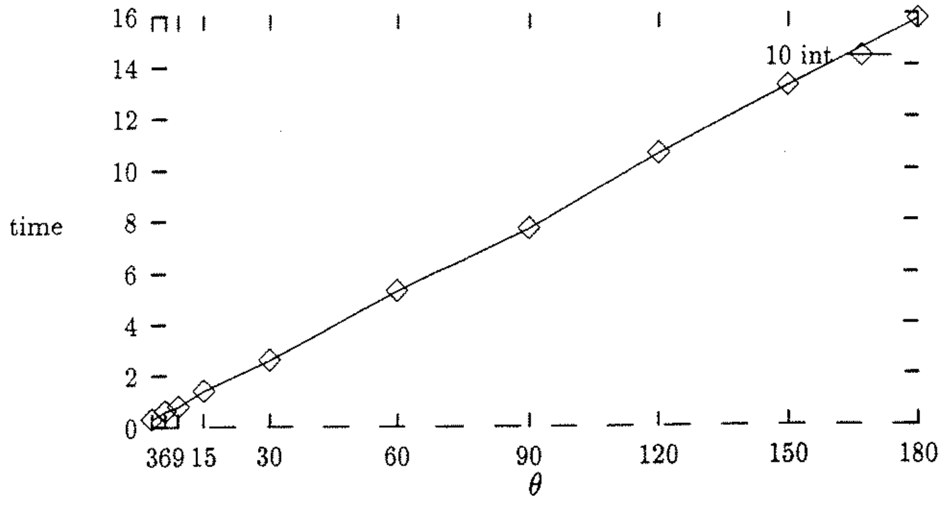


Figure 7.4: Rotation plots, 3x preprocessing, 100 tool movements, 10 and 100 intersection tests per tool movement

plots preprocessing curves for 100 tool movements with 10 and 100 intersections per tool movement. Intersection should also rise linearly since it involves testing every submovement in the list for each intersection test. We can see this in the graphs in figure 7.5.

The 3xtree program should also show $O(n)$ response to the increase in angle for preprocessing times since the hierarchy tree is built from the bottom up. This shows up in figure 7.6, which plots rotation angle versus preprocessing times for 100 tool movements with 10 and 100 intersection tests each.

The intersection times for the 3xtree program should be sublinear. Since it is a tree search with potential multiple branches, it will be $\Omega(\log n)$ in general. However, it should still be substantially less than $O(n)$ most of the time. As we see from the graphs in figure 7.7, the intersection running times are clearly less than linear. In the right-hand graph with 100 intersections per tool movement, we plotted the guide



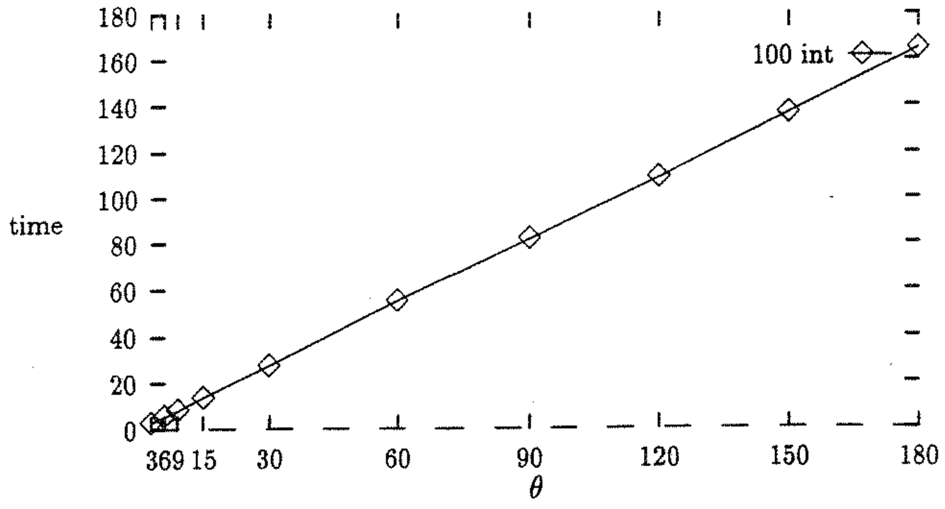


Figure 7.5: Rotation plots, 3x intersection times, 100 tool movements, 10 and 100 intersection tests per tool movement

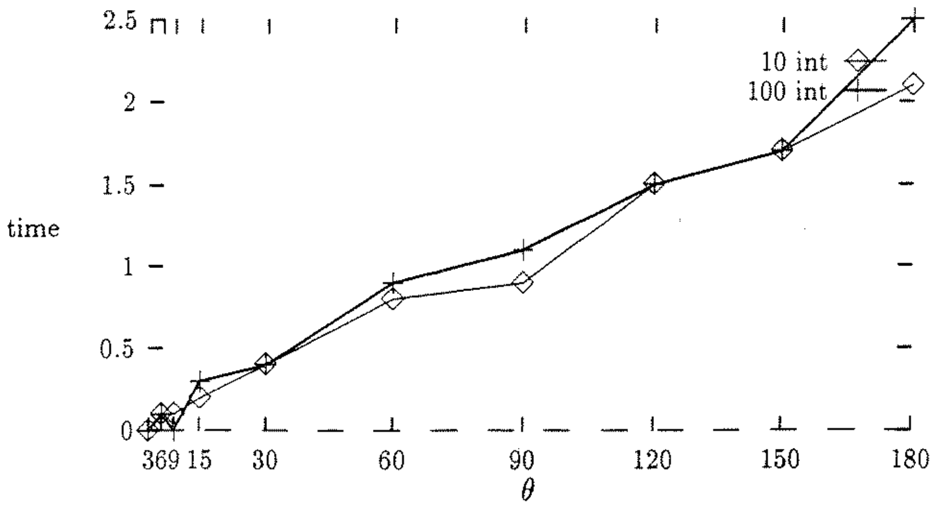
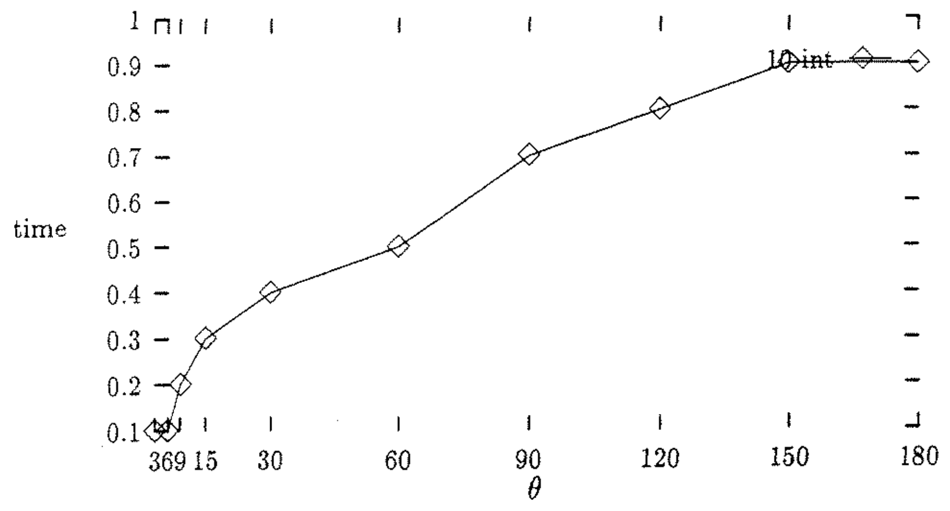


Figure 7.6: Rotation plots, 3xtree preprocessing, 100 tool movements, 10 and 100 intersection tests per tool movement

curve $\frac{1}{2}\theta^{.6}$, which appears to fit well. It suggests that intersection times are $O(n^{1-\epsilon})$ in general with decreasing rotation step.

poly

These are results for the unbounded version poly nb. Although the mesh generation is done with a different subdivision criteria, results may still be applicable to the bounded polygonal method. First, we see that preprocessing is super-linear. This is shown in figure 7.8 for both 10 and 100 intersection tests per tool movement. The curves differ because different tool movements occur in the test sets used. As the rotation angle rises, it is obvious that more triangles will be needed to approximate the surface, since a larger curved surface must be covered with polygons. However, that is only an approximately linear effect. What happens here is that the rotation increases while the linear portion of the tool movements stays constant. In effect,



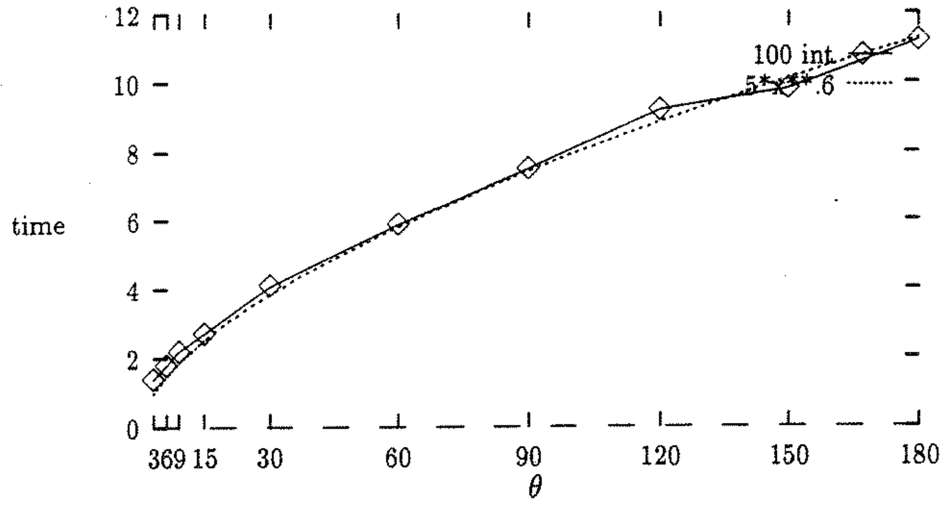


Figure 7.7: Rotation plots, 3xtree intersection times, 100 tool movements, 10 and 100 intersection tests per tool movement

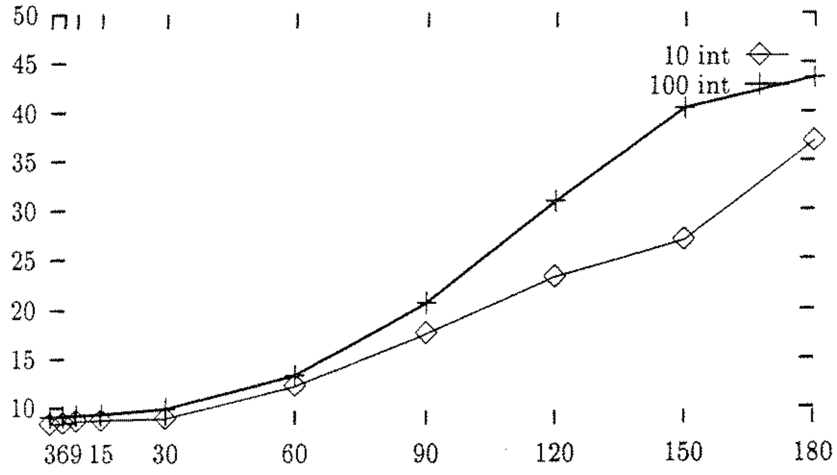


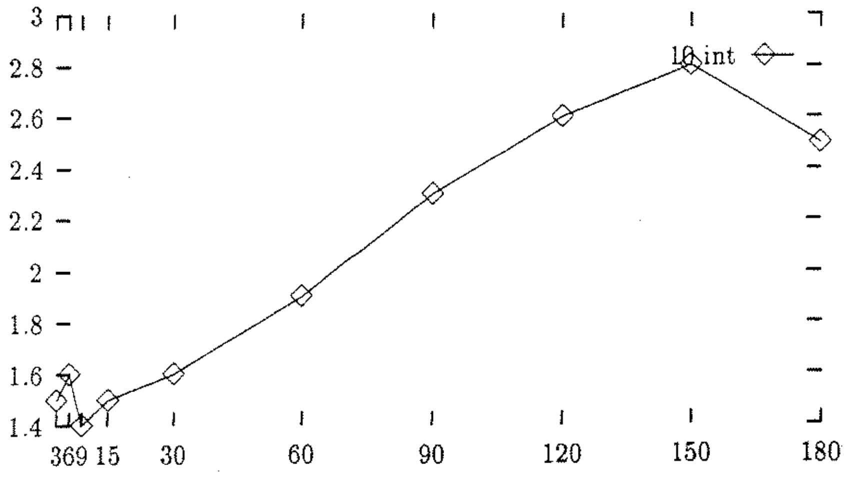
Figure 7.8: Rotation plots, poly nb preprocessing, 100 tool movements, 10 and 100 intersection tests per tool movement

this increases the pitch, or rate of rotation per linear distance, which increases the curvature of the surface. This in turn requires even more triangles to successfully approximate the surface. Therefore, overall, as rotation angle increases, both the surface area to be covered and the necessary density of triangles increase at the same time.

The rise in intersection times is to be expected given the rise in preprocessing. Preprocessing is responsible for generating triangles and inserting them into the tree, so there is a direct correlation between time spent and model size. What we see is a roughly linear growth in intersection time with rotation angle, shown in figure 7.9.

1d

The 1d program has a very complex dependency on the tool movement variables in terms of how changes affect the equations. In addition, even a predictable change in



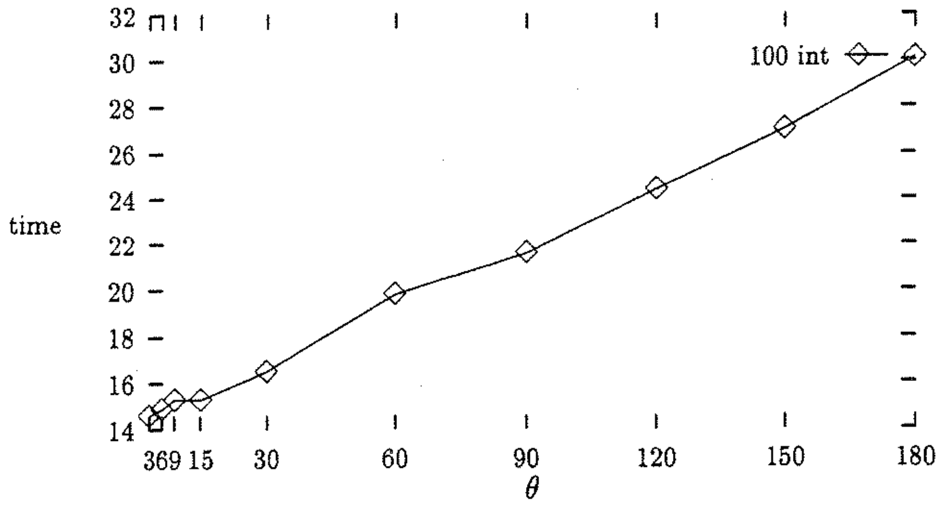


Figure 7.9: Rotation plots, poly nb intersection, 100 tool movements, 10 and 100 intersection tests per tool movement

the shape of a curve may lead to unpredictable effects in the root-finding processes. What we see in the graphs in figure 7.10 is a roughly linear relationship to changes in rotation angle.

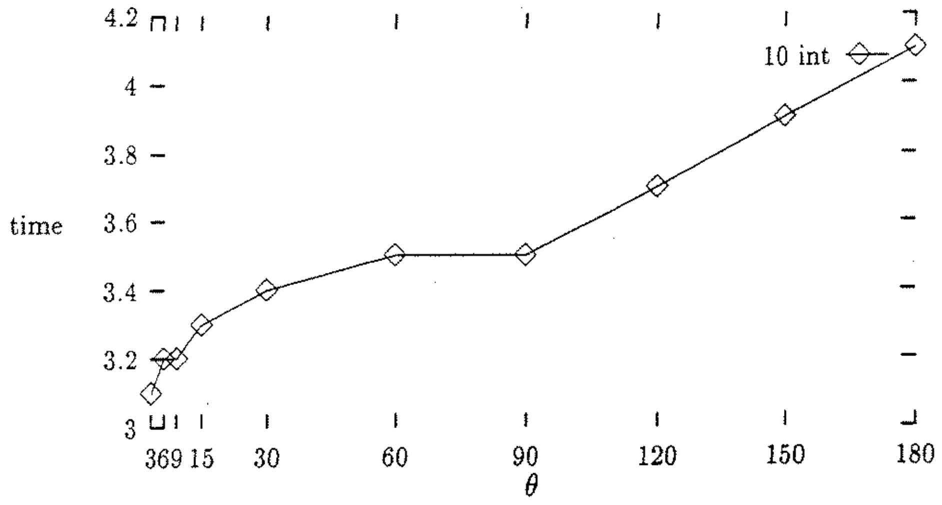
In the rad program, the graph for 10 intersections per tool movement is very chaotic. This illustrates the highly variable performance of the bounding technique. One intersection test can be quite fast while the next will take an incredibly long time. The variability is so large that any possible trend is obscured. In the graph for 100 intersections per tool movements, the variability is averaged out some and we see a trend. For very small angles, performance is very slow, with times decreasing quickly until around 45 degrees and then flattening out. The incredibly slow performance clearly justifies not pursuing theoretical reasons for the trends. The plots are in figures 7.10 and 7.11.

7.2.3 Linear Movement Length

Having looked at the effects of rotation angle on program performance, we now look closer at the linear portion of a tool movement. As linear movement length decreases, surface curvature increases and vice versa. We varied the lengths of the linear part of the tool movements from .0625 times tool length to 3 times tool length. The graphs are all plotted with a log scale on the x (length) axis. This is necessary to keep the data spread out, otherwise everything will clump up towards 0. As a result, a curve that appears linear is in fact logarithmic, while a truly linear function will be plotted as an exponential curve.

step

We can see from the graphs in figures 7.12 and 7.13 that both the zstep and fastep are affected similarly by change in tool movement length when rotation angle is limited



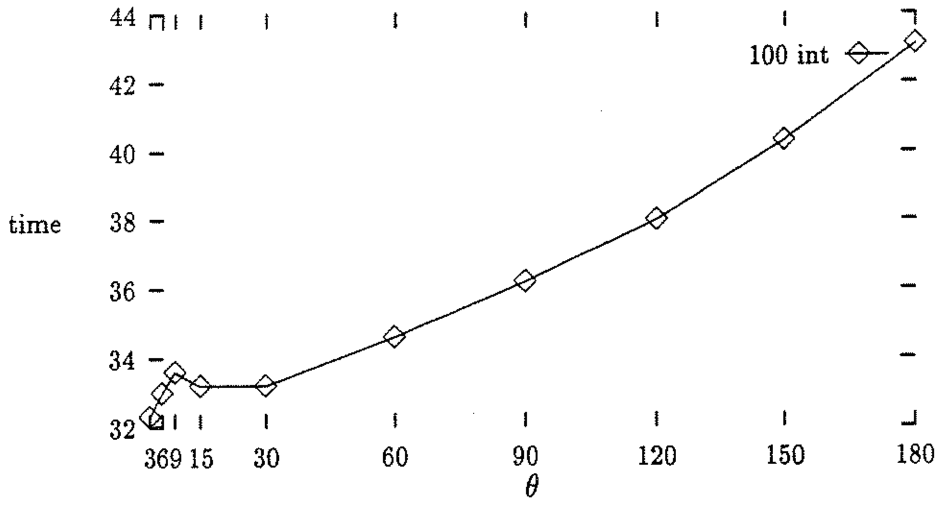
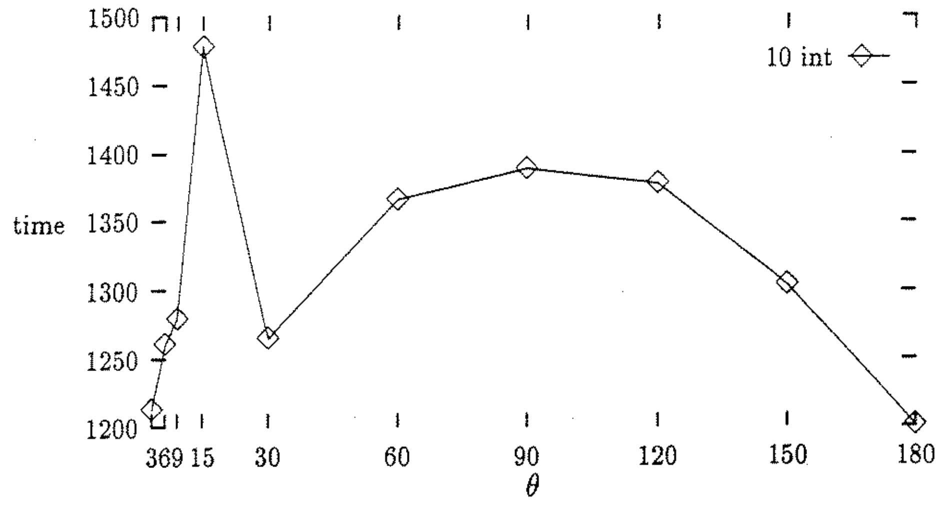


Figure 7.10: Rotation plots, 1d intersection, 100 tool movements, 10 and 100 intersection tests per tool movement



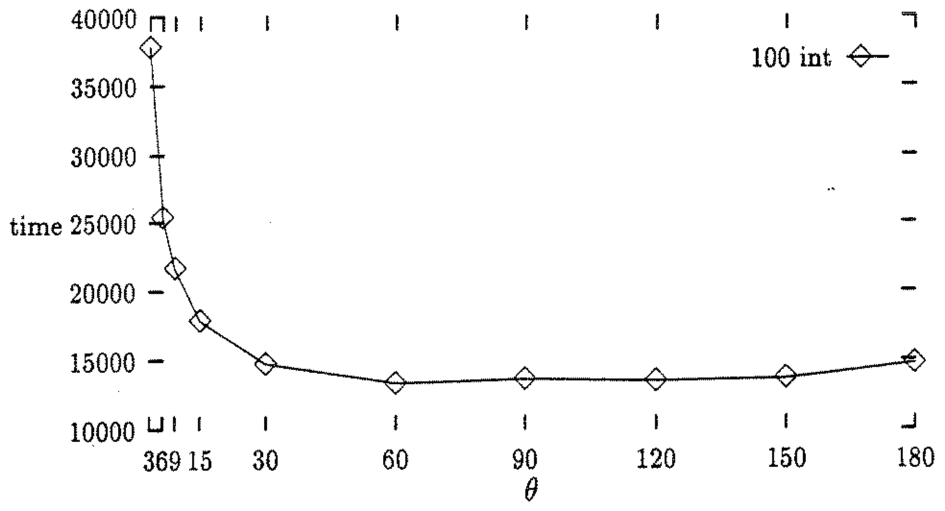


Figure 7.11: Rotation plots, rad intersection, 100 tool movements, 10 and 100 intersection tests per tool movement

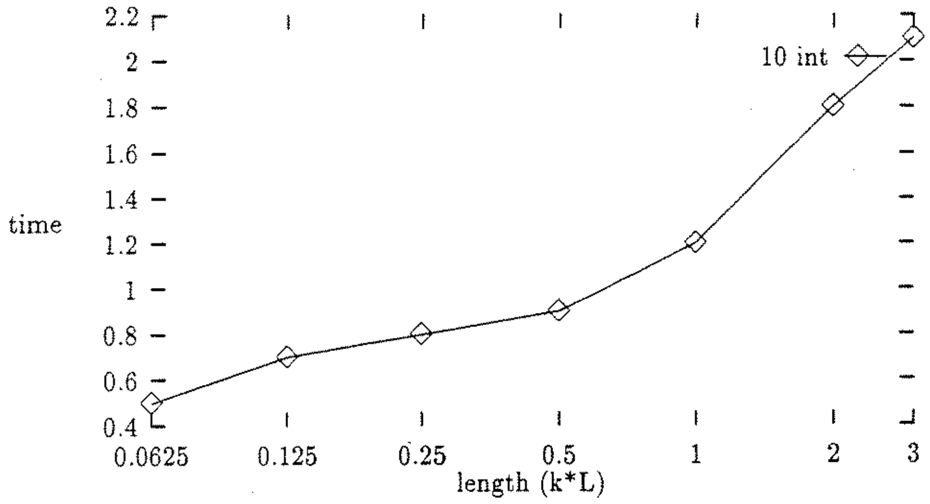
to 3 degrees. As the linear motion increases, the tool movement and bounding box are stretched out in the direction of the motion. For a line that heads straight in towards the tool movement, no change occurs. However, if the line comes in at an angle, the distance from the bounding box intersection to the envelope intersection has been stretched in the linear motion direction. Therefore, as the linear part of the tool movement increases, the average distance between bounding box and envelope intersections will increase. This is apparent in the graph of average distance versus linear tool movement distance in figure 7.14.

When the tool movement is allowed to swing 90 degrees, the zstep program becomes much slower than the fastep program. The bounding box to envelope intersection distance has increased enough to amortize the fastep algorithm overhead and take advantage of the savings it provides. The relationship of both methods to changes in linear movement remains similar to above as we see in figures 7.15 and 7.16. Again we see the importance of the increasing average distance traversed to this effect shown by the similarity in figures 7.14 and 7.17.

3x

For the 3x and 3xtree programs the most obvious effect of increasing linear distance is to stretch the individual submovements. With 3 degrees of rotation, the tool movements are fairly close to being flat, requiring few submovements to approximate. In the tests we performed, no noticeable effect due to length changes was seen.

With 90 degrees of rotation, we see some variation, shown in figures 7.18 and 7.19. The 3x and 3xtree programs both show slight decreases in intersection times with longer tool movements. In the case of the 3xtree program, the reason is easy to see. The 3xtree program does a pseudo-binary search that goes down both branches sometimes. This occurs because both children of a node tend to overlap highly in



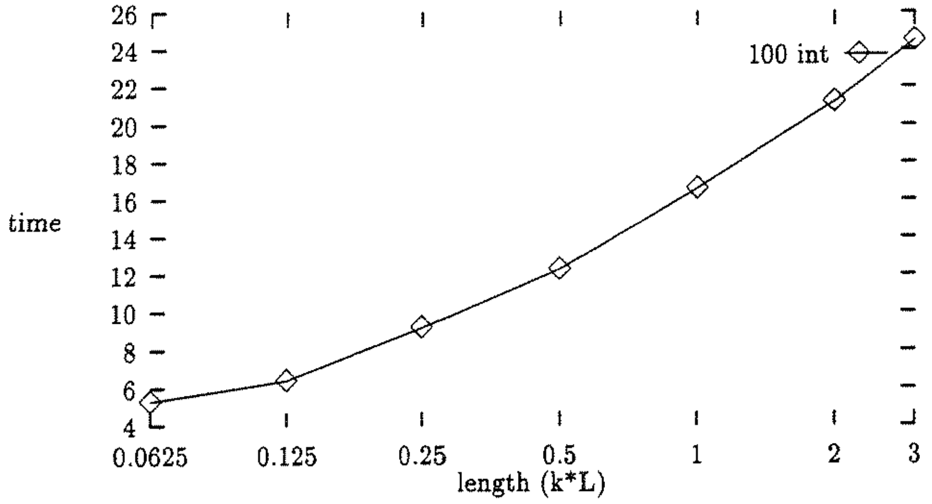
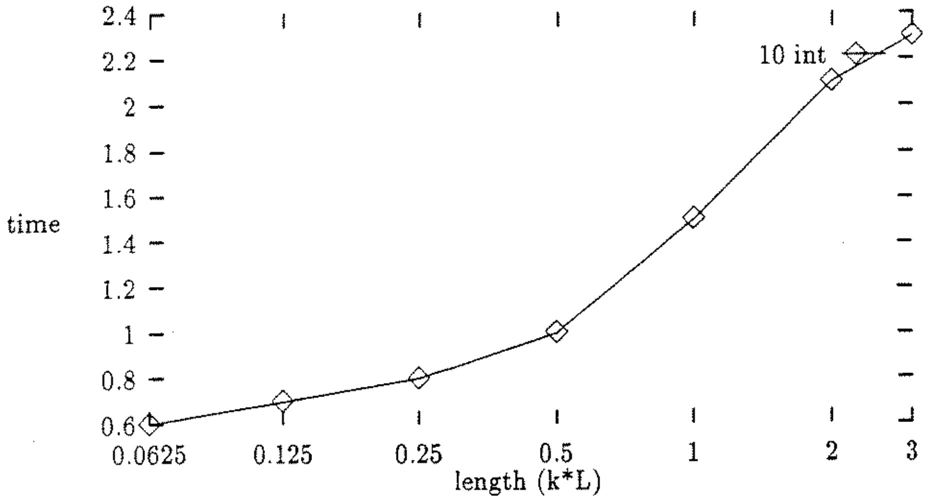


Figure 7.12: Linear distance plots at 3 degrees, zstep intersection, 100 tool movements, 10 and 100 intersection tests per tool movement



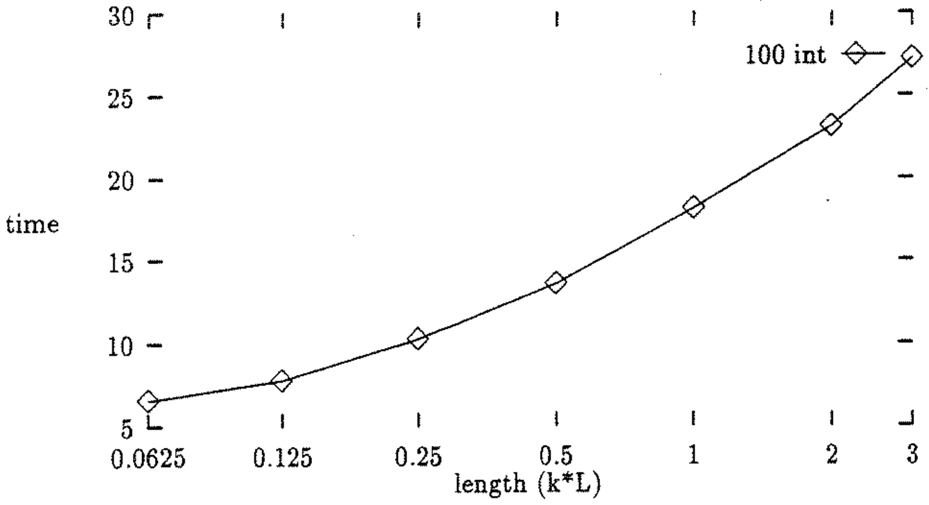


Figure 7.13: Linear distance plots at 3 degrees, fastep intersection, 100 tool movements, 10 and 100 intersection tests per tool movement

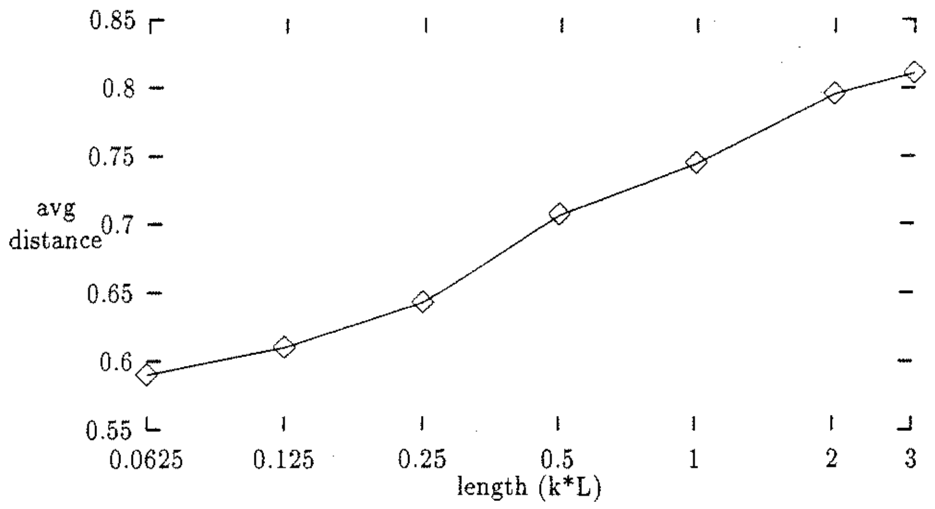
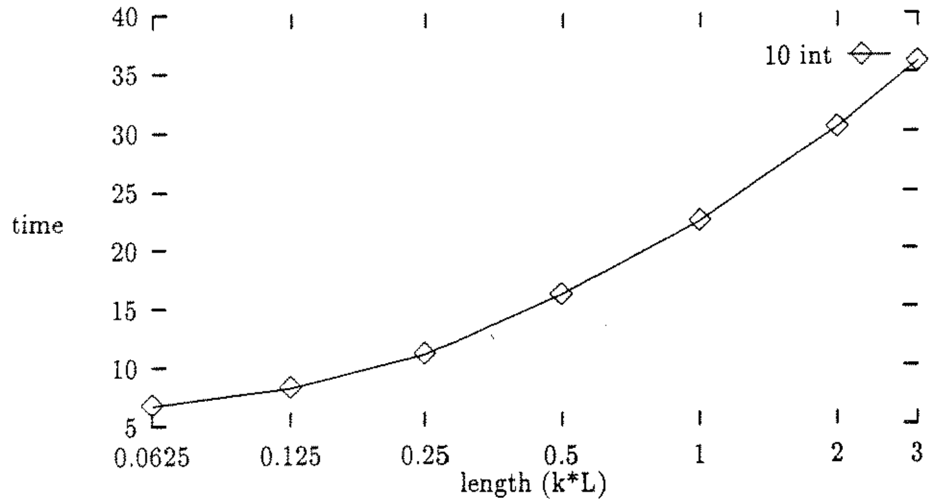


Figure 7.14: Average distance between bounding box and envelope intersection versus length, 3 degrees

space. For a longer tool movement, each submovement is longer as well, resulting in less overlap, and subsequently fewer trips down both sides of a node. This trend only shows up in the larger rotation angle tool movements. The larger the tree is, the more important this effect.

The 3x program exhibits similar behavior, but the reason is less obvious. The line is intersected with every tool movement, so it doesn't seem like any change should occur. However, as the tool movement lengthens the overlap decreases, cutting down the number of submovements that the line actually hits. The intersection of a line and a three-axis tool movement involves a number of expensive floating point operations such as square root calculation. The intersection routine therefore tries to avoid as many operations as possible. So, when the line intersects fewer submovements, more of the floating point operations are avoided, and therefore the program runs faster.

Preprocessing for both programs is unaffected, as expected.



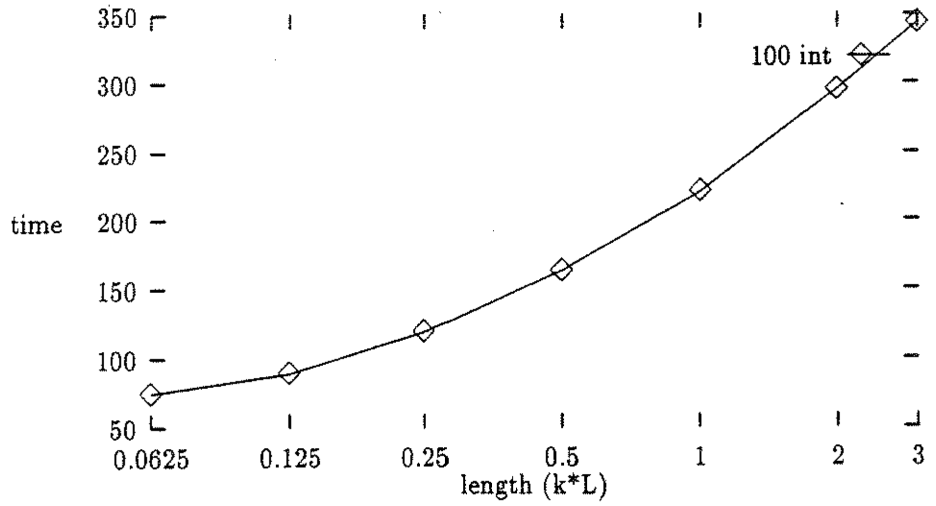
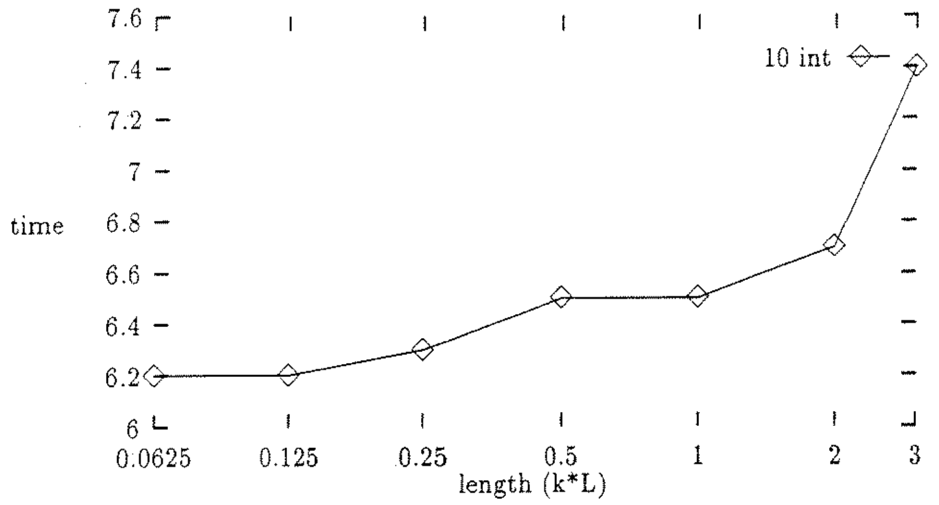


Figure 7.15: Linear distance plots at 90 degrees, zstep intersection, 100 tool movements, 10 and 100 intersection tests per tool movement



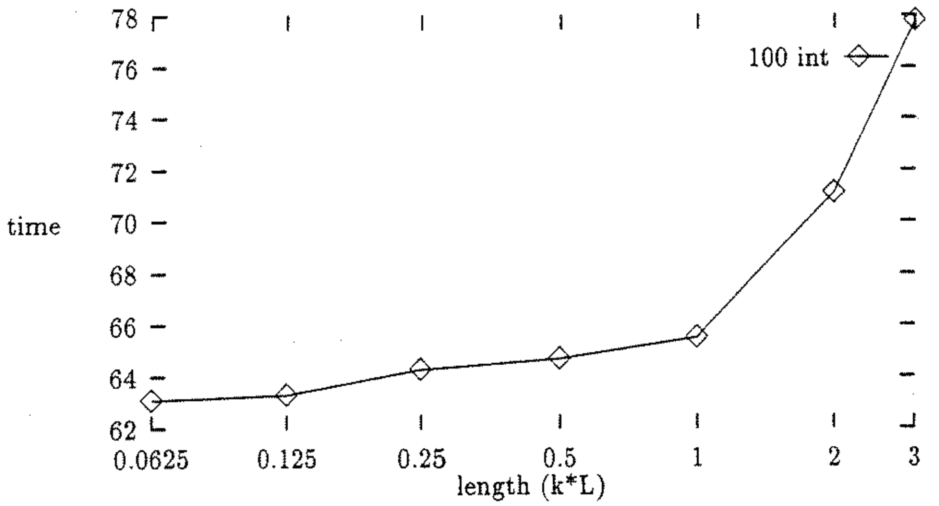


Figure 7.16: Linear distance plots at 90 degrees, fastep intersection, 100 tool movements, 10 and 100 intersection tests per tool movement

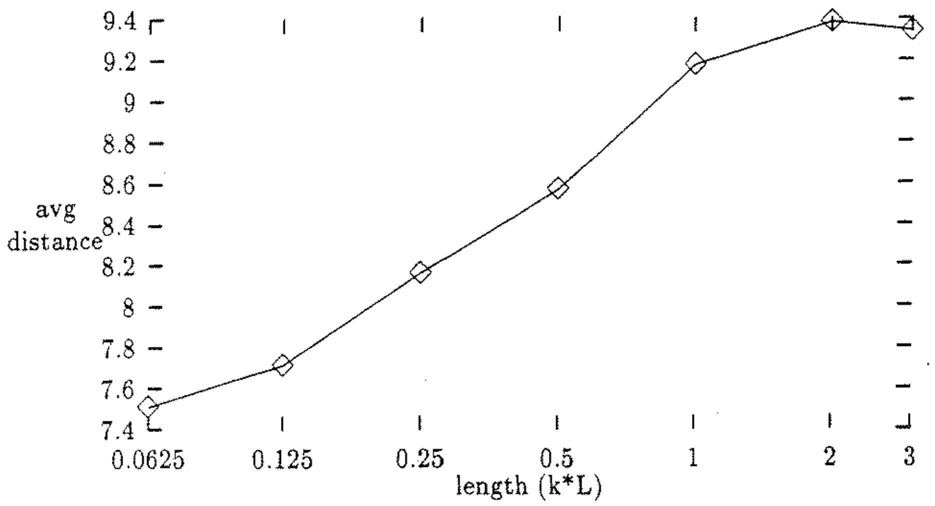
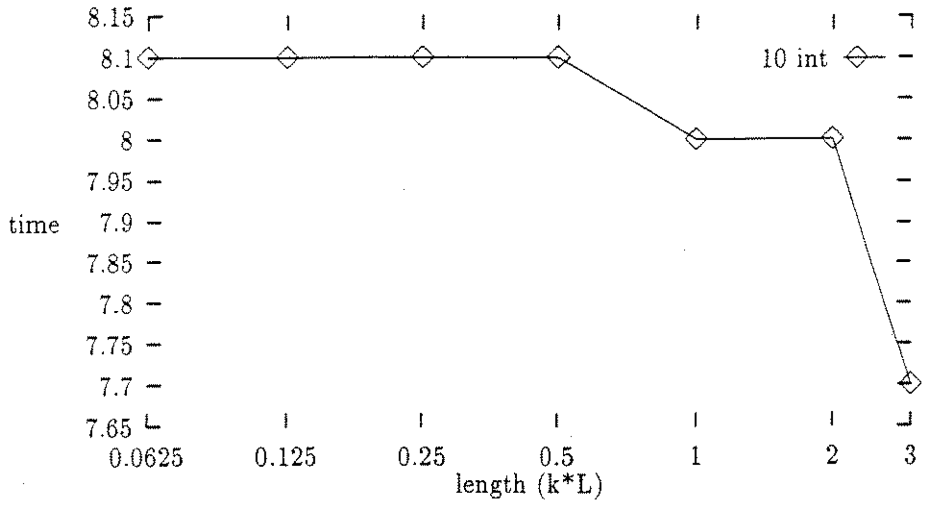


Figure 7.17: Average distance between bounding box and envelope intersection versus length, 90 degrees

poly

As was mentioned in the rotation angle discussion about the poly programs, for a given rotation angle, a shorter linear movement increases the pitch (rotation per distance traveled), while a longer movement reduces the pitch. Increased pitch is accompanied by increased surface curvature, therefore increasing the amount of subdivision necessary to satisfy the flatness criterion. This results in a larger model. We can see this in the preprocessing curves for both 3 degrees and 90 degrees shown in figure 7.20. Each graph shows the preprocessing curve for 10 and 100 intersection tests per tool movement. The 90 degree cases show a clearer trend.

Most of the intersection times don't show a trend. They bounce around in a narrow range. However, in the 90 degree case with 100 intersection tests per tool movement, we have a clear rise in times with lengthening tool movements. This is shown in figure 7.21. This indicates a poorer distribution in the BSP tree, since the



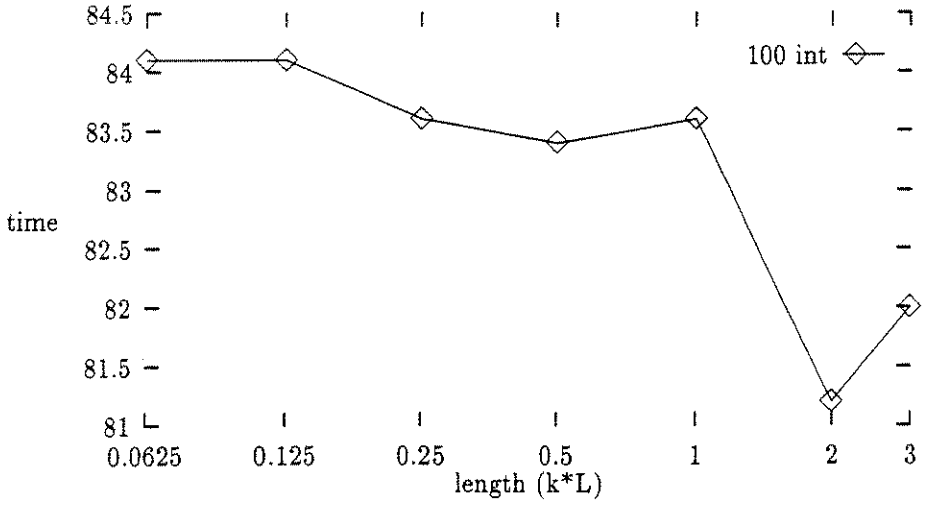
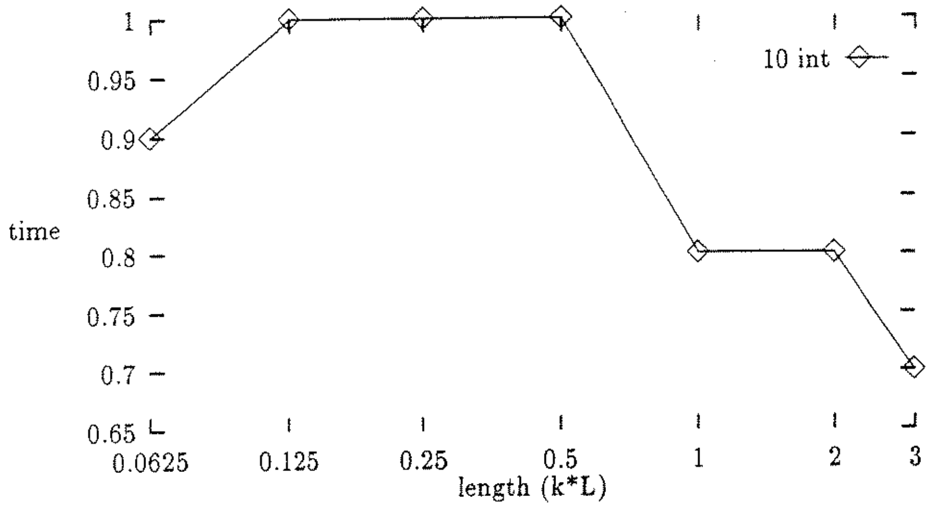


Figure 7.18: Linear distance plots at 90 degrees, 3x intersection, 100 tool movements, 10 and 100 intersection tests per tool movement



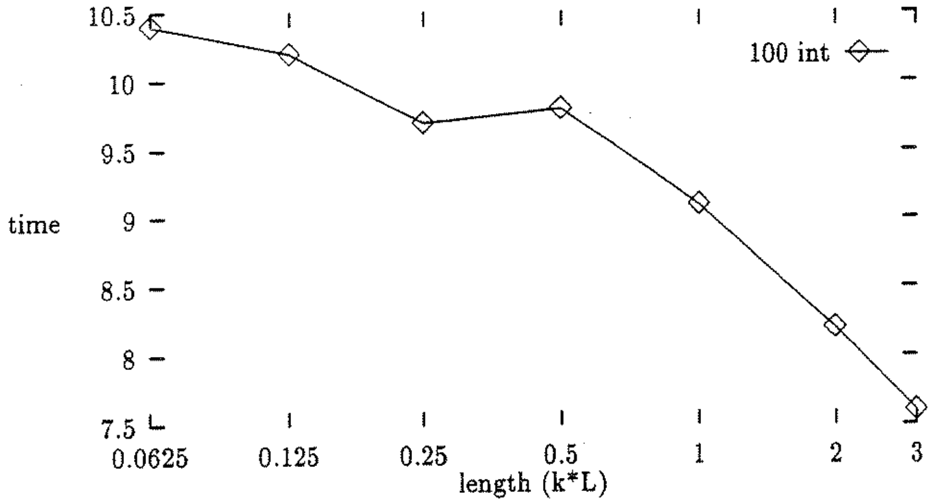
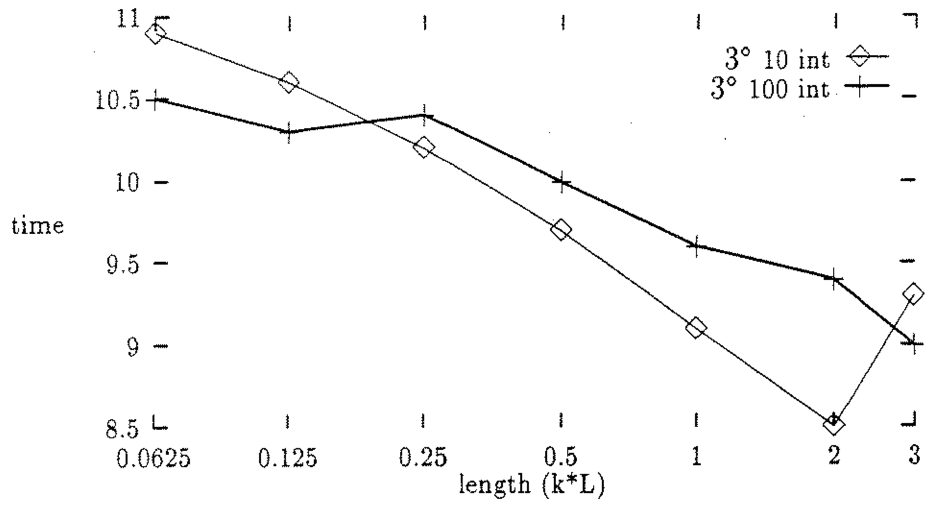


Figure 7.19: Linear distance plots at 90 degrees, 3xtree intersection, 100 tool movements, 10 and 100 intersection tests per tool movement



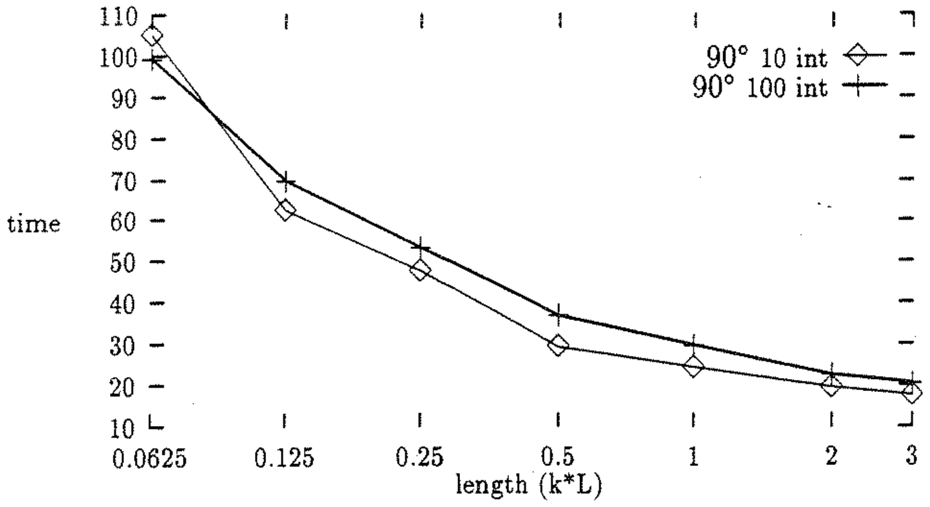


Figure 7.20: Linear distance plots at 3 and 90 degrees, poly nb preprocessing, 100 tool movements, 10 and 100 intersection tests per tool movement

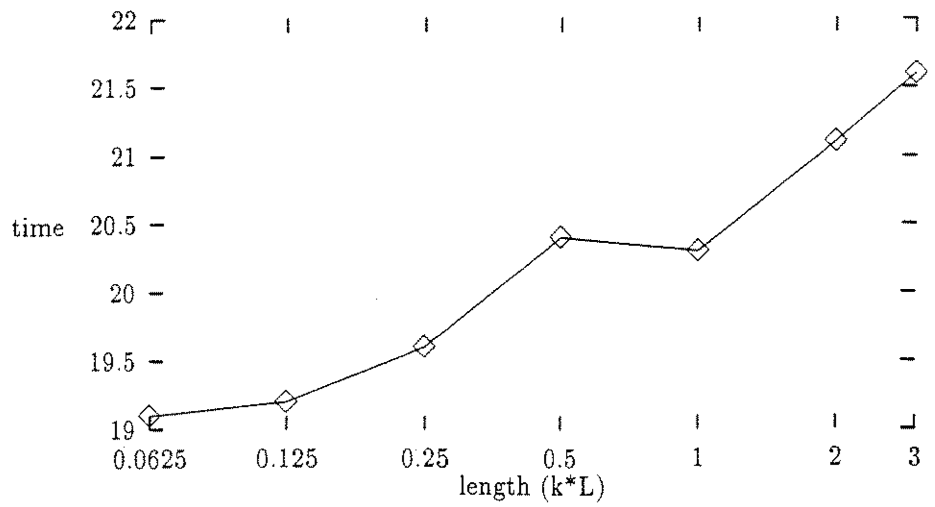


Figure 7.21: Linear distance plot 90 degrees, poly nb intersection, 100 tool movements, 100 intersection tests per tool movement

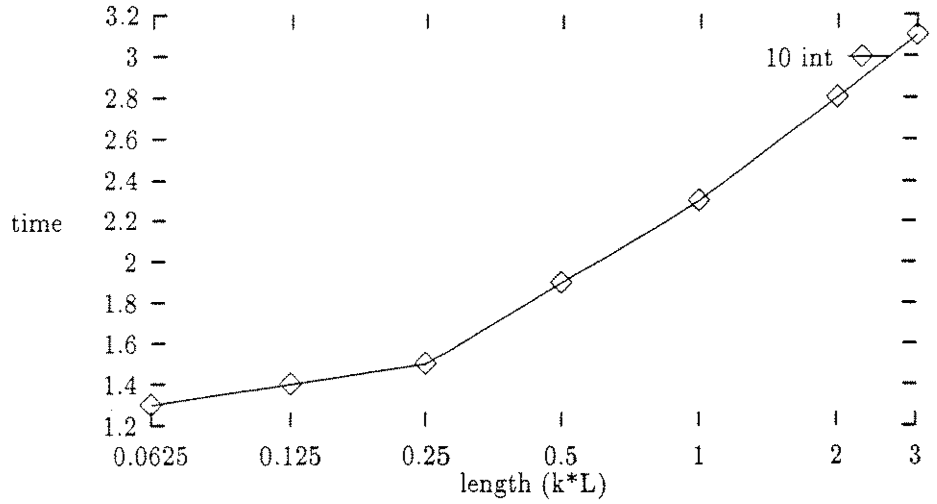
model size is decreasing with the lengthing of the tool movements.

1d

The 1d and rad programs show similar increases in running times with lengthening tool movements. This is true for both 3 degrees and 90 degrees. The plots are in graphs in figures 7.22, 7.23, 7.24, and 7.25, which contain data for the 1d program at 3 and 90 degrees and the rad program at 3 and 90 degrees, respectively. The cause of this effect is unknown, but we have not explored further due to the poor performance of the method.

7.2.4 Tolerance

The final variable in program performance we addressed is intersection accuracy. This was done using random test data and also some real world examples. As with



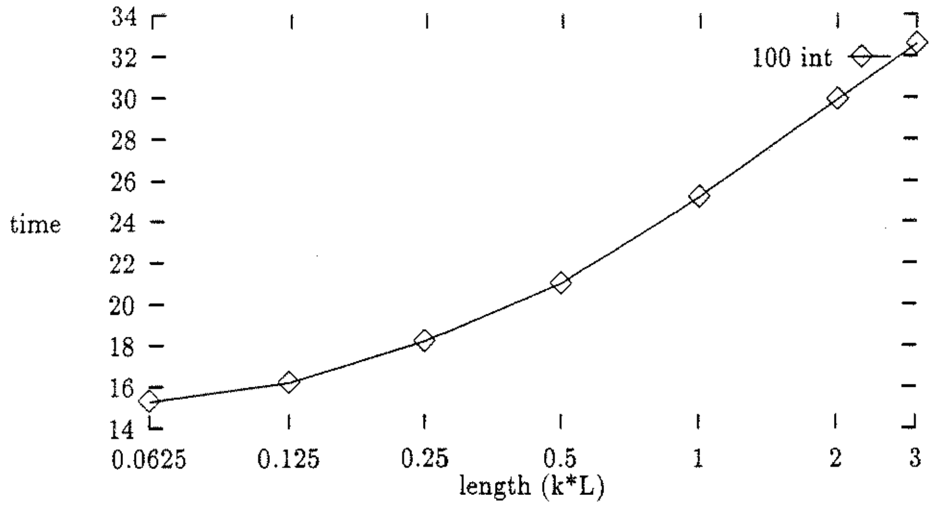
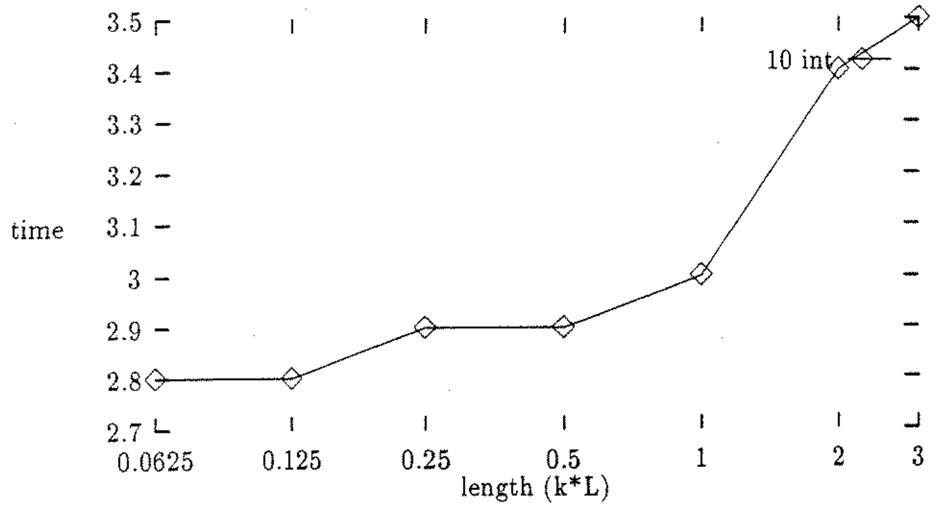


Figure 7.22: Linear distance plots at 3 degrees, 1d intersection, 100 tool movements, 10 and 100 intersection tests per tool movement



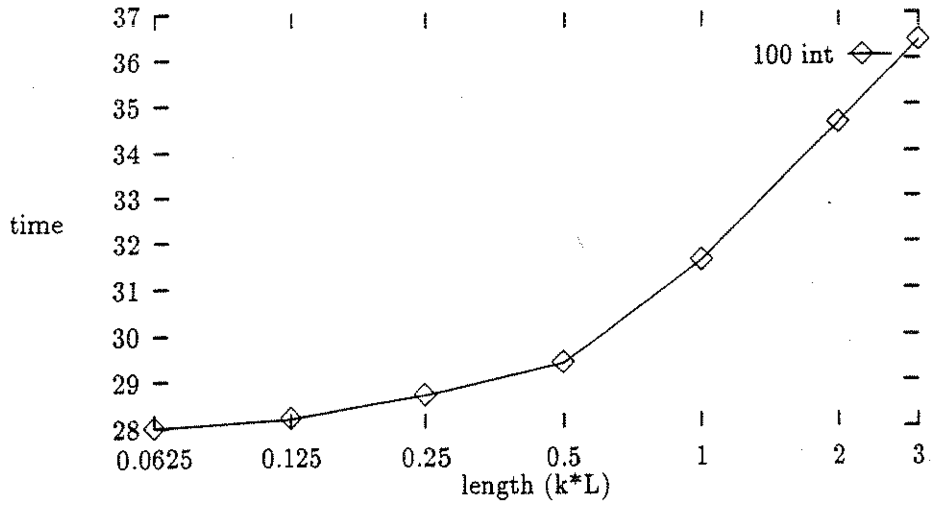
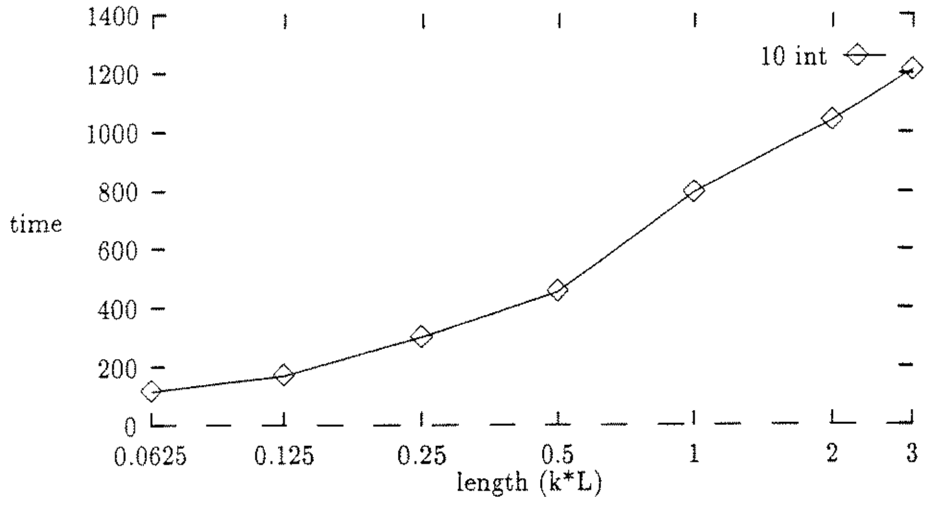


Figure 7.23: Linear distance plots at 90 degrees, 1d intersection, 100 tool movements, 10 and 100 intersection tests per tool movement



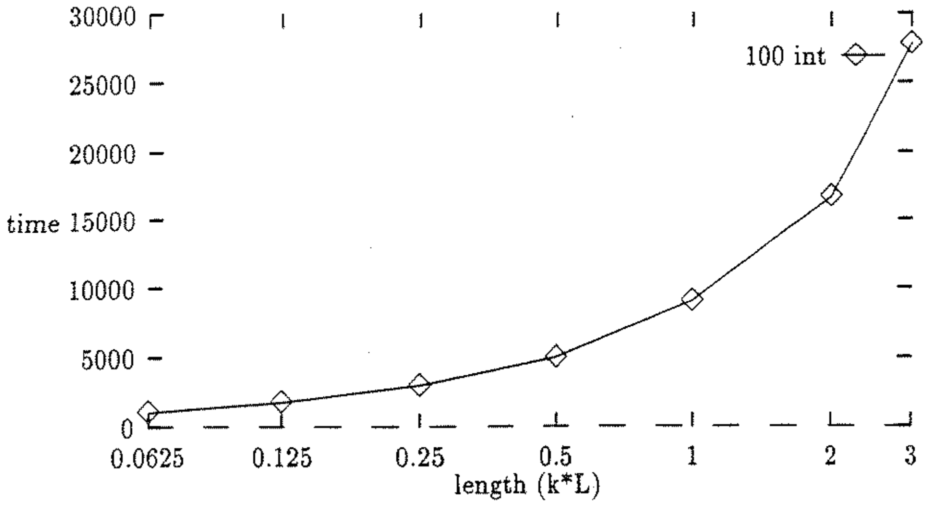
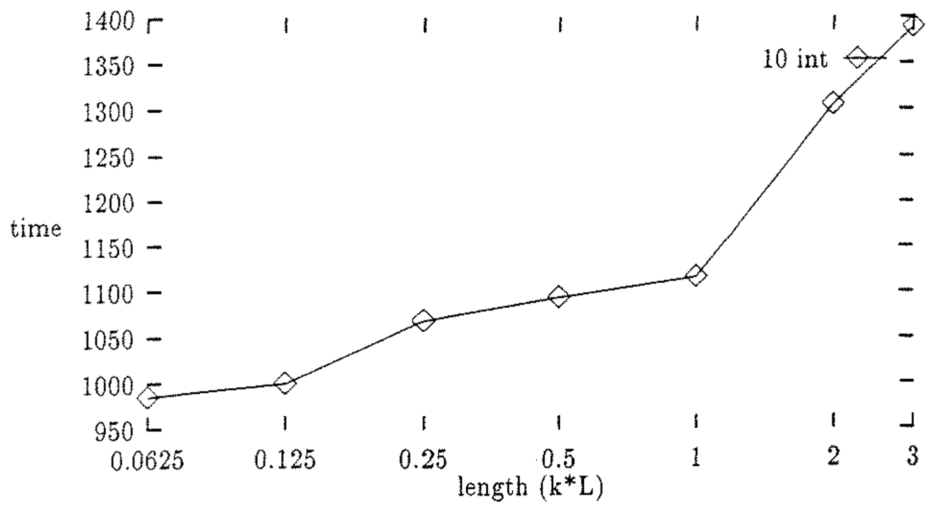


Figure 7.24: Linear distance plots at 3 degrees, rad intersection, 100 tool movements, 10 and 100 intersection tests per tool movement



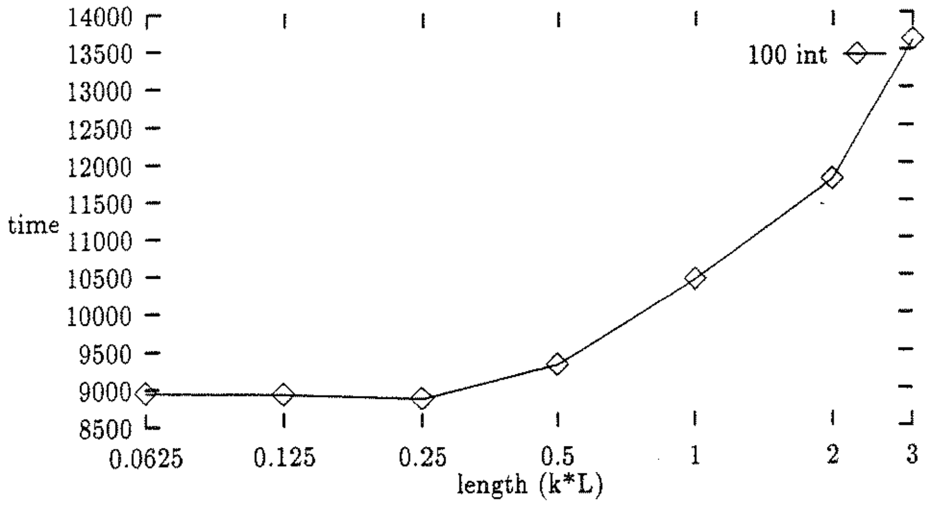


Figure 7.25: Linear distance plots at 90 degrees, rad intersection, 100 tool movements, 10 and 100 intersection tests per tool movement

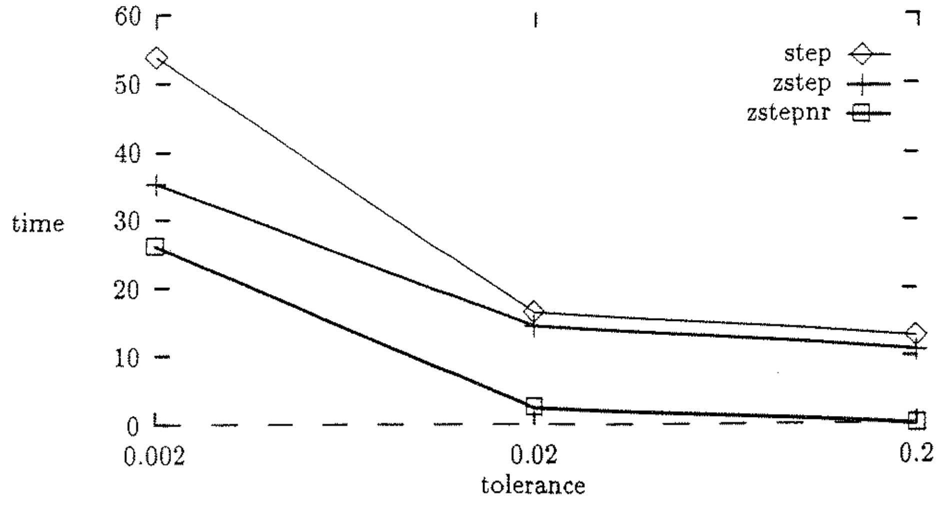
the linear movement changes, tolerance data is plotted using a log scale on the x (tolerance) axis. Increasing accuracy is to the left instead of to the right.

step

The basic step method takes constant-sized steps down the line until an intersection is found. Clearly, there is an inverse linear relationship between the step size and the number of steps taken to find an intersection. As the step size halves, the number of steps doubles. As a result, we should see a linear curve of runtimes for the step method. The zstep program still takes the same set of steps, so it too should see a linear relationship to tolerance. Since the zstep program's culling operation clearly wins over the simple step program, the step program was not evaluated on real data. The zstep program still serves as a good testbed for evaluating the constant step size approach.

The graphs in figure 7.26 and the data don't quite act linearly. A couple of factors affect the data. First, the refinement of intersection answers adds an almost constant factor to the times. It is not actually constant since increasing accuracy reduces step size and hences shaves off a few of the refinement steps. The graphs show the effect of removing the refinement from zstep quite well. We have added a guide curve of $.6/x$, which matches the zstepnr plot well. This supports the linear relationship hypothesis.

Examining the real data for zstepnr, we get similar results. However, it is not exact. The lower tolerance figures are not quite 10 times smaller, as they should be. What we are seeing is most likely constant costs skewing the curve at the low end. For instance, on door_small_def (table 7.15), the times become almost perfect if we subtract a constant of .8 from all the intersection times, giving 3.4, 34.1, and 343.2. Values for door_small_3 (table 7.16), reduced by a constant of 3.3, becomes



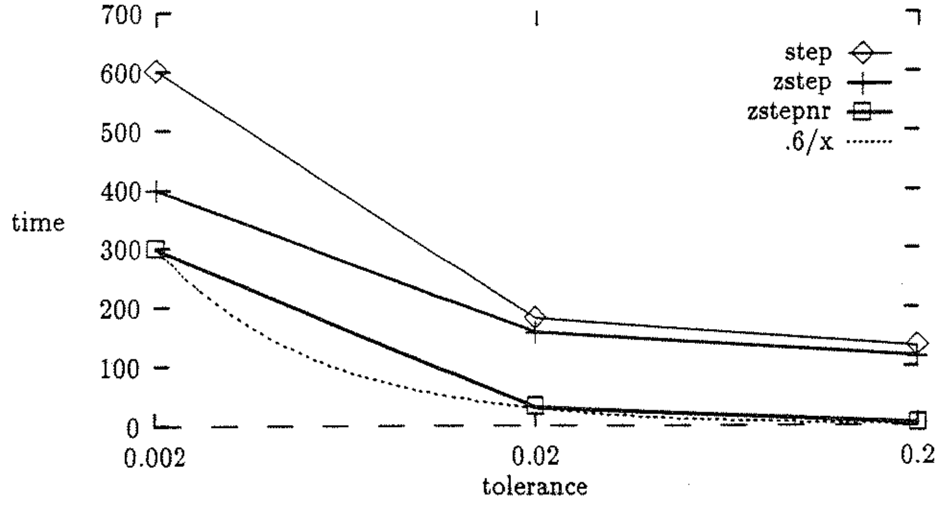


Figure 7.26: Tolerance, step, zstep, zstepnr intersection, 100 tool movements, 10 and 100 intersection tests per tool movement

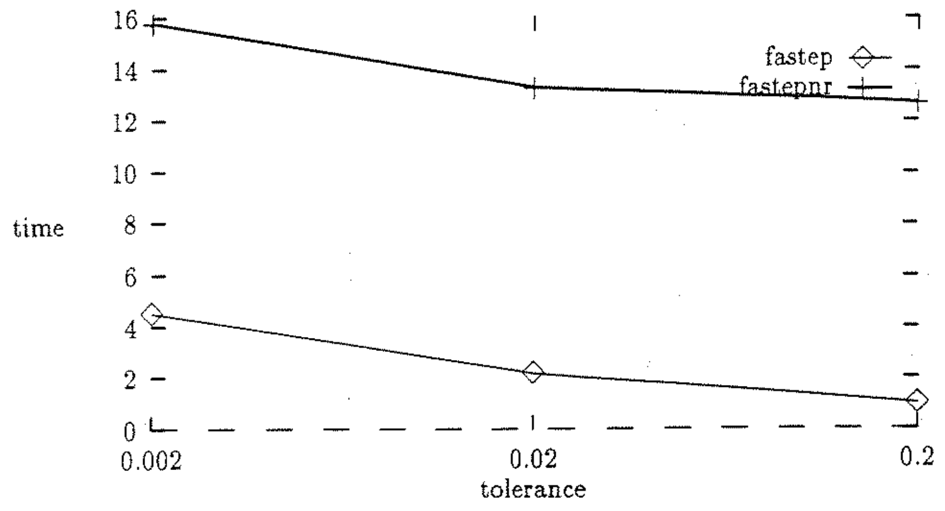
24.3, 242.7, 2434.3. For z6324r_def (table 7.17), if we reduce by 24, we get 47.0, 478.2, and 4731.3. And for z6324r_1.5 (table 7.18), reducing by 112.9 gives 260.1, 2711.9, and 25897.3, which is within reason, although not quite as good as the other three.

In most cases, the fastep program times will be sublinear in the step size. As stated at the end of Chapter 6, in a simplified optimal case, it would be logarithmic in the step size. If we look at just the fastep and fastepnr plots in figure 7.27, we can see that they are nearly flat, indicating performance that is probably polynomial with low fractional exponent or even polylogarithmic. An interesting note is that at the lowest accuracy level, the zstep programs are actually faster than the corresponding fastep programs. That quickly changes once the accuracy is high enough to require more than a few steps to locate intersection values. The real data backs up this conclusion. Graphs in figures 7.28 and 7.29 are curving upward just a little, pointing to performance that is only somewhat worse than logarithmic in the accuracy.

3x

As we have discussed previously, the 3x program intersection times should respond nearly linearly to changes in tolerance. In table 7.9, we don't see this. However, based on the value of 4 seconds for tolerance of .002, the other two times would have to be .4 and .04 seconds, both of which are small enough to obscured by imprecision in the timing. Table 7.10 gives a better set of figures although the value of .7 for tolerance of .2 is too large. However, it is only off by .3 seconds, which can also be explained by timing inaccuracy, constant costs, and random fluctuations since the programs run random tests.

The real data clearly shows the linearity of the 3x program. In both door_small runs and in z6324r_def (tables 7.15, 7.16, and 7.17), the run times are just about perfectly linearly increasing with accuracy. In z6324r_1.5, the entry for a tolerance of



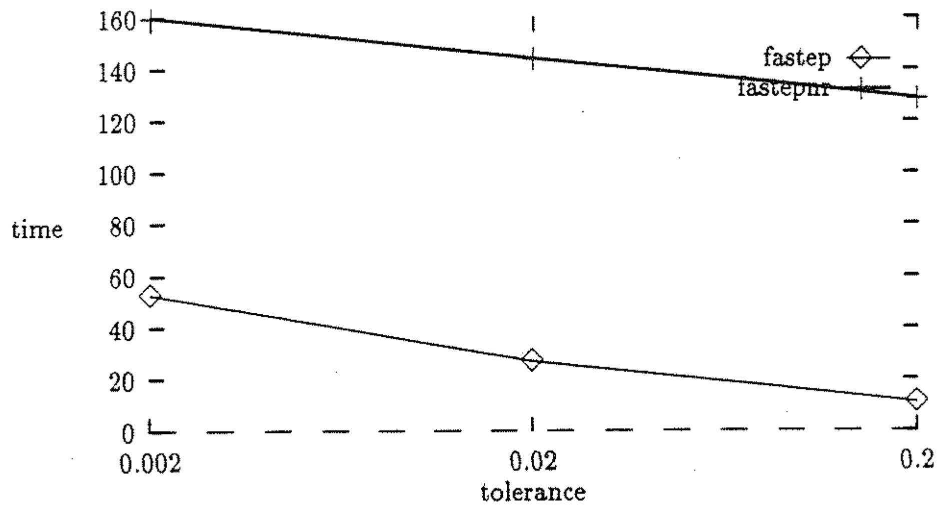


Figure 7.27: Tolerance, fastep, fastepnr intersection, 100 tool movements, 10 and 100 intersection tests per tool movement

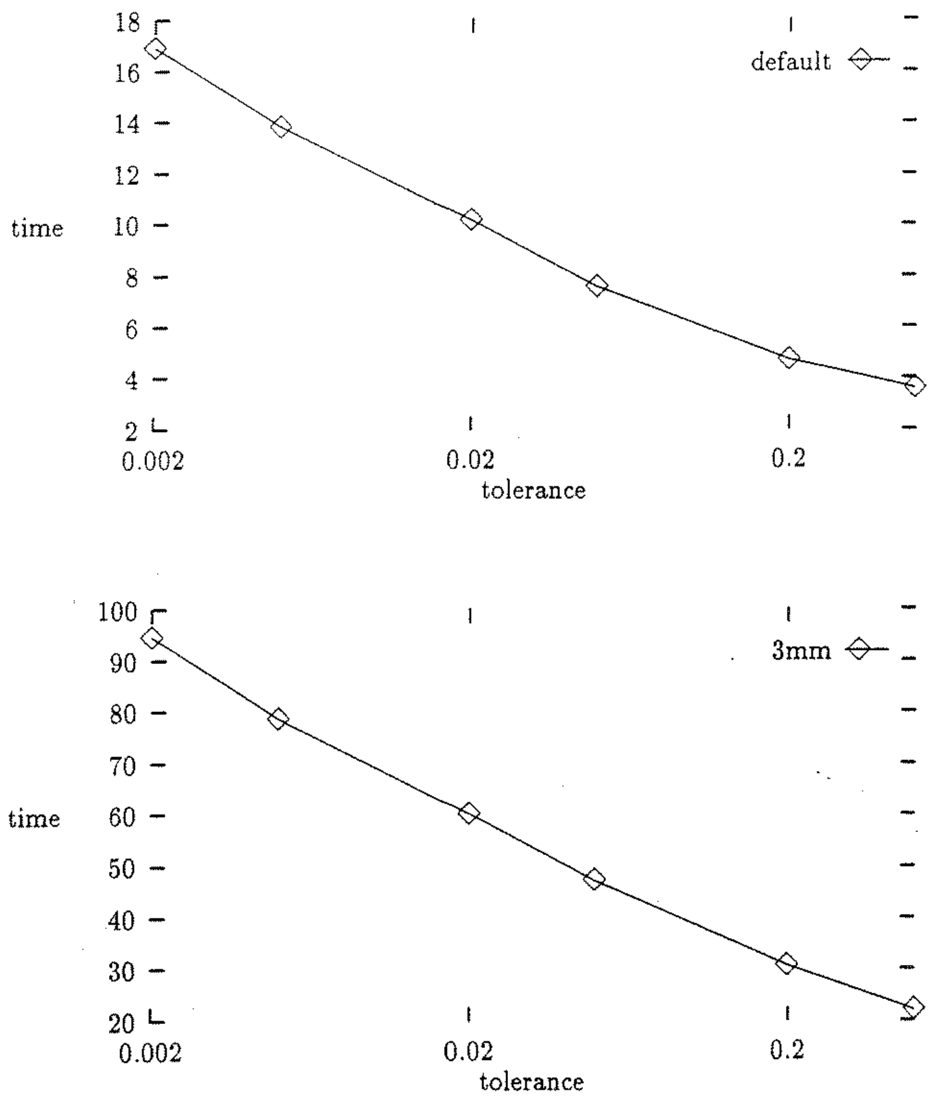
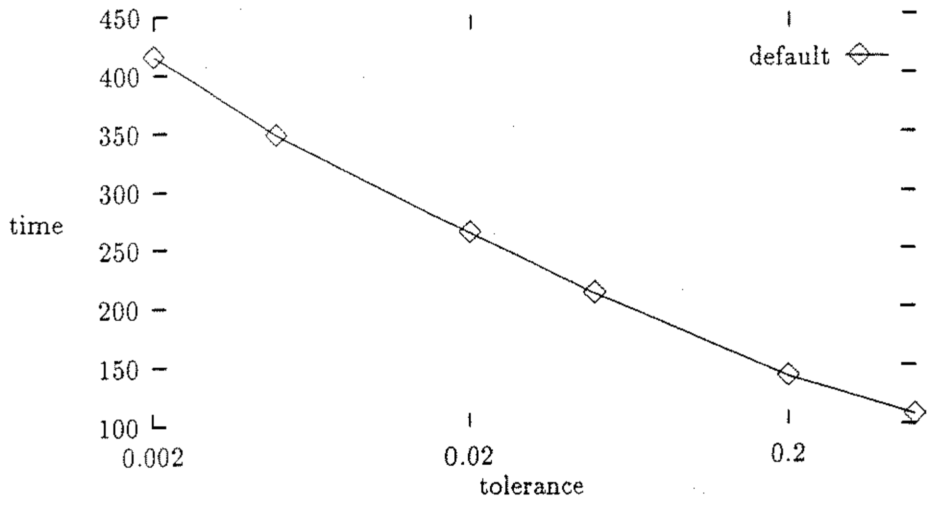


Figure 7.28: Tolerance on door_small, default and 3mm, fastepnr intersection



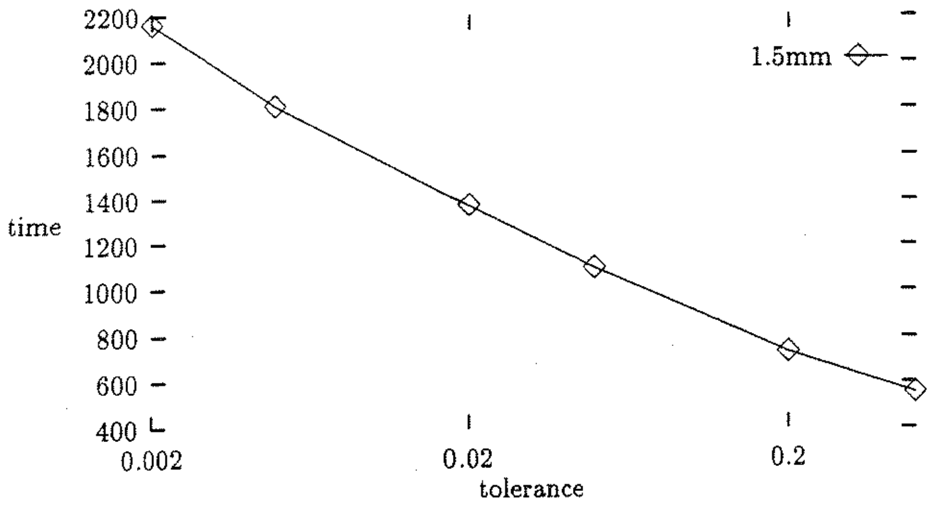


Figure 7.29: Tolerance on z6324r, default and 1.5mm, fastep intersection

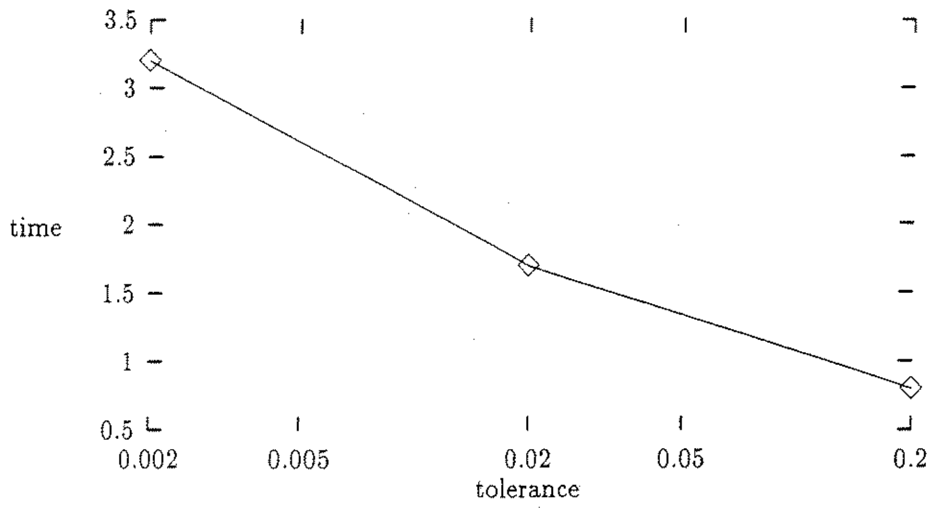
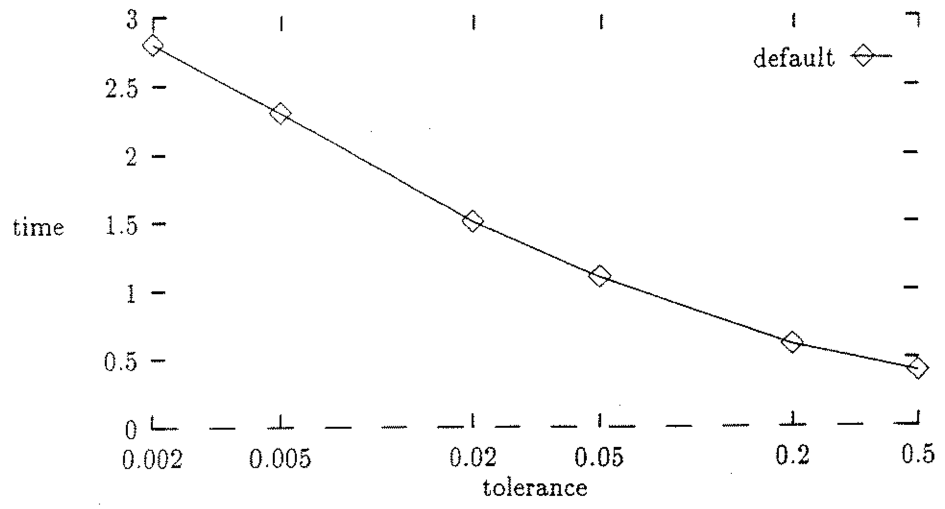


Figure 7.30: Tolerance, 3xtree intersection, 100 tool movements, 100 intersection tests per tool movement

2 is out of line. The constant overhead is part of the reason. However, if the accuracy is low enough that the formula for maximum allowable submovement rotation exceeds the actual tool movement rotation, the program must still use at least one three-axis movement to approximate the tool movement, in effect causing a flattening out of the curve at that end of the scale.

Little is revealed about the 3xtree program in the random test data times in table 7.9. The times are all so small as to be unreliable. However, as figure 7.30 shows the data from table 7.10, the intersection curve appears close to flat, indicating polylogarithmic or high order root of tolerance.

The real test cases provide similar information. They are plotted in figures 7.31 and 7.32. In each, the curve is curved gently upward indicating polylogarithmic or high order root behavior. The plot of z6324r_def shows the highest curvature, but even this is better than square root performance.



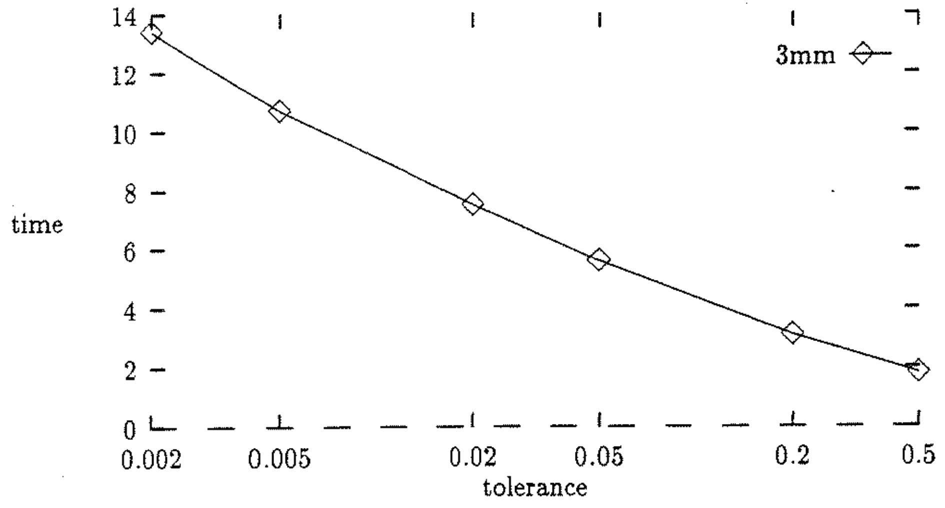
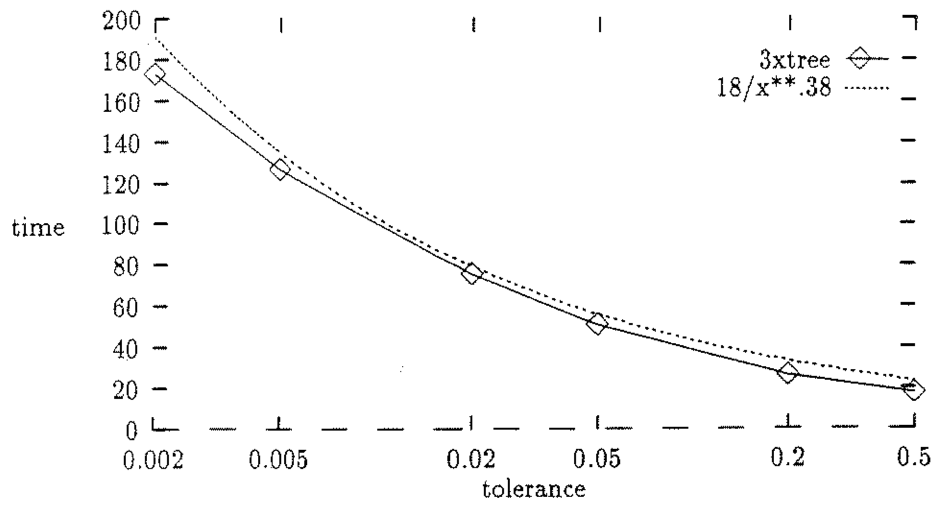


Figure 7.31: Tolerance on door_small, default and 3mm, 3xtree intersection



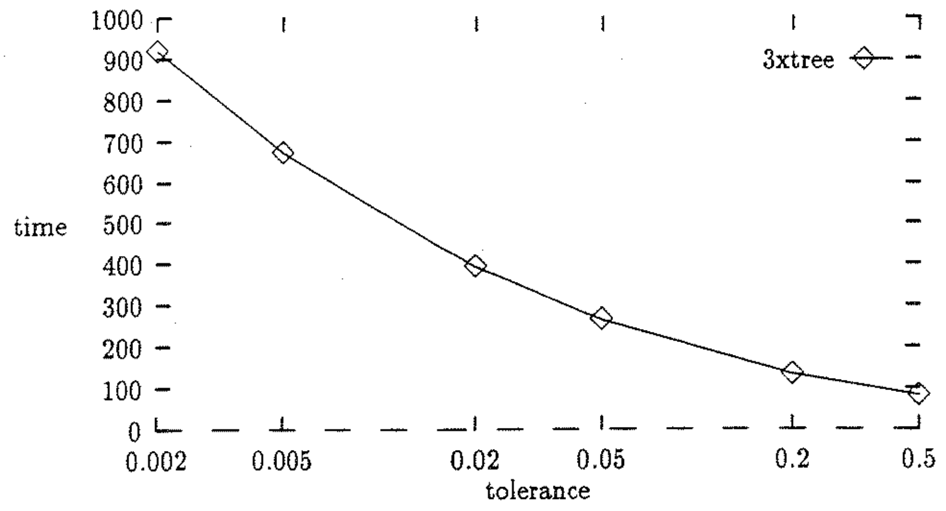


Figure 7.32: Tolerance on z6324r, default and 1.5mm, 3xtree intersection

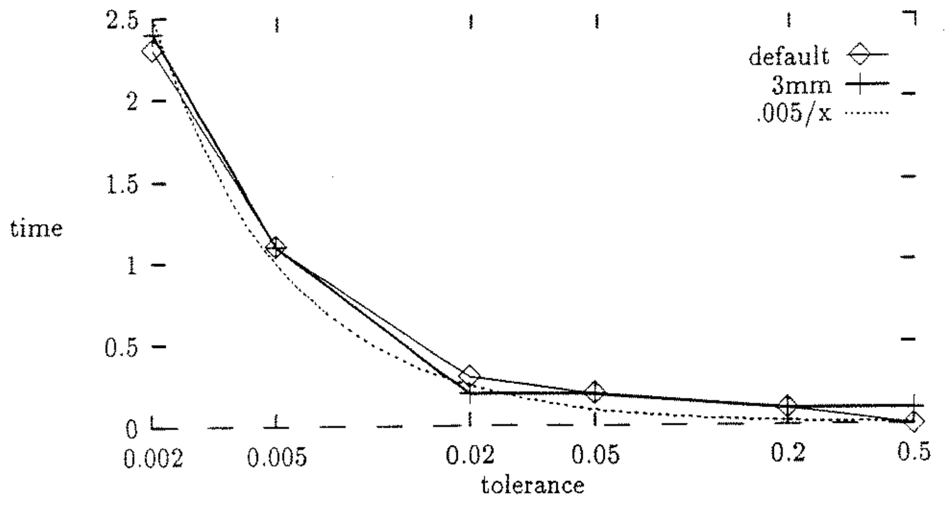


Figure 7.33: Tolerance on door_small, default and 3mm, 3xtree preprocessing

The preprocessing for 3xtree should increase at a linear rate, as previously discussed. The random test data times are too small to get much feedback from. The largest value, in table 7.10, is only 2.5, requiring values of .25 and .025 to show a linear trend. These values are way too small to expect accurate timing information. In the door_small test files, the preprocessing figures are once again too small at the lower end to judge reliably, although they do support the trend. In the higher accuracy range, the numbers appear to grow linearly. Plotting $.005/x$ alongside in figure 7.33 shows that overhead factors are important in the region we are examining. In the graphs of z6324r preprocessing (figure 7.34), we see that the curve $.027/x$ fits very well, suggesting the linear trend we should see. At the low end, we can once again see the overhead factors.

One thing we can conclude from this data is that preprocessing is an insignificant cost for both the 3x and 3xtree programs at these levels of accuracy. At high enough accuracy levels, the preprocessing will swamp intersection time for any problem, given

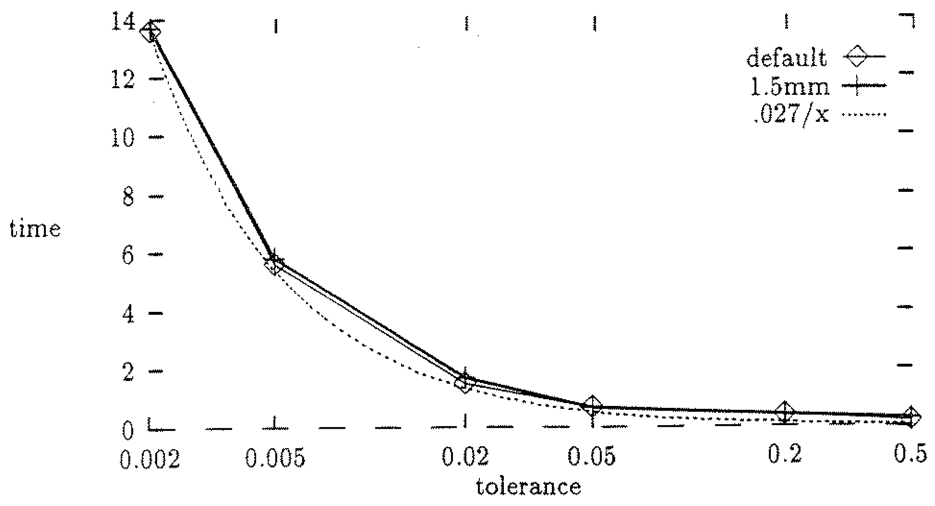


Figure 7.34: Tolerance on z6324r, default and 1.5mm, 3xtree preprocessing

the much lower growth rate that intersection exhibits. That possibility also raises the memory issue, discussed in Chapter 3.

poly

The data we could get on the poly program is extremely limited. It was too easy to find tool movements in the test cases that caused it to overflow available memory. The method is very vulnerable to this. A large number of tool movements had to default to three-axis approach in order to make it run at all. The program was totally unable to run on the file z6324r.

In the random tolerance tests, the increase in preprocessing was nowhere near linear. However, we have only two data points there, so we can't really draw any conclusions. If we look at the poly preprocessing data for door_small, the performance is still less than linear, although the poly program has a sizable overhead associated with it. The graphs are in figure 7.35.

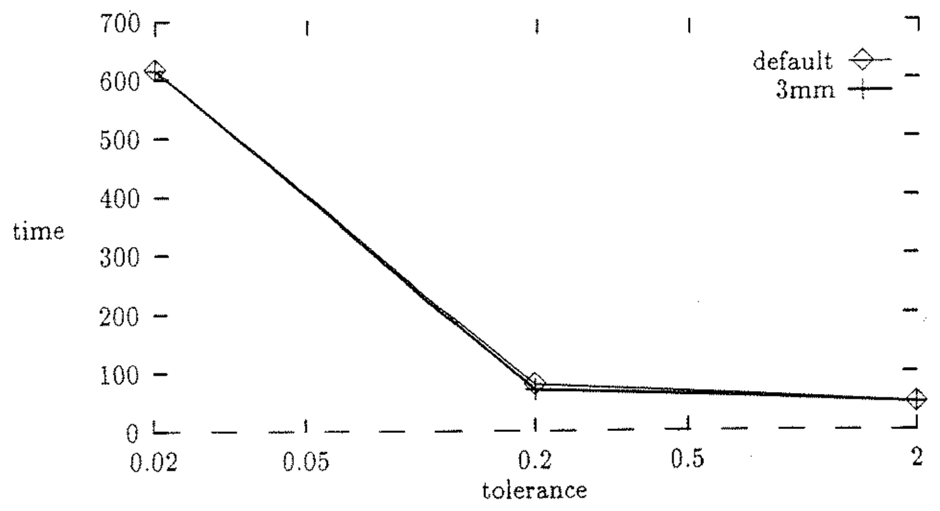


Figure 7.35: Tolerance on door_small, default and 3mm, poly preprocessing

The intersection times for the poly program's random tolerance tests showed minimal increase. Again we only have 2 data points. But, this would indicate that the data is not being efficiently distributed throughout the BSP tree structure. One reason for this is the high threshold of objects allowed in each leaf to prevent memory overflow due to too much subdivision. Memory costs had to be fought at every step.

In the door_small tests, shown in the graphs in figure 7.36, the jump from the second to the third data point in both graphs is nearly linear. The jump from the first data point to the second is very small, which points to high program overhead. The high costs of preprocessing due to the available bounds do not encourage further exploration at this time.

The poly nb program, while not necessarily giving good indications of how the poly program should operate, still offers a view of what we might expect with much better bounds and certainly provides a lower bound for the approach we have taken.

With preprocessing on the random test data, we see the slow growth between

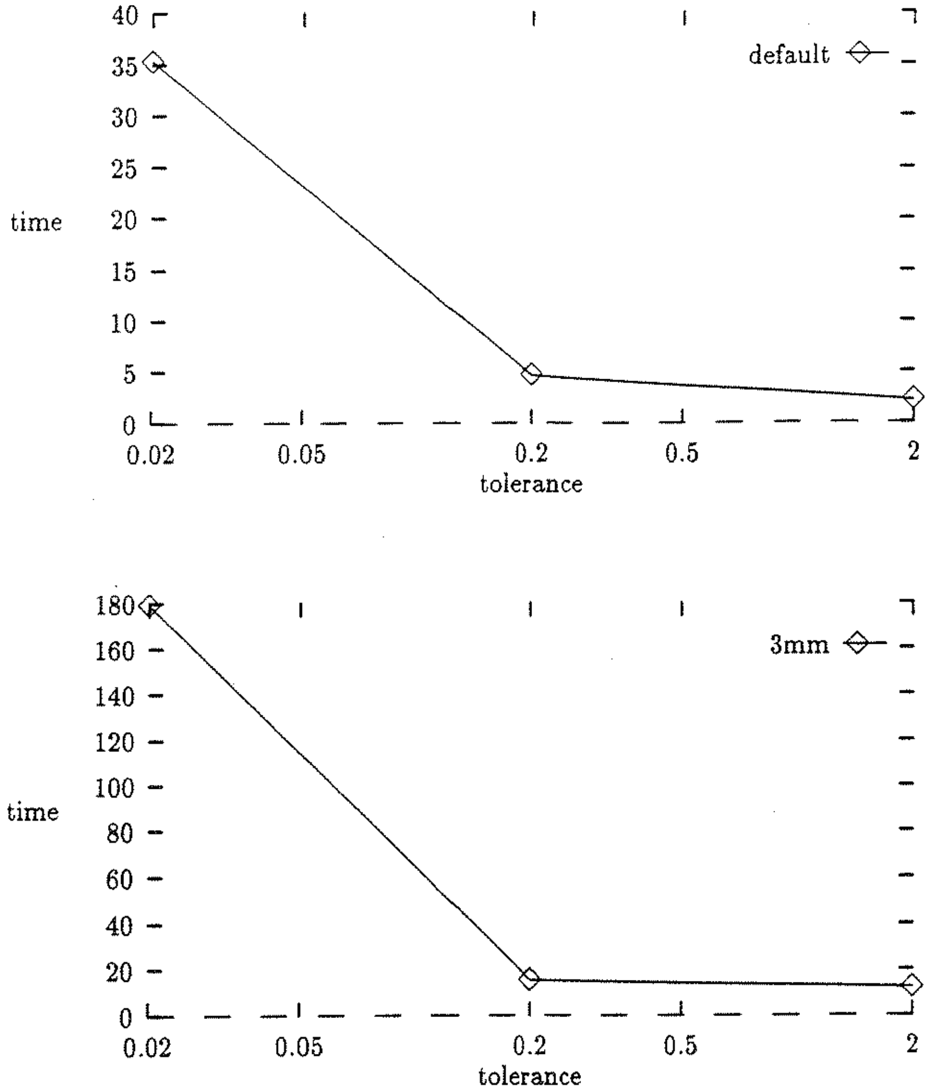


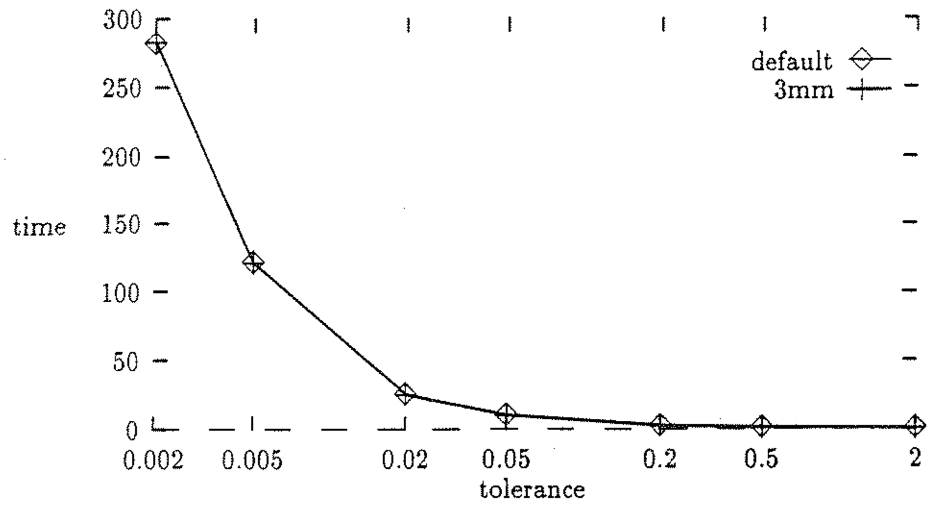
Figure 7.36: Tolerance on door_small, default and 3mm, poly intersection

the first and second points indicative of program overhead. From the second to the third points, there is more than a linear increase. On door_small, the preprocessing times progress nearly linearly, while for z6324r, the times progress at a rate slightly less than linear. The graphs are depicted in figure 7.37.

The random tolerance test data in figure 7.38 shows very flat growth. This implies that the BSP tree is helping out as the model size grows. In the door_small runs, plotted in figure 7.39, the higher tolerance data is very close to linear, flattening out at the lower tolerance levels. The data for z6324r is more interesting. Plotted in figure 7.40, we actually see a decline in intersection times until tolerance .05, when times being to climb. This unexpected effect occurs because the node size limits for the BSP tree are set to allow the large node size required by the poly program. When the model is too small, it doesn't force enough subdivision to take advantage of the BSP tree. As the model increases in size, the tree is used more efficiently, subdividing the model better and reducing the intersection work done in the average node. Finally, the model size grows enough that we start seeing the rise in times that we expect. Further exploration of these effects aren't warranted without improvements in the polygonal bounds.

1d

As mentioned in Chapter 4, the 1d program doesn't have adjustable tolerance.



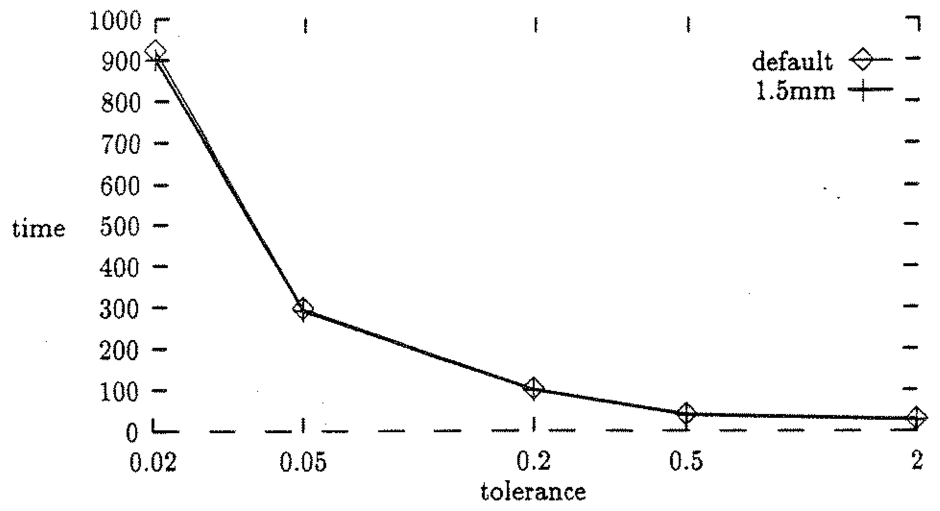
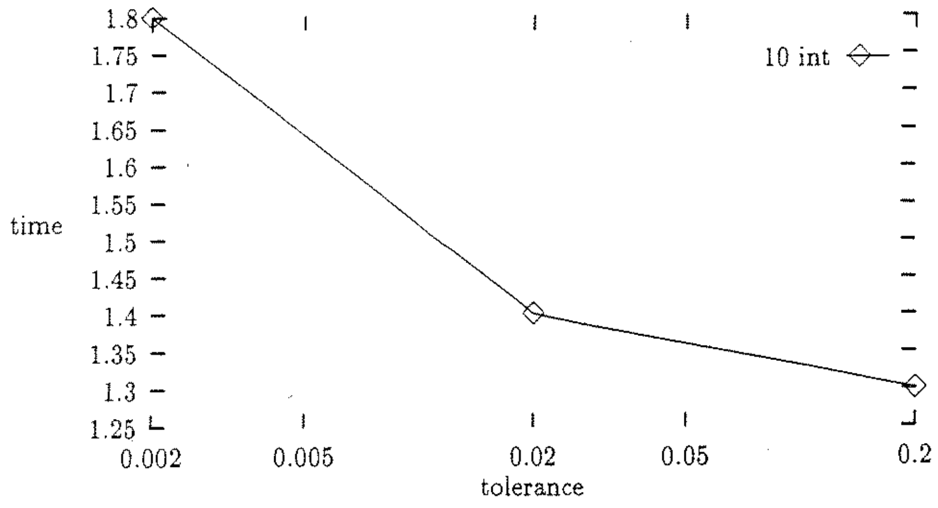


Figure 7.37: Tolerance on door_small at default and 3mm, z6324r at default and 1.5mm, poly nb preprocessing



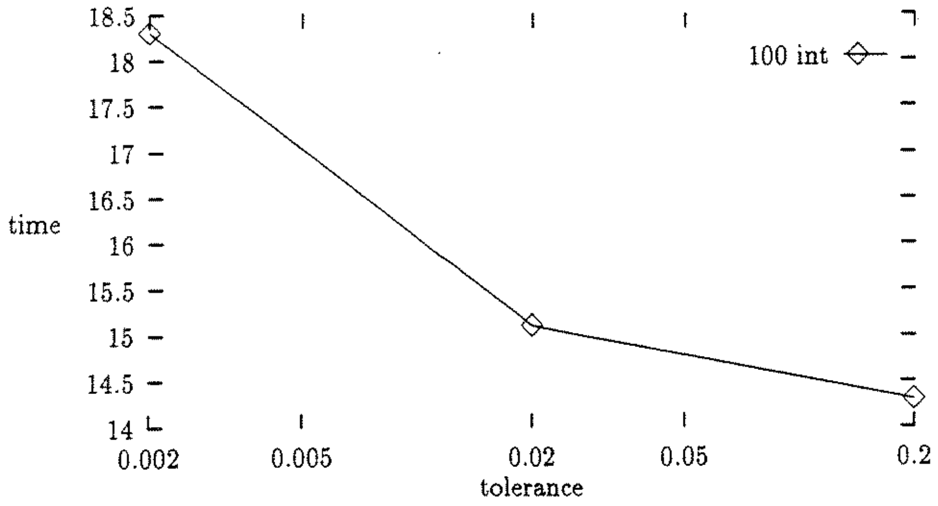
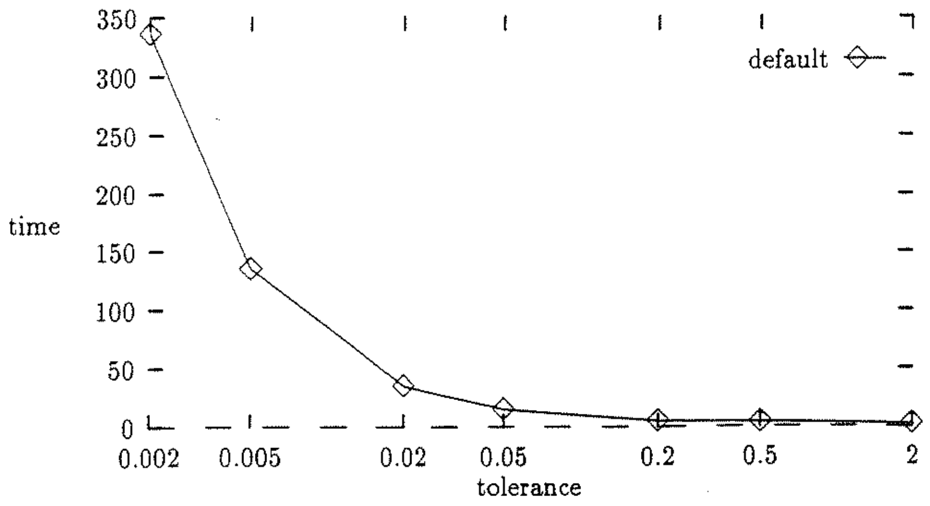


Figure 7.38: Tolerance, poly nb intersection, 100 tool movements, 10 and 100 intersections per tool movement



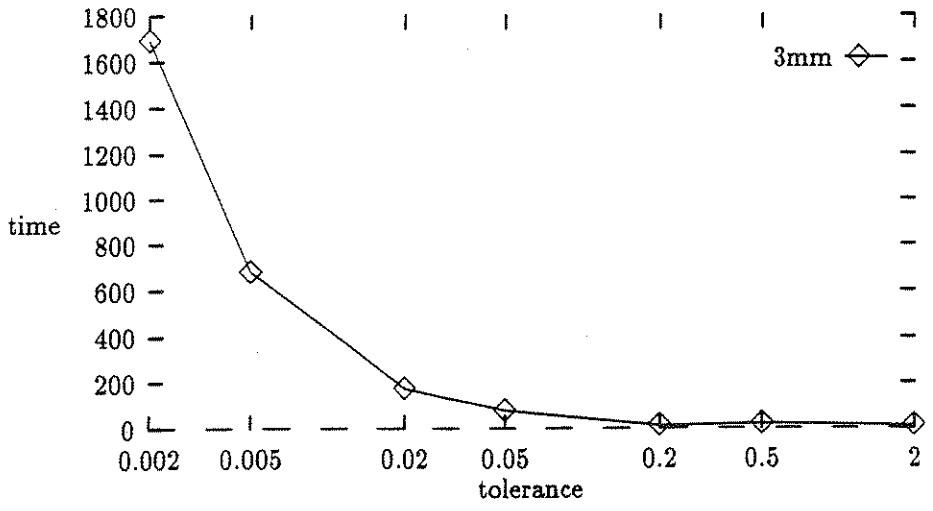
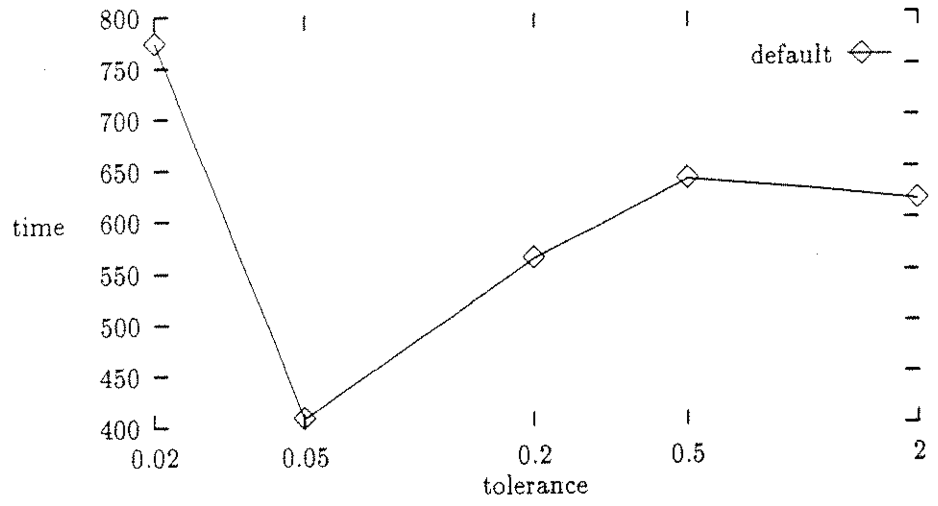


Figure 7.39: Tolerance on door_small, default and 3mm, poly nb intersection



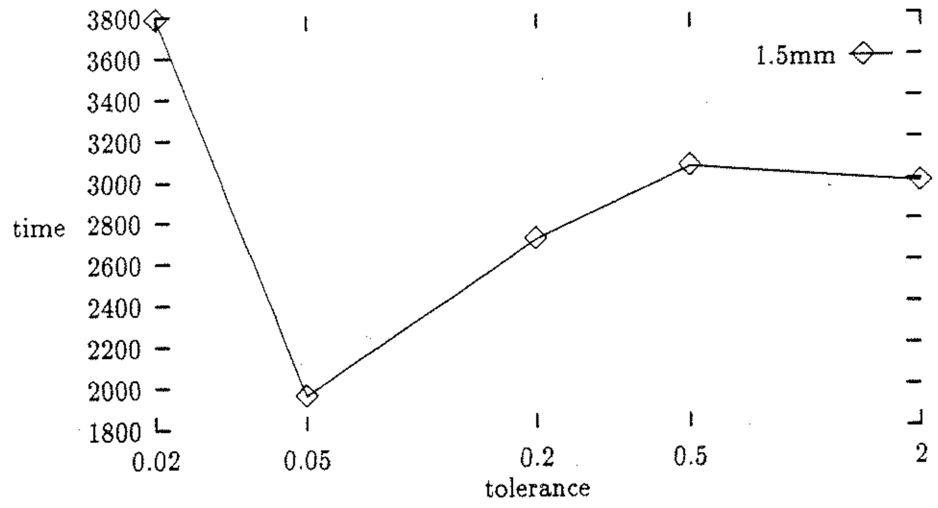


Figure 7.40: Tolerance on z6324r, default and 1.5mm, poly nb intersection

	3-Axis	Step	Poly	1D
Relative Speed	1	3–10	70–200	10–?
Code Complexity	Low (700)	Med (1500)	High (6600)	Med (2000)
Intersection Tolerance	$< O(n^{.4})$	$\ll O(n)$	$\sim O(n)$	—
Preprocessing Tolerance	Small $O(n)$	Tiny	Huge $O(n)$	Tiny
Memory Usage	$O(n)$	Tiny	Huge	Tiny
Flexibility	High	Low	Medium	Low

Table 7.21: Comparison of the Methods

7.3 Conclusions

We have presented several methods by which five-axis verification may be carried out with guarantees on the accuracy of the results. This is accomplished in the paradigm of discrete approximation of a desired part surface by finding the intersection of the five-axis tool movement envelope with a surface vector. All methods presented except the numerical approach can be used at varying tolerances to allow fast overviews and later high accuracy verifications.

We have investigated each method's response in running time to changes in problem size. Also, we have looked at the effects of rotation angle and tool movement length on the performance of each method. Finally, we evaluated the effects on changes in tolerance on each method. Table 7.21 gives an overview of how the methods compare. The speed entries are relative to the performance of the three-axis approximation with the object hierarchy optimization. Code complexity is in lines of C code. Flexibility refers to the relative ease or difficulty in adapting each of the methods to new tool shapes or tool movement equations.

From a practical viewpoint, the three-axis approximation method with bounding hierarchy, outlined in Chapter 3, is the clear winner overall. Its advantages are obvious: there is no complex equation-solving resulting in slow numerical code, implementation is very simple (especially compared to the polygonal approach), and it takes advantage of preprocessing. Certainly for current applications, this is the method of choice for accurate intersection calculation, and subsequently, verification as a whole. In addition, the three-axis approach is very easily extended. Any tool shape for which a three-axis intersection routine exists can be used with this method. The envelopes used in the tree need to be changed some to allow for wrapping the new tool shapes, but the method is very flexible. With a little more work, the method is extensible to other sweeps as well.

The timing tests make clear that the effort expended on preprocessing is pretty much inconsequential compared to intersection time at the tolerance levels tested. This is a very effective use of preprocessing. On the downside, if high enough levels of accuracy are demanded, it is possible that memory may be overrun trying to create the hierarchy tree. In that case, it may be necessary to compromise.

As we mentioned in Chapter 3, several possibilities exist to handle this situation. One, we could calculate data on the fly. This turns out to be an effective method. The memory usage is negligible and the intersection times degrade by about 10movements into smaller ones beforehand, since the memory size requirements can be determined based on tolerance and rotation angle. The final method is to use more than two children per node. This cuts memory usage somewhat but also results in somewhat slower intersection.

The closest competitor is the fast step method. It ranges from 3 to 10 times slower than the three-axis method under varying circumstances. Most importantly, as accuracy demands climb, the fast step program starts to narrow the gap. Also, as the rotation angle increase, the fast step program catches up some. If verification is to be carried out at extremely high accuracy levels, the fast step program may just be able to come out in front.

We have also seen that the z-culling optimization is an effective improvement to the step method. The large step optimization is even more effective. It makes the performance very sublinear in tolerance, outperforming the linear performance of the simpler implementations.

The step method is less amenable to changing the tool shape. The point test is highly dependent on the equations based on a cylinder. A new tool shape requires restructuring the equations and possibly even completely revamping the point test if the interval narrowing algorithm is no longer applicable.

The step method heavily relies on the worst case bounds for minimum finding in the second part of the point tests. These bounds are not very tight in general. Better bounds would be directly realized in speedups in the step method. It might even become the front runner if tight enough bounds could be had.

The polygonal approach has turned out to be the biggest disappointment. On paper it would seem to be the most potentially efficient approach. However, the looseness of the bounds on surface approximation render this approach ineffective.

The fact that the unbounded version did not perform well either leads to the conclusion that the methods used to create the polygonal surface are not very useful in a computational setting. The representation performs worst in cases that should be the easiest, and requires a totally different approach to handle them. A different method of generating triangles as well as improved bounds would be necessary to the success of this method.

However, the overhead involved in the programming the polygonal method is fairly high. The code complexity is unlikely to reduce significantly since any polygonal program will still have to build a mesh, and probably will do it in an adaptive manner. The meshes must be sewn together, although, different generation schemes may be able to eliminate this step. Finally, some method of efficiently finding intersections, such as octrees, is vital and requires both data structure construction routines as well as intersection routines. While not ideal, the programs used do give some idea of how complex any polygonal program will have to be.

The numerical approach is a mixed bag. In some examples it is as fast as the large step method, while in others, especially the real data tests, it is incredibly slow. This is even more true when bounds on the rad function are used. As a practical matter, this method is a curiosity without much improved bounds on root finding in the equations. Adding the bounds as described to the remaining equations will only make matters worse. Another downside is that both the equations and algorithm are almost completely dependent on the use of a flat-end cylinder. As with the step method, changing the tool shape would require changes in the equations used, and very possibly necessitate a new search algorithm. At the very least, it would be a significant project.

A simple, if potentially slow, method of finding the intersection between a five-axis tool movement and a line is to generate a set of still tool placements and find the intersection of the line with each one in turn. As the density of tool placements increases, it evolves into the numerical approach that we described. It turned out that finding a bound for this model was significantly more difficult than for the three-axis approximation. However, given the similarity of this to the numerical approach, bounding this model may give insight into improving the bounds for the numerical approach.

The result of this thesis is to offer a method of finding the intersection of a swept cylinder with a line that can be guaranteed to a desired level of accuracy, is fast, simple to implement, and flexible. In addition we have explored other avenues of solving the same problem and shown them to be less desirable at this time. The information we have provided applies not only to the obvious application of NC verification, but also to any application in which the intersection of a swept object and a line must be found with accuracy guarantees.

7.4 Open Problems

While any work tries to solve every aspect of a problem, some questions inevitably remain unanswered. In addition, a solution to a problem may raise new and interesting questions to explore. This dissertation is no different. We have done an extensive, though not exhaustive, look at methods for finding intersections with guaranteed accuracy between lines and five-axis tool envelopes. In the process, several unsolved questions arise.

First, the method we employed to guarantee global root-finding and minimum location is very conservative. In some cases it works rather well. But in others, the bounds are overly cautious and cause significant slowdowns in the efficiency of all but the three-axis approach. The step method in particular can benefit from improving the bounds on searching the second equation of the point test. The polygonal and numerical approaches would also clearly benefit.

The step method may also benefit from taking advantage of curve coherence. The parameters change very little from step to step and equation searching might be facilitated by this fact.

As we have already pointed out, the polygonal method we have presented is not very effective. The testing results from the unbounded version seem to suggest that even improving the bounds on the equations involved will not be sufficient to make our polygonal approach viable. However, the poor results are also tied to the surface representations used in this dissertation. Finding better parameterizations may prove fruitful despite the likely complexity of the programs.

Related to better bounds for equation searching, the numerical approach may benefit from a deeper look at the static cylinder idea. If a bound can be derived for the static cylinder approach, it may lead to a better understanding of the equations involved in the numerical approach, and subsequently, better search methods.

Finally, even though the three-axis approximation is the best of the methods as presented, it too can benefit from further research. Although the bound derived here is good, it is not as tight as possible. The tolerance distance from a point on the surface of the approximation is measured to the corresponding position in the true tool movement. However, the real location of this point is not necessarily on the surface of the true tool envelope and the envelope itself may be closer still. A better bound would mean even fewer three-axis submovements could model a five-axis tool movement, resulting in faster intersection times.

References

[1] R. O. Anderson. "Detecting and Eliminating Collisions in NC Machining". *Computer Aided Design*. Vol 10, No 4, 1978. pp 231–237.

[2] P.R. Atherton, C. Earl, and C. Fred. "A Graphical Simulation System for Dynamic Five-Axis NC Verification". *Proc. Autofact*. SME, Dearborn, Mi., Nov 1987. pp 2-1–2-12.

[3] Denis Blackmore and Ming C. Leu. "Analysis of Swept Volume via Lie Groups and Differential Equations". *International Journal of Robotics Research*. 1992.

[4] Denis Blackmore, Ming C. Leu, and Wen Wang", "Classification and Analysis of Robot Swept Volumes". *Japan-USA Symposium on Flexible Automation*. 1992.

[5] Ki-Yin Chang and Erik D. Goodman. "A Method for NC ToolPath Interference Detection for a Multi-Axis Milling System". *Control of Manufacturing Processes*. Proceedings of Winter Annual Meeting of the ASME, 1991.

[6] I.T. Chappel. "The Use of Vectors to Simulate Material Removal by Numerically Controlled Milling". *Computer Aided Design*. Vol 15, No 3, May 1983. pp 156–158.

[7] B. K. Choi, C. S. Lee, J. S. Hwang, and C. S. Jun. "Compound Surface Modelling and Machining". *Computer-Aided Design*. Vol 20, No 3, April 1988.

[8] Robert L. Drysdale, Robert B. Jerard, Barry Schaudt, and Ken Hauck. "Discrete Simulation of NC Machining", *Algorithmica*. No 4, 1989. pp 33–60.

[9] Robert L. Drysdale, Jerome L. Quinn, Kamran Ozair, and Robert B. Jerard. "Discrete Surface Representations for Simulation, Verification, and Generation of Numerical Control Programs". *Proceedings of NSF Design and Manufacturing Systems Conference*. 1991.

[10] L. P. Eisenhart. *Differential Geometry*. Ginn and Co., 1909.

[11] I. D. Faux and M. J. Pratt. *Computational Geometry for Design and Manufacture*. Halsted Press, New York, 1979.

[12] R. Frishdal, K. P. Cheng, D. Duncan, and W. Zucker. "Numerical Control Part Program Verification System". *Proceedings of the Conference on CAD/CAM Technology in Mechanical Engineering*. MIT Press, Mar 1982. pp 236–254.

[13] M. A. Ganter and J. J. Uicker, Jr. "Dynamic Collision Detection Using Swept Solids". *Journal of Mechanisms, Transmissions, and Automation in Design*. Vol 108, 1986. pp 549–555.

[14] ed. Andrew S. Glassner. *Graphics Gems*. Academic Press, New York, 1990.

[15] ed. Andrew S. Glassner. *An Introduction to Ray Tracing*. Academic Press, Boston, 1989.

[16] Allan Hansen and Farhad Arbab. "Fixed-Axis Tool Positioning with Built-in Global Interference Checking for NC Path Generation". *IEEE Journal of Robotics and Automation*. Vol 4, No 8, 1988. pp 610–621.

[17] W. A. Hunt, and H. B. Voelcker. *An Exploratory Study of Automatic Verification of Programs for Numerically Controlled Machine Tools*. Production Automation Project Tech Memo No. 34, University of Rochester, Jan 1982.

[18] Robert B. Jerard, Jennifer Angleton, Robert L. Drysdale, and Peter Su. "The Use of Surface Point Sets for Generation, Simulation, Verification, and Automatic Correction of NC Machining Programs". *Proceedings of NSF Design and Manufacturing Systems Conference* Society of Manufacturing Engineers, Jan 1990. pp 143–148.

[19] Robert B. Jerard, Robert L. Drysdale, and Ken Hauck. "Geometric Simulation of Numerical Control Machining". *ASME International Computers in Engineering Conference*. San Fransisco, 1988.

[20] Robert B. Jerard and Robert L. Drysdale. "Methods for Geometric Modeling, Simulation and Spatial Verification of NC Machining Programs". *Product Modeling for Computer Aided Design*. ed. M.J. Wozny, J.U. Turner, and J. Pegna. North Holland, 1991.

[21] Robert B. Jerard, Robert L. Drysdale, Ken Hauck, Barry Schaudt, and John Magewick. "Methods for Detecting Errors in Numerically Controlled Machining of Sculptured Surfaces". *IEEE Computer Graphics & Applications*. Vol 9, No 1, 1989. pp 26–39.

[22] Robert B. Jerard, S.Z. Hussaini, Robert L. Drysdale, and Barry Schaudt. "Approximate Methods for Simulation and Verification of Numerically Controlled Machining Programs". *Visual Computer*. No 5, 1989. pp 329–348.

[23] Xiaoxia Li, *Automatic Tool Path Generation for Numerically Controlled Machining of Sculptured Surfaces*. Ph.D. dissertation. Mechanical Engineering Dept. University of New Hampshire, May 1993.

[24] Ashish P. Narvekar. *Representation and Application of Swept Solids for Numerically Controlled Milling*. Masters thesis. SUNY Buffalo, 1991.

[25] J. H. Oliver and E. D. Goodman. "Color Graphic Verification of NC Milling Programs for Sculptured Surface Parts". *First Symposium on Integrated Intelligent Manufacturing*. ASME Winter Annual Meeting, Anaheim, Ca. 1986.

[26] J. H. Oliver and E. D. Goodman. "Direct Dimensional NC Verification". *Computer-Aided Design*. Vol 22, No 1, 1990.

[27] Kamran Ozair. *NC Machining Simulation and Verification Using Triangles Rather Than Points*. Honors Thesis, Dartmouth College, 1990.

[28] Joseph Pegna. *Variable Sweep Geometric Modeling*. Ph.D. thesis. Stanford University, 1987.

[29] William H. Press, Brian P. Flannery, Saul A. Teulosky, and William T. Vetterling. *Numerical Recipes*. Cambridge University Press, New York, 1986.

[30] Kumaraguru Sambandan. "Graphics Simulation and Verification of Five-Axis NC Machining". Technical Report 60, Cornell University, 1988.

[31] Kumaraguru Sambandan. *Geometry Generated by sweeps of Polygons and Polyhedra*. Ph.D. thesis. Cornell University, 1990.

[32] G. B. Thomas, Jr. and R. L. Finney. *Calculus and Analytic Geometry*. Addison-Wesley, Reading, Mass, 1988.

[33] T. Van Hook. "Real-Time Shaded NC Milling Display". *Computer Graphics* (proc. SIGGRAPH). Vol 20, No 4, Aug 1986. pp 15–20.

[34] H. B. Voelcker and W.A. Hunt. "The Role of Solid Modeling in Machining - Process Modeling and NC Verification". SAE technical Paper 810195, 1981.

[35] W. P. Wang and K. K. Wang. "Geometric Modeling for Swept Volume of Moving Solids". *IEEE Computer Graphics & Applications*. Vol 6, No 6, 1986. pp 8–17.

[36] John D. Weld and Ming C. Leu. "Geometric Representation of Swept Volumes with Application to Polyhedral Objects". *International Journal of Robotics Research*. 1990.