FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES

by

Craig L. Cole

January 8, 2004

A thesis submitted to the
Faculty of the Graduate School of
State University of New York at Buffalo
in partial fulfillment of the
requirements for the degree of

Master of Science

Department of Mechanical & Aerospace Engineering
State University of New York at Buffalo
Buffalo, New York 14260

For my Grandmother,

who taught me

to split 8's against an ace

every time.

Acknowledgements

Great thanks go to my thesis advisor Dr. John L. Crassidis for his help, for his guidance and for giving me the opportunity to explore this very exciting field of aerospace engineering. His ability to communicate difficult concepts and paint pictures of their historic significance made it impossible to be disinterested. I hope in our professions we will cross paths often.

I would also like to thank Dr. D. Joseph Mook and Dr. Tarun Singh for their advice with my thesis and for being part of my thesis defense committee. Additional thanks go to Dr. William Rae and Dr. Christina. L. Bloebaum for their help paving my way back to graduate school.

The year 2003 will be remembered by those who know me as the year I reclaimed my direction in life, and it would not have been possible without the never-ending support of my family, one very special friend and two cats. Sir Isaac Newton once said, "If I have seen further than others, it is because I have stood on the shoulders of giants." To me, they are giants. Thank you.

Contents

List of Figures

List of Tables

Abstract

A current method by which star trackers determine attitude is to match the angles between stars within its field of view to angles stored in a catalog. If the angle can be matched to one pair of stars, the attitude of the star tracker can be easily determined. However, the measurement of the angle is likely to include an error, and so the angle can only be known to lie within a certain range. What is likey to happen is, after comparing the selected angle to the catalog of angles, more than one possible pair of stars can be the correct solution. Narrowing down to one solution requires employing many angles within the field of view, called "pivoting," which can be time consuming and does not always yield a solution.

The method presented here attempts to match spherical triangles made from stars within the field of view to spherical triangles stored within a catalog. By using both the area and polar moment properties of the spherical triangle, the range of possible solutions is very quickly narrowed, few pivots to other spherical triangles are required, and is far more likely to yield the correct solution than the angle method.

1. Star Tracker Attitude Determination

1.1. Introduction

A very important device on any spacecraft is the one which determines the spacecraft's attitude. There are many different methods by which attitude can be determined, but one of the mostly widely used is the star tracker. It is a device capable of imaging stars and does its best to determine attitude by identifying the stars within its field of view. If it can identify at least two stars, the attitude of the star tracker can be determined, and then transformed to attitude of the spacecraft to which it is attached. The spacecraft can use this information to check its heading, make course changes or even recover from a "lost-in-space" condition.

Other methods for determining attitude exist, including sun sensors and magnetometers, but cannot report attitude with the precision a star tracker can. At best, a sun sensor-magnetometer system can only report attitude within 0.1 deg, while star trackers are capable of arc-second accuracy. Often, on high-budget missions, these other sensors will complement or back-up a star tracker. On missions with tighter budgets, the star tracker will be trusted to determine attitude completely on its own.

The technology behind star trackers has changed a lot over the years. Evolving from gimbaled to direct-head, the latest star trackers now use charge-coupled devices (CCD) for imaging, which offer high accuracy. Still, they are not perfect devices and measurement

error still exist. Even the small amount of error that exists in the position measurement of the stars within the star tracker's field of view makes identifying the stars a very challenging problem. There are many different methods for identifying the stars within the field of view. One well known method uses the angular separation between stars in the star tracker field of view and tries to match them to an angle within a catalog of angles between stars[1]. It is called an "angle" method and has been the basis for many star recognition algorithms. It does not rely on other sensors for assistance, and can be used in a "lost-in-space" condition.

Other methods for identifying the stars include using the magnitude of the stars in the field of view[2]. While magnitude is very useful for identifying stars, measuring the magnitude of stars using CCD's is not simple, requiring knowledge of the CCD sensitivity to different spectra of light, for example. Other methods attempt to use "a priori" information to help the star tracker to determine the stars within its field of view. [1, 3, 4, 5, 6, 7, 8] Methods based on knowing the attitude of the spacecraft at an earlier point in time would not be suitable for "lost-in-space" conditions. Even neural networks are being tested for their potential to recognize stars within the star tracker's field of view[9]. It can be seen that the "perfect" star pattern recognition algorithm does not yet exist, and the field is wide open for new methods.

The method presented here is called the "spherical triangle method." It makes spherical triangles out of the stars in the field of view and uses a catalog of spherical triangles to find a match. It also has its roots in the angle method and so will be used as a control to which to compare the spherical triangle method.

To make the testing of both methods realistic, a "typical" star tracker has been chosen as a model. The Ball CT-601 star tracker has an 8×8 degree field of view, and can see down to magnitude 6.0 stars [3, 10]The testing model created here assumes an 8 degree circular field of view for simplicity, but the star pattern recognition algorithms tested can

be applied to square fields of view.

1.2. Motivation

The importance of star trackers on spacecraft is hard to overestimate. Errors in navigation can result in damage to or the complete loss of a spacecraft. An ideal star tracker would be able to report attitude instantly without chance of error in lost-in-space conditions without aid from other sensors. In addition it would require little in the way of computer resources such as storage and CPU speed. These are certainly conflicting constraints, and decisions have to be made when designing the star tracker as to what is most important.

For this thesis, rate of success and speed were given the highest priorities in the design of the algorithms for the spherical triangle method. A star tracker's ability to report attitude quickly is desirable in situations where the spacecraft is tumbling; if the result lags too far behind the actual position of the spacecraft, the spacecraft becomes very difficult to control. The idea behind the spherical triangle method is that by using more than one property to recognize a pattern of stars, it is more likely to reach the correct solution, and do it very quickly. The angle method, also explored in this thesis, can only use one property, the angle itself, for pattern matching, and as a result will not arrive at a solution as quickly. Time measurements comparing both methods will be made to show this.

The penalty for success and speed is storage. It will be seen that the database used with the spherical triangle method is quite large when compared to that required for the angle method. It is only large by spacecraft standards, however; the database would easily fit onto a keychain-sized flash drive. The computer hardware used in spacecraft are designed to withstand the harsh environment of space, and as a result lag years behind the current state of the art in processor speed and storage. The technology continues to evolve, however, and tasks that are too demanding of spacecraft hardware today will likely be practical in the

not too distant future.

1.3. Outline of Thesis

In chapter 2, the mathematics required of both the angle and spherical triangle method will be explained. The properties of both and the effects of error in their measurement will also be explored.

For the angle and spherical triangle methods to work, databases have to be constructed. In chapter 3, the methods by which these catalogs are created is shown in detail.

In order to test either method, the simulation under which each method will be tested has to be created. The reasoning and methods for testing are discussed in Chapter 4.

Chapter 5 goes into detail explaining the algorithms used for both the angle and spherical triangle methods. A wide variety of data structures are employed, the object being to make the code as fast as possible.

The results for testing are given in chapter 6. Success and failure rates are graphed and the time results reviewed. A method for optimizing each method's results are also discussed.

The conclusions reached and suggestions for further study are given in chapter 7.

2. Method Mathematical Analysis

2.1. Introduction

In this chapter, the methods and mathematics behind the angle method and the spherical triangle method are explained. For the angle method, the equation for determining the angle between stars from its vectors, and the standard deviation of the measurement will be required. For the spherical triangle method, the area and polar moment will be required, as well as the standard deviation for both.

2.2. Angle Method

There are many methods by which a star tracker can determine what stars are within its field of view, many of which are proprietary. A well-known method is to try matching the angle of separation between pairs of stars within the field of view to a catalog containing all of the angles between stars in the celestial sphere which can fit within the star tracker's field of view[1]. For instance, a circular 8 deg field-of-view star tracker would require a catalog of all the angles between stars 8 deg or less. If the measured angle, such as shown in Figure 2.1 can be matched to a single angle in the catalog, the stars that make up the angle would be known, and so then would the attitude of the star tracker.

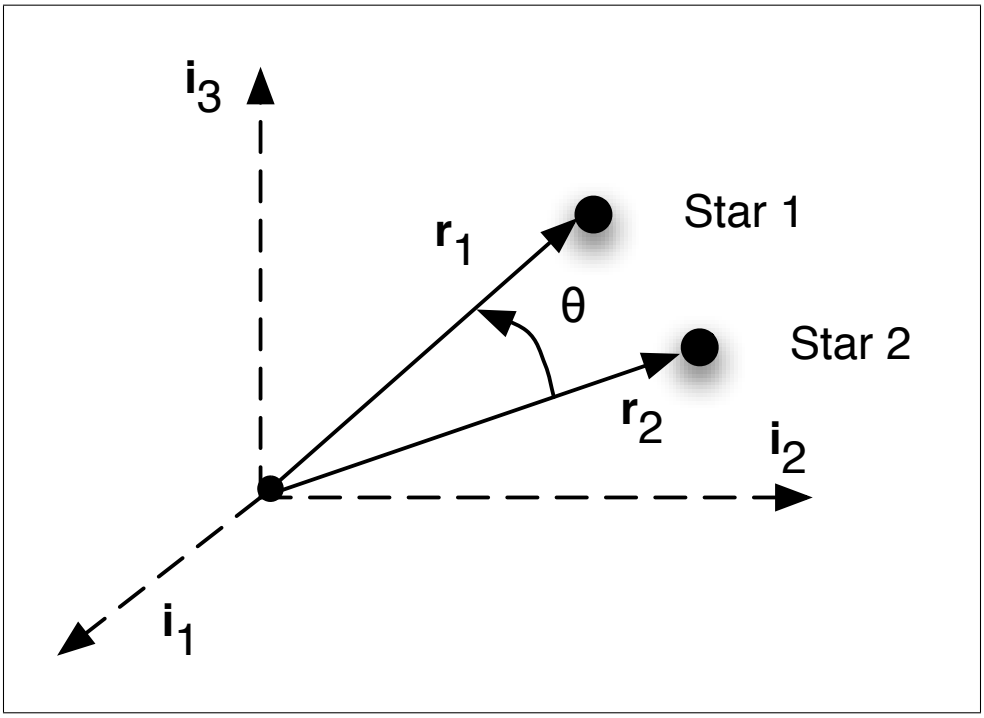

Figure 2.1.: Angle Between Stars Used To Determine Attitude

2.2.1. Angle Between Two Stars

The angle between the vectors pointing to the stars is:

$$\theta = \cos^{-1}(\mathbf{r}_1 \cdot \mathbf{r}_2) \tag{2.1}$$

where $\mathbf{r}_1$ and $\mathbf{r}_2$ are unit vectors pointing to each star. The problem is that the angle measured between the stars within the field of view of the star tracker will likely contain a certain amount of error, and must not be ignored. If the measurement follows a normal distribution, its standard deviation can be determined and used to establish a range within which the true measurement is likely to lie. For instance, if the range is chosen to be the measurement angle ± 3 times the standard deviation, 3σ, the true measurement is expected to be within this range 99.7% of the time.

2.2.2. Standard Deviation of Angle Between Stars

The position error made by a typical modern-day star tracker nearly follows a normal distribution and has a standard deviation of 0.87 microradians of arc. [11] What will be needed for the angle method is the standard deviation of the angle between two stars when each star measurement possesses the position error described.

To obtain the standard deviation, the variance of an attitude-independent measurement is taken, so it must be shown that the dot product of two vectors is attitude independent. Beginning with the standard coordinate transformation equation:

$$\mathbf{b}_i = A\mathbf{r}_i \tag{2.2}$$

where $\mathbf{r}_i$ is the direction of the star in Earth-Centered Interial (ECI) coordinate system, A is the direction cosine matrix, which is orthogonal and proper, and $\mathbf{b}_i$ is the direction of the

star in the star tracker coordinate system. Taking the dot product of two body observations gives

$$\begin{aligned} \mathbf{b}_i^T \mathbf{b}_j &= \mathbf{r}_i^T A^T A \mathbf{r}_j \\ &= \mathbf{r}_i^T \mathbf{r}_j \end{aligned} \tag{2.3}$$

Equation (2.3) shows that the dot product is an attitude-invariant measurement. The attitude-independent approach for determining the variance of a measurement can safely be used. Consider the two body measurements:

$$\tilde{\mathbf{b}}_i = A\mathbf{r}_i + \mathbf{v}_i \tag{2.4}$$

$$\tilde{\mathbf{b}}_j = A\mathbf{r}_j + \mathbf{v}_j \tag{2.5}$$

where noise is now added to the measurement and $\mathbf{v_i}$ and $\mathbf{v_j}$, are uncorrelated. Define the following effective measurement:

$$\begin{aligned} z &\equiv \tilde{\mathbf{b}}_i^T \tilde{\mathbf{b}}_j \\ &= \mathbf{r}_i^T \mathbf{r}_j + \mathbf{r}_i^T A^T \mathbf{v}_j + \mathbf{r}_j^T A^T \mathbf{v}_i + \mathbf{v}_i^T \mathbf{v}_j \end{aligned} \tag{2.6}$$

Since $\mathbf{v_i}$ and $\mathbf{v_j}$ are uncorrelated, then

$$E\{z\} = \mathbf{r}_i^T \mathbf{r}_j \tag{2.7}$$

where E denotes expectation. Define the following variable

$$\begin{aligned} p &\equiv z - E\{z\} \\ &= \mathbf{r}_i^T A^T \mathbf{v}_j + \mathbf{r}_j^T A^T \mathbf{v}_i + \mathbf{v}_i^T \mathbf{v}_j \end{aligned} \tag{2.8}$$

Then taking $E\{p^2\}$ yields

$$\begin{aligned}\sigma^2 &\equiv E\{\sigma^2\} \\ &= \mathbf{r}_i^T A^T R_j A \mathbf{r}_i + \mathbf{r}_j^T A^T R_i A \mathbf{r}_j + \text{Trace}(R_i R_j) \\ &= \text{Trace}(A\mathbf{r}_i \mathbf{r}_i^T A^T R_j) + \text{Trace}(A\mathbf{r}_j \mathbf{r}_j^T A^T R_i) + \text{Trace}(R_i R_j)\end{aligned} \tag{2.9}$$

where $E\{p^2\} = \text{Trace}(R)$ was used in the derivation. The last term is typically higher-order, which can be ignored. Also if $A\mathbf{r}_i$ and $A\mathbf{r}_j$ are replaced by $\tilde{\mathbf{b}}_i$ and $\tilde{\mathbf{b}}_j$ respectively, then the variance is given by:

$$\begin{aligned}\sigma_p^2 &\equiv E\{p^2\} \\ &= \tilde{\mathbf{b}}_i^T R_j \tilde{\mathbf{b}}_i + \tilde{\mathbf{b}}_j^T R_i \tilde{\mathbf{b}}_j + \text{Trace}(R_i R_j) \\ &= \text{Trace}(\tilde{\mathbf{b}}_i \tilde{\mathbf{b}}_i^T R_j) + \text{Trace}(\tilde{\mathbf{b}}_j \tilde{\mathbf{b}}_j^T R_i) + \text{Trace}(R_i R_j)\end{aligned} \tag{2.10}$$

Therefore, no errors are introduced by this substitution, and the variance is completely independent of the attitude matrix A. Schuster [12] has shown that nearly all of the probability of the errors is concentrated on a very small area about the direction of $A\mathbf{r}_i$, so that the sphere containing that point can be approximated by a tangent plane, characterized by

$$\begin{aligned}\tilde{\mathbf{b}}_i &= A\mathbf{r}_i + v_i \\ v_i^T A\mathbf{r}_i &= 0\end{aligned} \tag{2.11}$$

where $\tilde{\mathbf{b}}_i$ denotes the ith measurement and the sensor error v_i is approximately Gaussian, which satisfies

$$\begin{aligned}E\left(v_i\right) &= \mathbf{0} \\ E\left(v_i v_i^T\right) &= \sigma^2 \left[I - (A\mathbf{r}_i)(A\mathbf{r}_i)^T\right]\end{aligned} \tag{2.12}$$

If $R_i = \sigma_i^2 I$ and $R_j = \sigma_j^2 I$, which is typical of star trackers, then

$$\sigma_p^2 = \sigma_i^2 + \sigma_j^2 + 3\sigma_i^2\sigma_j^2 \tag{2.13}$$

Furthermore, if $\sigma_i^2 = \sigma_j^2 \equiv \sigma^2$ then $\sigma_p^2 = 2\sigma^2 + 3\sigma^4$. If σ is small, then the $3\sigma^4$ term can be excluded. The variance of the angle between the stars is then:

$$\sigma_p^2 = 2\sigma^2 \tag{2.14}$$

The standard deviation of the angle measurement is then

$$\sigma_p = \sqrt{2}\sigma \tag{2.15}$$

The σ bound chosen has a great effect on the results of both the angle and spherical triangle method. If the correct solution lies outside the bounds of measurement, neither method will find the solution, or worse, arrive at an incorrect solution. If the σ bound is set high, it is less likely an incorrect solution will be reached, since it is more likely the correct solution lies with the σ bound. However, if the σ bound is set too high, too many possible solutions will exist for each angle and spherical triangle in the field of view and a single solution will not be reached. For this thesis, a $3\sigma_p$ is used, meaning the probability that the correct angle will be within the range of the measured angle will be 99.7%.

2.3. Angle Pivot

If there exists more than one possible solution to a measured angle, the method by which the correct solution can be approached is called "pivoting." After all the possible solutions to the first angle are determined, a second angle within the field of view is selected such

that it shares one star in common with the first angle. Once all the potential solutions to the second angle are found, the stars that make up both angles are examined, and any angles that do not share one star in both lists are rejected. If the number of solutions for each angle isn't reduced to one after the elimation, another pivot is made; a third angle is chosen such that it has at least one star in common with the second angle, and the possible solutions for the second and third angles that do not share a common star are rejected. The pivoting process continues until a single solution is reached, or the star tracker runs out of angles to which it can pivot. If it runs out of angles before obtaining a single solution, the result is inconclusive. Ideally, the angles in the field of view should be ordered so that the angle with the fewest number of solutions is examined first. The second angle chosen should be the angle with the next-fewest number of combinations that shares one star in common with the first angle and so on. This method reduces the number of combinations that need to be examined and makes it more likely to reach a solution quickly. The effect of pivoting in this manner is shown in Figure 2.2. Despite the number of solutions per angle increasing, hopefully, the number of solutions after comparing stars between each, the number of solutions will drop eventually to one. In the figure, it shows that 3 pivots were required to reach a singular solution. How many pivots can be made is a function of the number of stars within the field of view of the star tracker and the magnitude of stars to which it is sensitive. For a star tracker with an 8 deg field of view, sensitive down to magnitude 6.0 stars, a distribution of the number of stars that it will see for a large number of random attitudes can be plotted, and is shown in Figure 2.3. The percentage that at least a certain number of stars will be seen can also be plotted, and is shown in Figure 2.4.

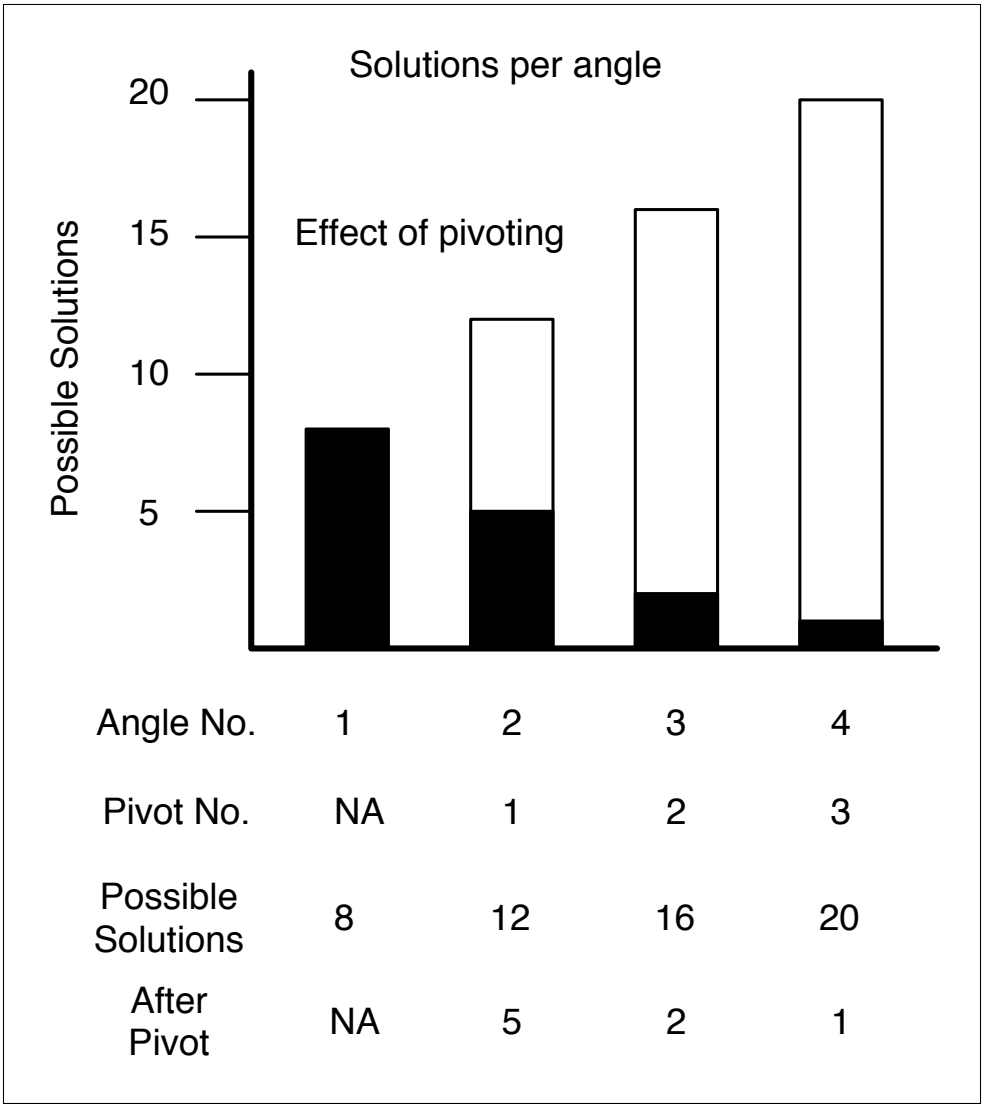

Figure 2.2.: Effect of Pivoting on the Number of Solutions

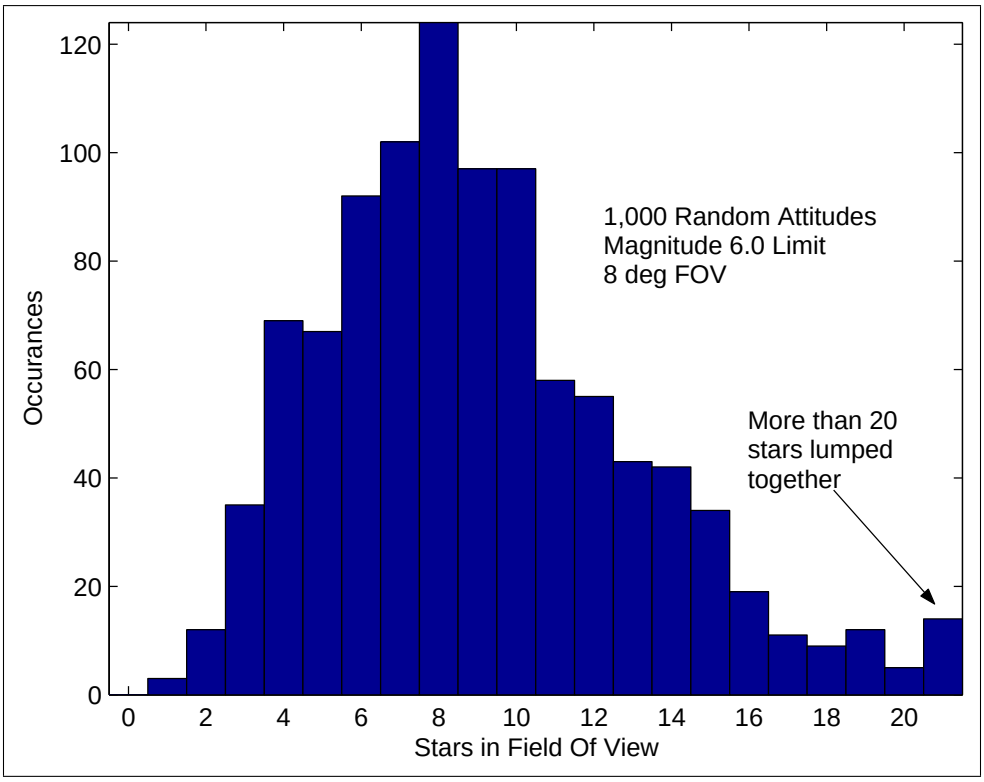

Figure 2.3.: Distribution of the Number of Stars in the Field of View

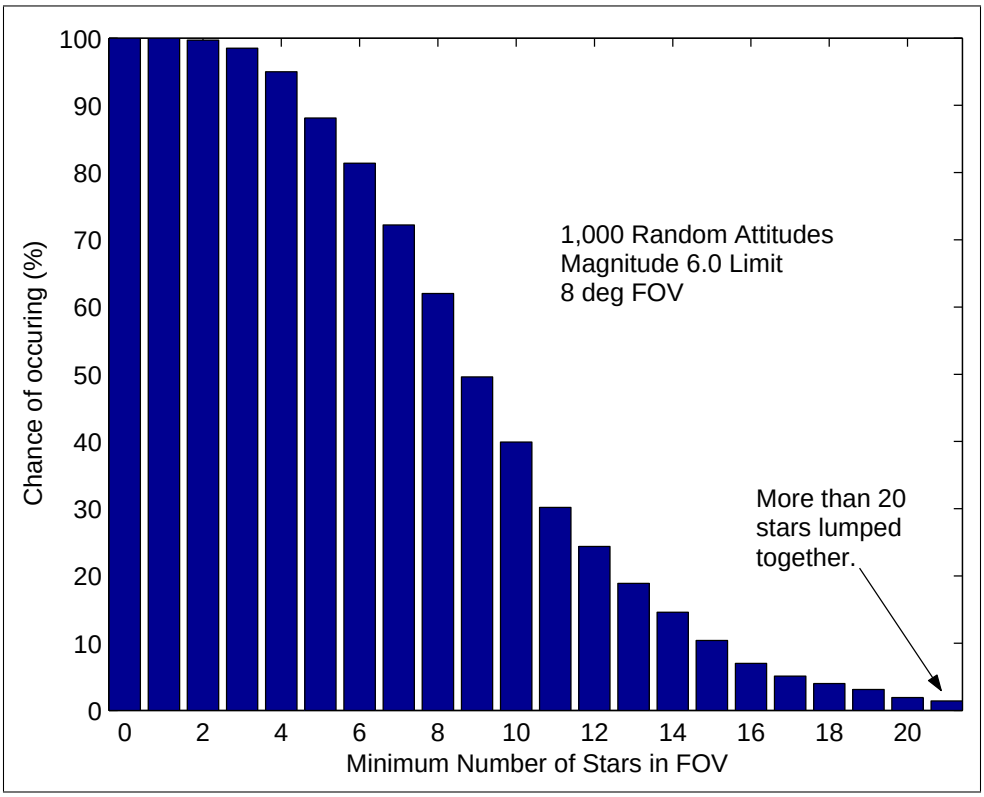

Figure 2.4.: Probability of Seeing at Least a Certain Number of Stars

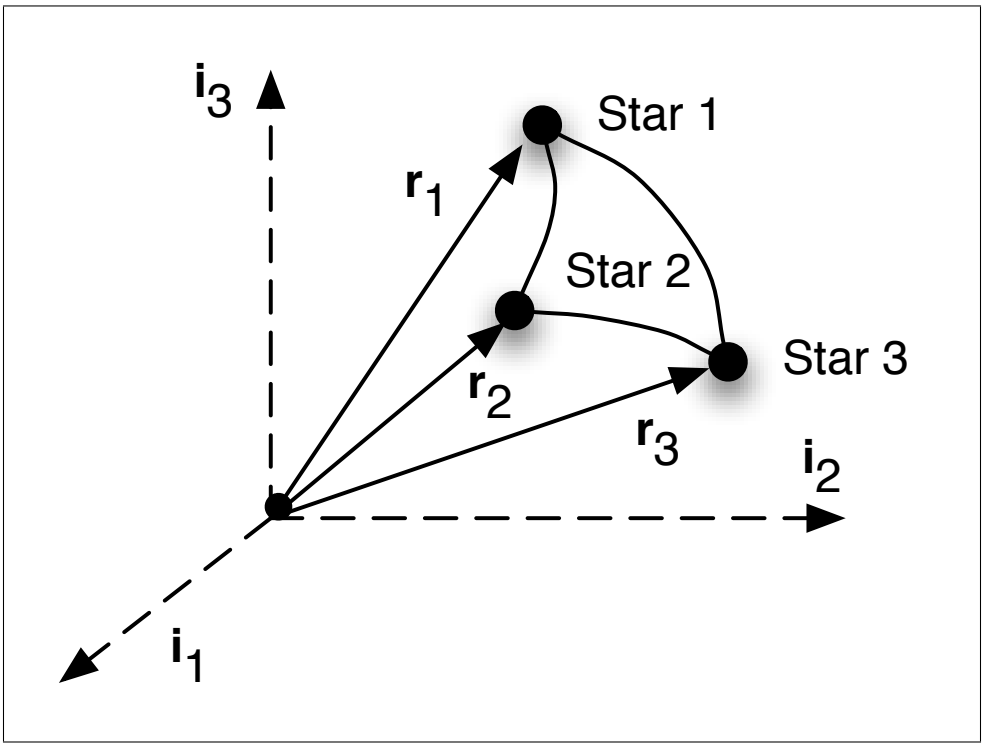

Figure 2.5.: Spherical Triangles Used to Determine Attitude

2.4. Spherical Triangle Method

Instead of measuring the angle between pairs of stars, the spherical triangle method creates spherical triangles of combinations of three stars, as shown in Figure 2.5. The idea is that more information can be obtained from a spherical triangle than an angle, and may enable a star tracker to determine the identity of stars more quickly and using fewer stars overall than the angle method. For instance, in the algorithm presented here, the area and polar moment of spherical triangles are used to determine what spherical triangle is being examined by the star tracker.

One drawback of the spherical triangle method is that it will be impossible to determine attitude with less than three stars in the field of view, whereas the angle method needs only two stars. It was already shown, however, after inlcuding the measurement error present in the star tracker, more than two stars are required to arrive at a solution using the angle method. What will be shown is that the spherical triangle method will require fewer pivots than the angle method and will more likely yield a single solution when there are at least four stars in the field of view.

Since the star pattern recognition algorithm relies on spherical triangle area and polar moment, the catalog of all the spherical triangles must include both properties.

2.4.1. Spherical Triangle Area

Given three unit vectors pointing toward three stars, the area of a spherical triangle can be found using the following equation: [13]

$$A = 4\tan^{-1}\sqrt{\tan\frac{s}{2}\tan\frac{s-a}{2}\tan\frac{s-b}{2}\tan\frac{s-c}{2}} \tag{2.16}$$

where

$$s = \frac{1}{2}(a+b+c) \tag{2.17}$$

$$a = \cos^{-1}\left(\frac{\mathbf{b}_1 \cdot \mathbf{b}_2}{|\mathbf{b}_1||\mathbf{b}_2|}\right) \tag{2.18}$$

$$b = \cos^{-1}\left(\frac{\mathbf{b}_2 \cdot \mathbf{b}_3}{|\mathbf{b}_2||\mathbf{b}_3|}\right) \tag{2.19}$$

$$c = \cos^{-1}\left(\frac{\mathbf{b}_3 \cdot \mathbf{b}_1}{|\mathbf{b}_3||\mathbf{b}_1|}\right) \tag{2.20}$$

The equation is given for the star tracker frame, but can also be used in the ECI frame. To get a range for the measurement, the standard deviation of the calculated area must be known.

2.4.2. Standard Deviation of Area

The determination of the standard deviation of error for a spherical triangle made up of vectors containing position error can be done with an analysis of the error[14]. Starting with a basic measurement equation:

$$\tilde{\mathbf{y}} = f(\mathbf{v}) \tag{2.21}$$

where $\tilde{y}$ is the measurement, and v is the noise. Expanding a Taylor Series about the mean of $\mathbf{v}$, which is $\mathbf{0}$,

$$\tilde{\mathbf{y}} = \mathbf{y} - \frac{\partial \mathbf{y}}{\partial \mathbf{v}}\bigg|\mathbf{v} \tag{2.22}$$

where $\mathbf{y}$ is the truth.

Define $\frac{\partial \mathbf{y}}{\partial \mathbf{v}} \equiv H$, which is called the basis function matrix, then

$$\tilde{\mathbf{y}} - \mathbf{y} = H\mathbf{v} \tag{2.23}$$

Compute the covariance

$$\begin{aligned} E\{(\tilde{\mathbf{y}}-\mathbf{y})(\tilde{\mathbf{y}}-\mathbf{y})^T\} &= H\,E\{\mathbf{v}\mathbf{v}^T\}H^T \\ &= H\,I^2\,H^T \end{aligned} \tag{2.24}$$

Applying this to the spherical triangle area equation, which is a function of a, b and c, the angles between the vectors toward the stars, the basis function matrix is

$$H = \frac{4}{1+z^2}\begin{bmatrix} \dfrac{dz}{da} & \dfrac{dz}{db} & \dfrac{dz}{dc} \end{bmatrix} \tag{2.25}$$

where

$$z = \sqrt{\tan\left(\frac{s}{2}\right)\tan\left(\frac{s-a}{2}\right)\tan\left(\frac{s-b}{2}\right)\tan\left(\frac{s-c}{2}\right)} \tag{2.26}$$

and

$$\frac{\partial z}{\partial a} = -\frac{u_1 - u_2 + u_3 + u_4}{8z} \tag{2.27}$$

$$\frac{\partial z}{\partial b} = -\frac{u_1 + u_2 - u_3 + u_4}{8z} \tag{2.28}$$

$$\frac{\partial z}{\partial c} = -\frac{u_1 + u_2 + u_3 - u_4}{8z} \tag{2.29}$$

where

$$u_1 = \left[\cos^2\left(\frac{a+b+c}{4}\right)\right]^{-1}\tan\left(\frac{b+c-a}{4}\right)\tan\left(\frac{a+c-b}{4}\right)\tan\left(\frac{a+b-c}{4}\right) \tag{2.30}$$

$$u_2 = \tan\left(\frac{a+b+c}{4}\right)\left[\cos^2\left(\frac{b+c-a}{4}\right)\right]^{-1}\tan\left(\frac{a+c-b}{4}\right)\tan\left(\frac{a+b-c}{4}\right) \tag{2.31}$$

$$u_3 = \tan\left(\frac{a+b+c}{4}\right)\tan\left(\frac{b+c-a}{4}\right)\left[\cos^2\left(\frac{a+c-b}{4}\right)\right]^{-1}\tan\left(\frac{a+b-c}{4}\right) \tag{2.32}$$

$$u_4 = \tan\left(\frac{a+b+c}{4}\right)\tan\left(\frac{b+c-a}{4}\right)\tan\left(\frac{a+c-b}{4}\right)\left[\cos^2\left(\frac{a+b-c}{4}\right)\right]^{-1} \tag{2.33}$$

Solving for the covariance,

$$\sigma_p^2 = \sigma^2 H^T H \tag{2.34}$$

Replacing the true quantities with measurements, second order errors result which are negligible. Since the standard deviation is derived analytically, the bounds over which the true area is likely to exist can be determined precisely, no matter what shape or size the spherical triangle is.

2.4.3. Polar Moment of Spherical Triangle

The polar moment makes a good counterpart to area, since it is possible for two spherical triangles that have the same area to have very different second moments. The reverse is also true; two spherical triangles that have the same polar moments may have very different areas. When it comes time to match spherical triangles seen within the field of view of the star tracker to spherical triangles in the catalog, use of these two measures will very quickly reduces the number of possible solutions.

The polar moment of the triangle about its centroid is calculated by breaking the triangle into smaller triangles. The area of each of the smaller triangles, dA, is multiplied by the square of the arc distance from the centroid of each smaller triangle to the centroid of the overall triangle. The result of each smaller triangle is then summed:

$$I_p = \sum \theta^2 dA \tag{2.35}$$

A recursive algorithm is used to determine the area, breaking each spherical triangle into four smaller triangles until a target depth of recursion is met and is shown using a sample spherical triangle in Figure 2.6. Since the accuracy of the method depends on how many triangles the original triangle is broken into, it was important to determine how many levels

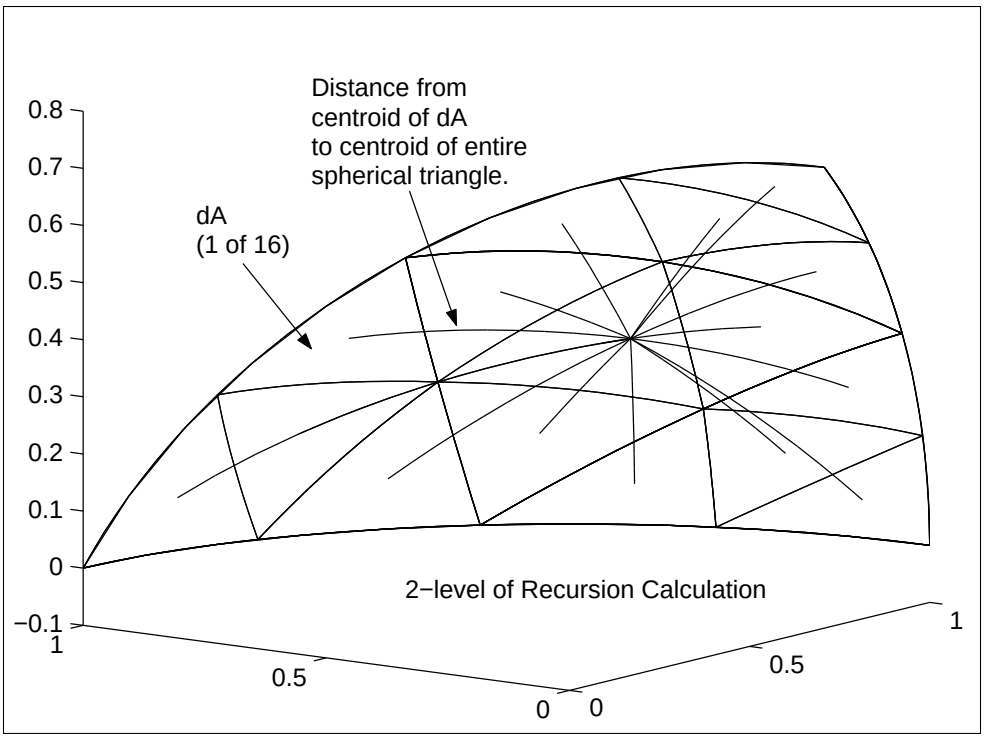

Figure 2.6.: Method for Calculating the Polar Moment of a Spherical Triangle

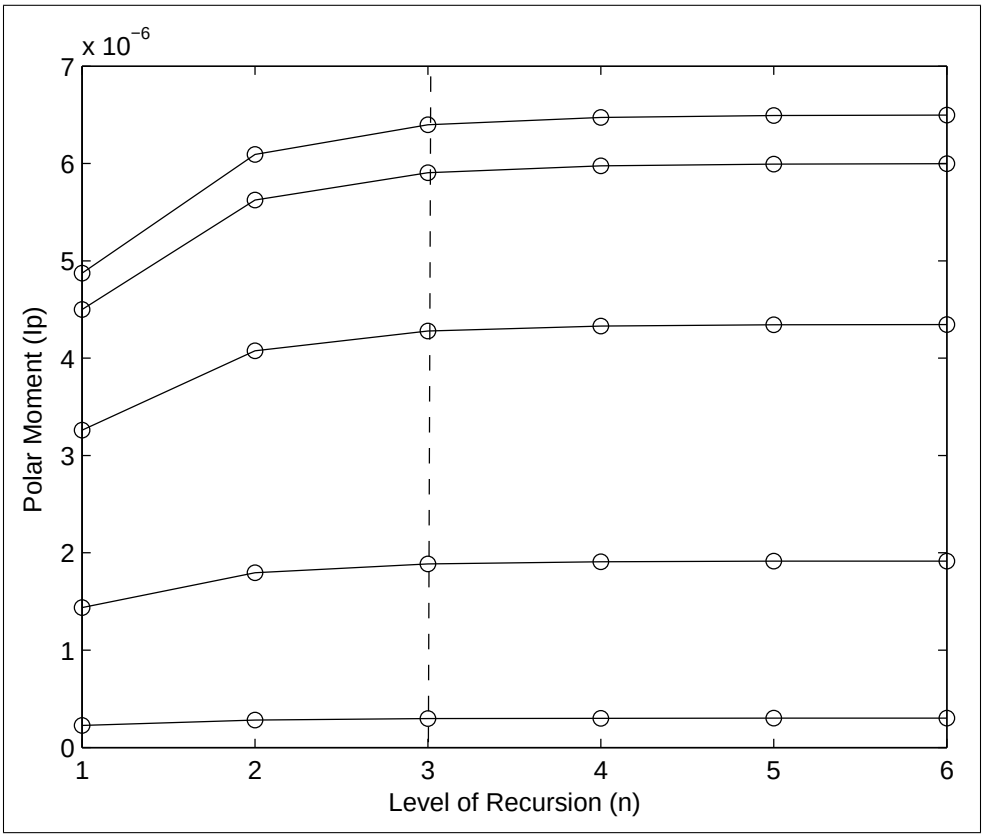

Figure 2.7.: Convergence of Polar Moment Recursive Algorithm

or recursion are required to achieve a good approximation of the actual solution. A piece of Matlab code has been written to randomly select spherical triangles from the catalog and determine the polar moment using different levels of recursion. The results are shown in Figure 2.7. It can be seen that three levels of recursion, where each triangle is is broken into 4^3 or 64 pieces, are relatively close to the values at which each converged and is used throughout the routines.

2.5. Standard Deviation of Polar Moment

The standard deviation of the second moment calculation is calculated by simulation. A thousand random triangles from the catalog were chosen, and one hundred measurements of the second moment for each triangle were made including random measurement error. Figure 2.8 shows how the standard deviation of the measurements varies as a function of the polar moment of the triangle. What can be seen is triangles with similar polar moments may have a different standard deviation when the same amount of random position error is included in the measurements.

To put a bound on the measurement, a line was drawn slightly above the maximum standard deviations on the chart. It is safe to say that the standard deviation of any second moment measurement will lie under this line, and will ensure, for instance a 3-sigma bound, the probability that the true measurement will be within the bounds at least 99.7% of the time.

2.6. Spherical Triangle Pivot

The method for matching spherical triangles is similar to the method for matching angles A spherical triangle is made from three stars in the field of view, and its area and polar

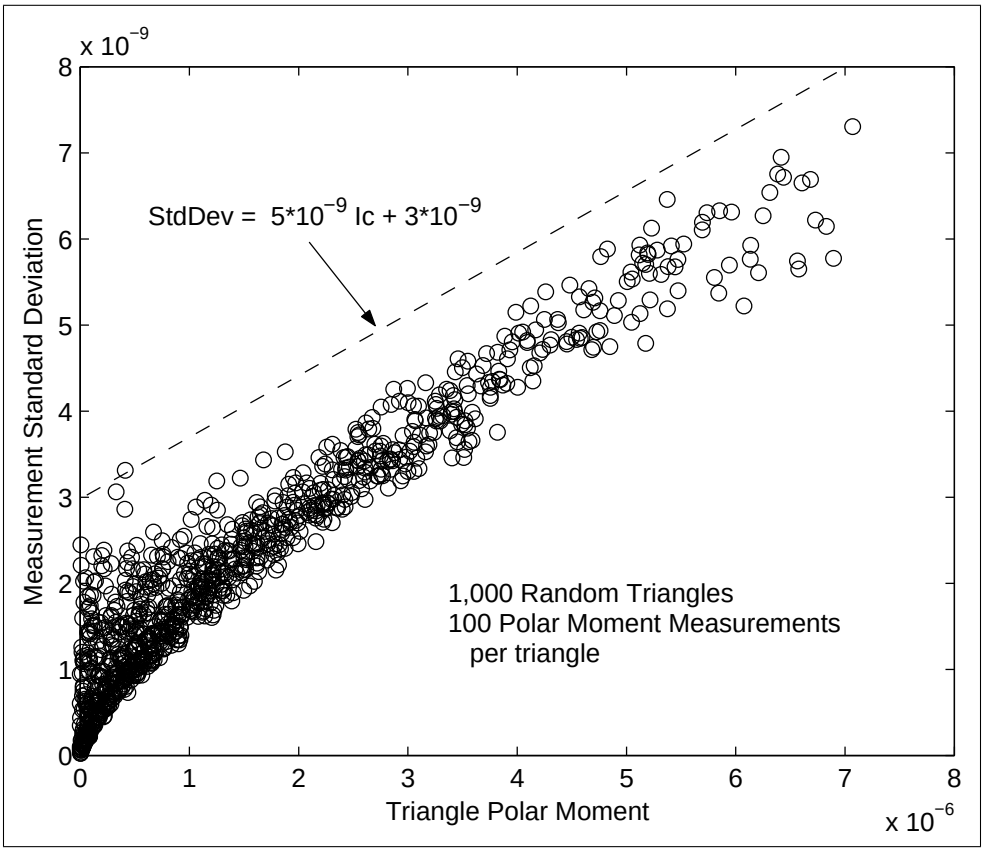

Figure 2.8.: Standard Deviation as a Function of Spherical Triangle Polar Moment

moment are calculated. A range over which the true area and polar moment exist are calculated using the standard deviations for each. Going through the catalog of triangles, triangles that have an area and polar moment that fit within the bounds calculated for the triangle in the field of view are tabulated. Hopefully, only one possible solution exists, but that does not happen.

If more than one solution exists, a pivot similar to the angle method is made. Another spherical triangle is made from the stars in the field of view such that there are two stars in common with the first triangle. A list of possible solutions is made, then the list of solutions between the first spherical triangle and second spherical triangle are compared. Any solution in each list that does not have two stars in common with a solution in the other lists is discarded. After the comparison is made, if more than one solution exists, another pivot is made. Pivoting continues until either a singular solution is found, or there are no more spherical triangles with which to pivot. Pivoting such that only one star is shared between the first and second spherical triangles could be done, but would be less effective. The number of spherical triangles that are likely to share one star is greater than two, so the solution would require a greater number of pivots.

2.7. Conclusion

The biggest difference between angle method and the spherical triangle method is that the angle method uses only one property, the angle between stars, to approach a solution. The spherical triangle method uses two properties, area and polar moment. By eliminating spherical triangles that do not have the appropriate area and polar moment, the number of possible solutions to spherical triangles in the field of view can be very quickly reduced.

Calculating the area and polar moment of the spherical triangles in the field of view is computationally intensive. However, if only a few pivots are required to approach a solution,

the total CPU time required by the method will be relatively small. The angle method, however has the opposite situation; while its mathematics are simpler and will be able to test angles at a greater rate of speed, it will require far more pivots than the spherical triangle method to reach a solution, consuming more CPU time overall than the spherical triangle method.

3. Preparing the Angle and Spherical Triangle Catalog

3.1. Introduction

For the angle method a catalog of all the angles between stars that can fit into the field of view of the star tracker must be created; for the spherical triangle method, a catalog of all the spherical triangles that can fit into the field of view of the star tracker must be created. Done inefficiently, cataloging the angles and especially the spherical triangles can take quite a long time on a typical personal computer. In this chapter, an efficient method for creating the catalogs of angles and spherical triangles is presented.

3.2. Star Data

The star data used is from Skymap.com, and includes 12,443 stars down to magnitude 12 in brightness[15]. The unit vector to each star in space is provided in cartesian coordinates in the ECI coordinate frame.. For this thesis, the star tracker is assumed sensitive to magnitude 6.0, which results in 8,118 stars being used.

3.3. Approach to Catalog Generation

Creating a catalog of angles and triangles suitable for the star pattern recognition algorithms is deceptively simple. All that needs to be done is to create a list of all the angles and spherical triangles that can be made using stars, magnitude 6.0 or brighter, which will fit into an 8 degree field of view. The simplest method would be to test every combination of 3 stars to see if it will fit into the field of view. The equation for calculating this is shown below:

$$ {}_nC_r = \frac{n!}{(n! - r!)r!} \tag{3.1} $$

where $n = 8118$ stars and $r = 3$ level of combination

Limiting the triangle search to stars magnitude 6.0 or brighter results in almost 89 billion combinations to test! On the hardware and software used, it would have taken weeks to test them all. There are ways to greatly reduce this number however. If the angle between any two stars is greater than the field of view of the star tracker, there is no point in testing any combinations of three stars which include the two. Even so, this method is still computationally intensive. Angle searches, requiring a test of every combination of two stars, will require "only" 33 million tests, which is manageable using brute force under the hardware being used, but would still benefit from a more efficient method.

If there were a way scan the stars for angle and triangles such that stars that were very far away from each other would not be tested, the number of combinations that would need to be tested might drop tremendously, and the time for computation would become more practical. Such a structure is called a quad-tree.

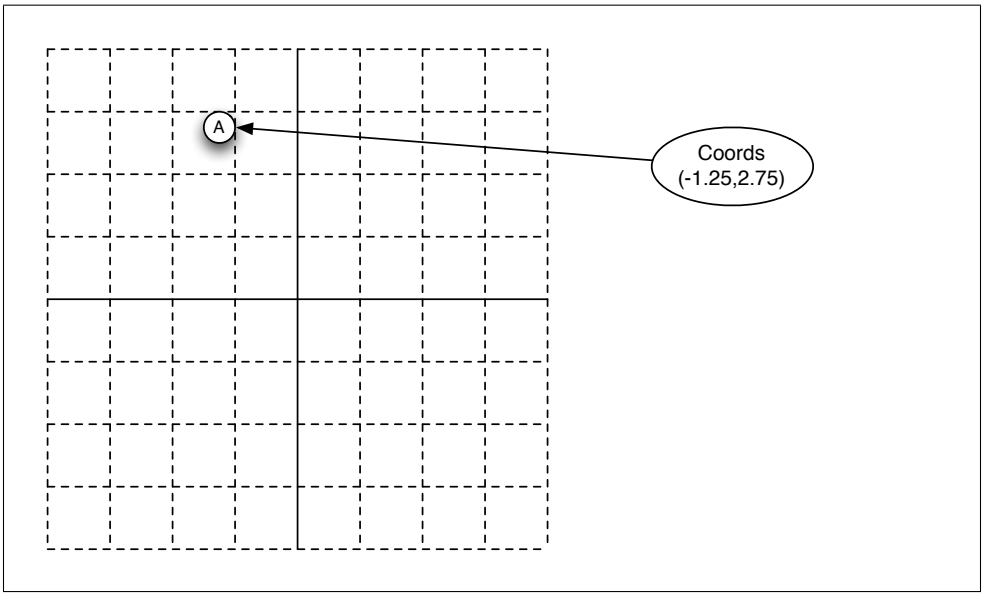

Figure 3.1.: Object A Located in an 8 × 8 Grid

3.3.1. Quad Tree Structure

A quad-tree is a data structure by which objects in 2-D space are stored so that their position in space can be inferred by their location in the data structure. It will be modified so that, given an object in space, objects near it can be easily located. It will allow efficient cataloging of angles and triangles for the star pattern recognition algorithms employed later.

To demonstrate, a quad-tree will be constructed to store several objects located within an 8 × 8 grid, beginning with a single object, Object A, as shown in Figure 3.1. The quad-tree structure is started with a root quadrant, which encompasses the entire space, as shown in Figure 3.2. All searches for an object in space using a quad-tree will start with the root quadrant.

In addition to the root quadrant, another data structure is created and is indicated by diamonds located in the corners. These are vertex structures, and at this point nothing is stored within them. Each quadrant structure, however, stores the vertices that exist

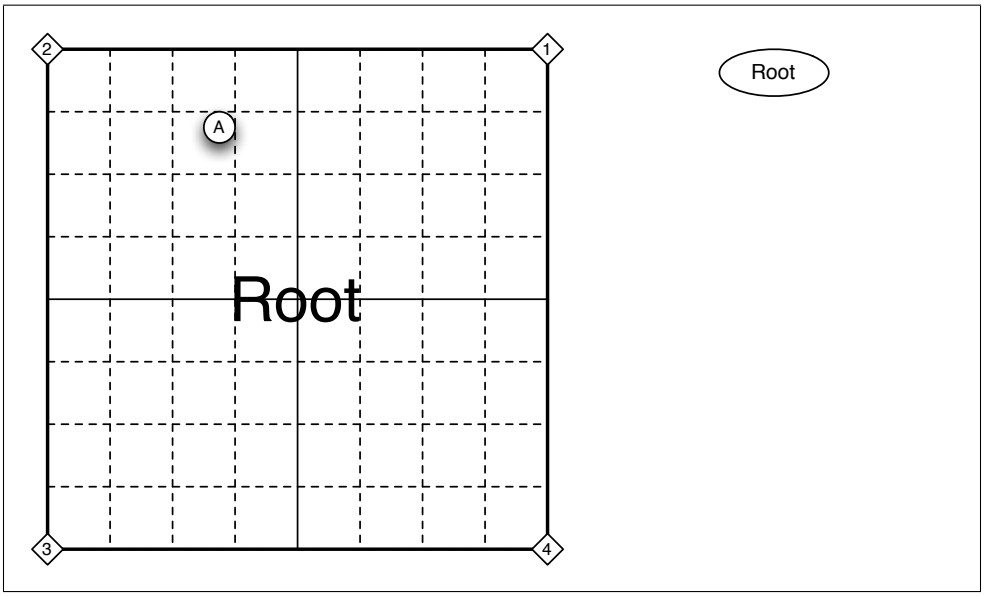

Figure 3.2.: Root Level of Quad-Tree Created

at its corners. So, stored within the root quadrant structure are vertices 1, 2, 3, and 4. The vertices are not typical of a quad-tree, but will be necessary when it comes time for searching for objects near objects, and will be explained in more detail later.

So far, object A is only known to exist within the 8×8 space, which isn't very useful. To more precisely locate object A in space using the quad-tree structure, the root quadrant is divided into four quadrants, numbered 1 through 4 as shown in Figure 3.3. These are called "first-level" quadrants and it can be seen that Object A, as a result, is located within quadrant 2. The schematic for the quad-tree structure can be seen at the right side of the figure. In addition to the additional quadrants, five new vertices are created for the corners of the new quadrants created within the space. As before the, vertices that make up corners of each quadrant are stored within that quadrant.

To further refine the location of object A, quadrant 2 is divided into four smaller, second-level quadrants as seen in Figure 3.4. Since the quadrants are being numbered in the order

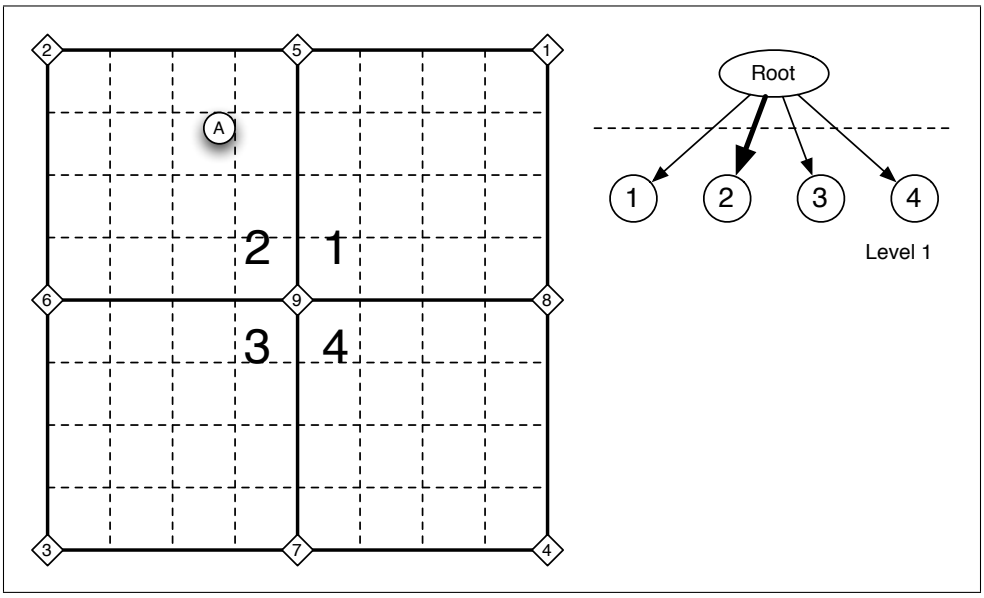

Figure 3.3.: First Level of Quad-Tree Created

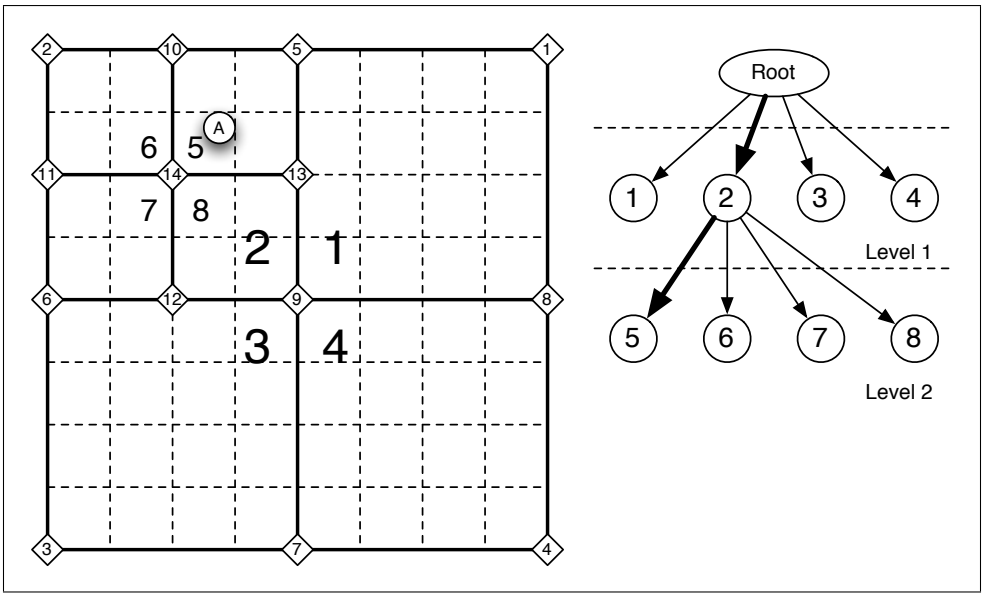

Figure 3.4.: Second Level of Quad-Tree Created

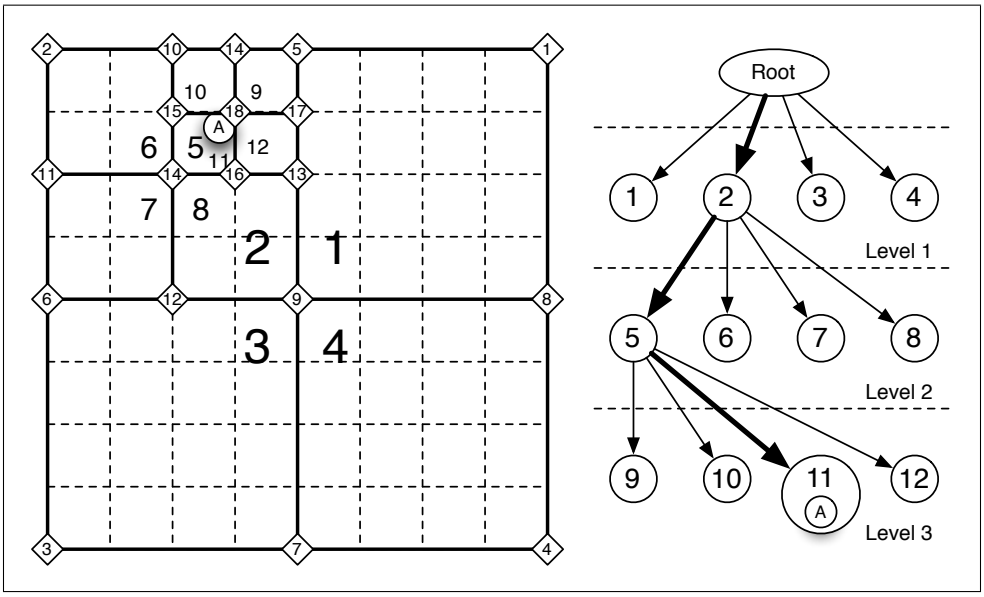

Figure 3.5.: Object A Stored in Third level of Quad-Tree, Quadrant 11

that they are created, these are quadrants 5 through 8. It can be said that quadrants 5 through 8 are the "children" of quadrant 2, and that quadrant 2 is the "parent" of quadrants 5 through 8. Nodes 5 through 8 can also be considered "siblings."

New vertices are created as well; nothing is yet stored within them.

The process can be continued to as many levels as necessary to achieve the precision necessary to define the object's position in space. In Figure 3.5, Object A is located in the quad-tree three levels deep and stored in quadrant 11. Object A is now known to within 1 unit of the space.

To demonstrate how a quad-tree can be used to locate objects in space, the quad-tree will be traversed in such a way as to find what object lies within the area bounded by (-1,2) to (-2,3). This area is determined to lie inside of area defined by quadrant 2, and then determined to lie inside the area defined by quadrant 5. It is finally matched to quadrant 11, where, it is determined, Object A resides. The advantage to this method is that the

objects themselves did not have to tested. If there were a thousand objects stored in the space, rather than test every object to determine its location, only a short traversal through the quad-tree gives the result.

For this example, the objects are to be stored in the quad-tree three levels deep. This level is called the target level. When quadrants are created at the target level, the vertices that corner these quadrants store the target-level quadrants they touch. For instance, stored within vertex 15 are quadrant numbers 10 and 11; stored within vertex 18 are quadrant numbers 9, 10, 11 and 12. This information will become useful when it is time to determine what level three quadrants border each other.

The process of storing objects within the quad-tree continues in this manner. In this example, three more objects, B, C, and D, have been added to the space as shown in Figure 3.6. Object B requires no new quadrants be made and is recorded in quadrant 12. Objects C and D require a new second level node, number 13 (not labeled), and four children, numbered 14 through 17. Four more vertices are also required, numbered 19 through 22. Object C is stored in quadrant 14 and object D is stored in quadrant 16.

To catalog angles and triangles of stars, what is needed is the ability to know what objects lie within a certain distance of another object. The quad-tree can be used to find what objects lie within a particular area of space, but cannot, without modification, be used to locate the objects near another. To demonstrate, the quad-tree and vertex structures will be used to locate all the objects within one unit distance of object A, as shown in Figure 3.7. In the figure, a circle is drawn around object A to show the area that must be searched.

The first place to look is quadrant 11, the quadrant in which object A is stored. It is important to realize that since the quadrants are square, 1 unit in length, their diagonal is 1.44 units in length. As a result, not every object in quadrant 11 will necessarily be within

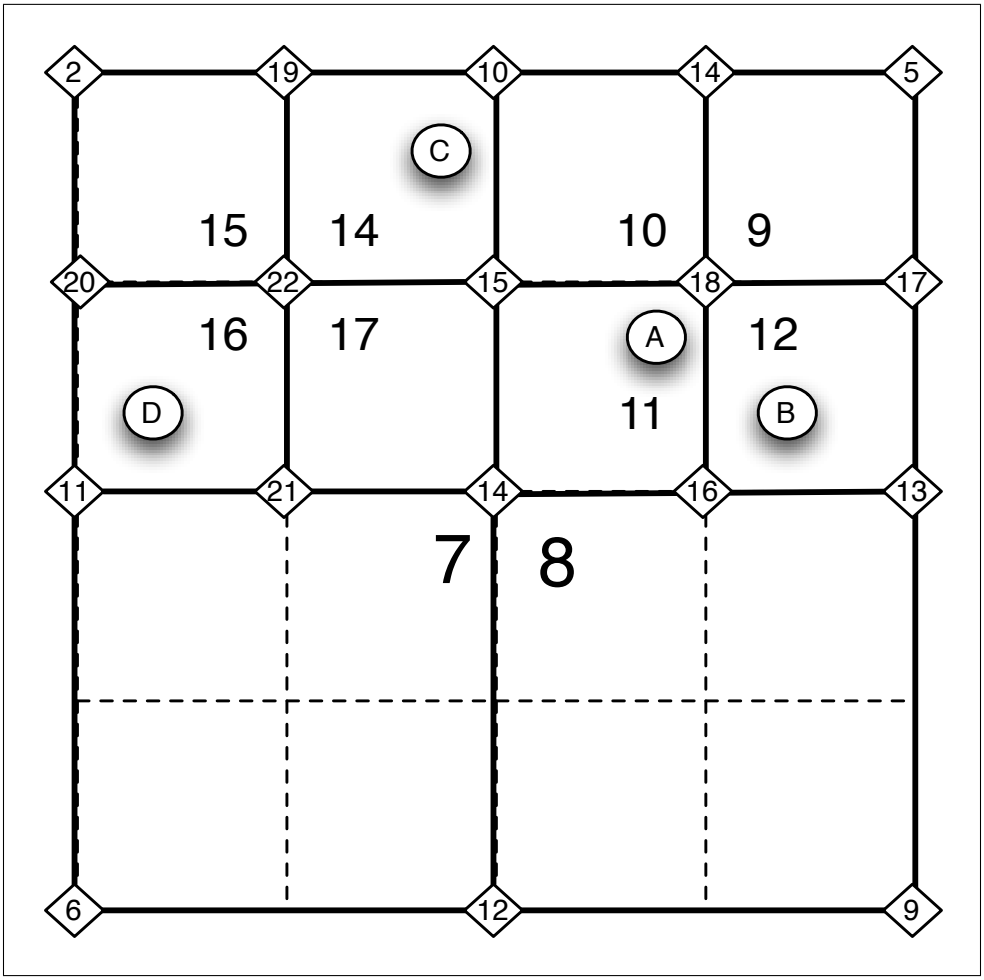

Figure 3.6.: Insertion of Additional Objects into the Space

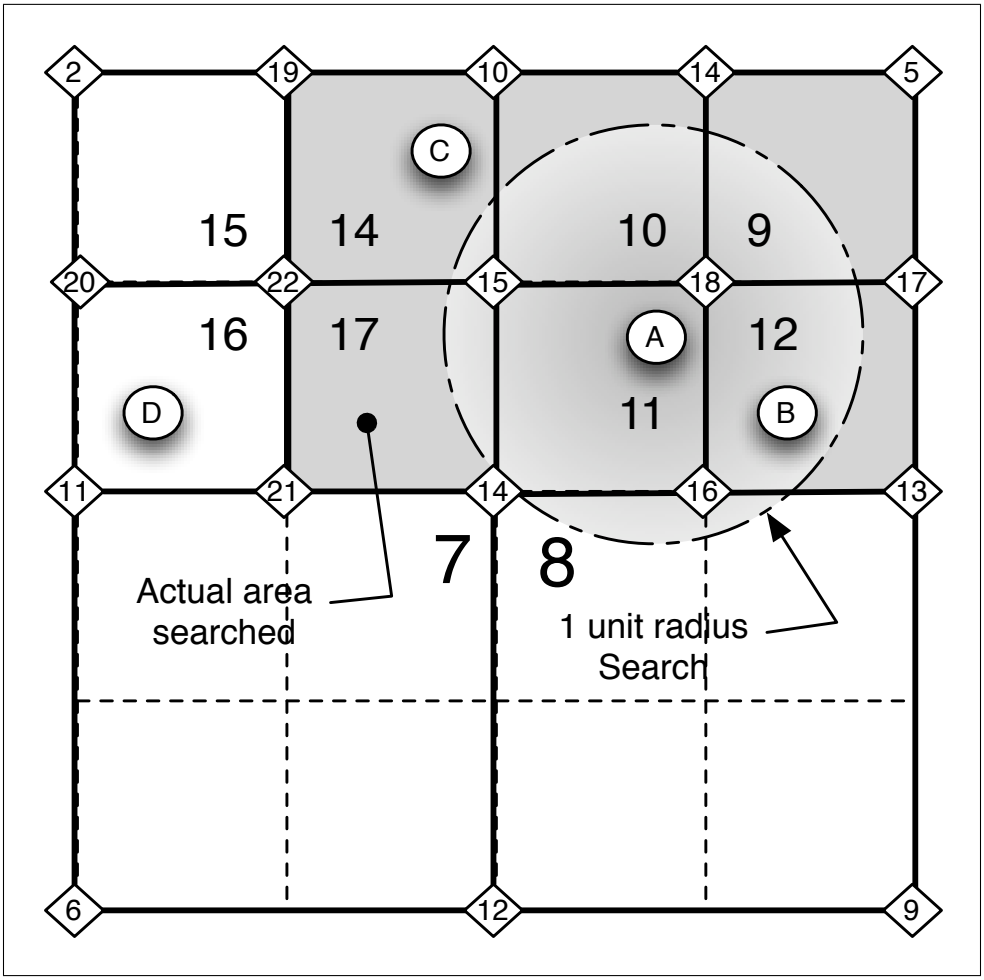

Figure 3.7.: Search Area Around Object A

Table 3.1.: Vertices Cornering Quadrant 11 and Their Contents

Vertex	Quadrants Cornering
18	9, 10, 11, 12
15	10, 14, 17, 11
14	17, 11
16	12, 11

one unit of every other object in quadrant 11. The distances between object A and other objects within quadrant 11 must be checked to be sure they are actually within 1 unit of object A. For this example, however, there are no other objects other than A in quadrant 11.

It can be easily seen that the circle cuts not only through quadrant 11, but also neighboring quadrants 9, 10, 12, 14 and 17. The quad-tree structure cannot be used to determine this, since there is no way to discern from the structure that, for instance, quadrant 17 neighbor's quadrant 11. This is where the vertex data structure becomes useful. To determine what quadrants border quadrant 11, the vertices that are at its corners are queried for all the quadrants that they corner. This data is shown in Table 3.1.

Merely reducing the search distance so that it does not have to cut through other quadrants is not possible, since the target object may exist near the border of the quadrant in which it resides. Even a very small search distance will require searching neighboring quadrants. The objects in nodes 9, 10, 14, 17 and 12 must then be checked to see if they lie within one unit of object A. The actual positions of the objects are used for this measurement. In this case, Object B does lie within 1 unit of object A, but object C does not.

Studying the circle that defines a 1 unit radius about object A, it can be seen that it cuts into higher level quadrant 8. Since it is not a third level quadrant, there can be no

objects there, and nothing will be missed by not searching there. This is why the vertices only record the third-level quadrants they corner; it confines the search to one unit third-level node distance away from the object A, and ignores any upper level quadrants that the search area may cover.

At this point, it becomes clear that what level you divide the space determines how far away from an object other objects can be found. If the quad-tree were only two levels deep, the distance that can be "scanned" will be double that of a three level deep tree. But if the distance to be searched is still only 1 unit radius, an unnecessarily large area of the space will be searched, and will take longer than necessary to finish.

If the quad-tree is made too deep for a particular search distance, a worse situation develops. For instance, if the quad-tree created here were made four levels deep, the size of the fourth-level quadrant will be 1/2 unit. The fourth-level quadrants that surround the quadrant containing Object A will not be large enough to cover the entire area occupied by the 1 unit radius circle about object A. Objects lying outside the covered area that are within 1 unit of Object A will be missed.

If a quad-tree contains a great many objects, many within the search distance of a particular object, an interesting test of the algorithm can be done. Starting with a root-level search, a certain number of items will lie within the search distance. As the quad-tree is made deeper and deeper, the number of items found will remain constant until a critical level is reached. At this point, the node search area will be smaller than the area required, and objects at the outer edges of the required search area will be missed. The number of objects detected will then drop suspiciously, indicating the tree is too deep for the search distance asked.

The most important aspect of this example is that object D was completely ignored. Since it was not in a quadrant neighboring quadrant 11, there was no point testing it; it

could not have been within 1 unit of object A. In this example, a lot of effort was spent to ultimately exclude only one object, but when it comes time to catalog angles and spherical triangles, thousands of stars that would otherwise be tested are eliminated from testing, and the computational effort required greatly reduced.

3.3.2. Spherical Quad-Tree Construction

To catalog the stars using a structure similar to the quad-tree required finding a way to divide the surface of a sphere into regular shapes that could be continuously divided into the same regular shapes. The structure chosen for this is spherical triangle. The surface of the celestial sphere is divided into four equal spherical triangles, making a spherical tetrahedron, as shown in Figure 3.8. A similar approach was used by Bauer for cataloging stars so that star density in the celestial sphere could be controlled. [16]

If that spherical tetrahedron is unfolded and flattened for purposes of clarity, the four triangles that envelope the celestial sphere would look as they do in the left-most part of Figure 3.9. Notice that vertex number 1 appears to be in three places. This only due to the tetrahedron being unfolded; vertex one corners nodes 1, 2 and 3. Positioning an object in the spherical triangle quad-tree is done in a manner similar to using a 2-D grid of quadrants. The root triangle is divided into the four parts shown at the left most part of Figure 3.9. Node 3, which contains object A, is divided into four triangles by connecting the midpoints of the node. Object A is located within level-two node 5, which is then divided one more time, into third-level nodes. Object A is then stored in node 10.

Vertices are also created as needed, storing the numbers of the third-level quadrants they touch. They will be needed so that objects within a certain distance of another object can be found.

Figure 3.9 shows was would happen if the triangles were two-dimensional. Figure 3.10

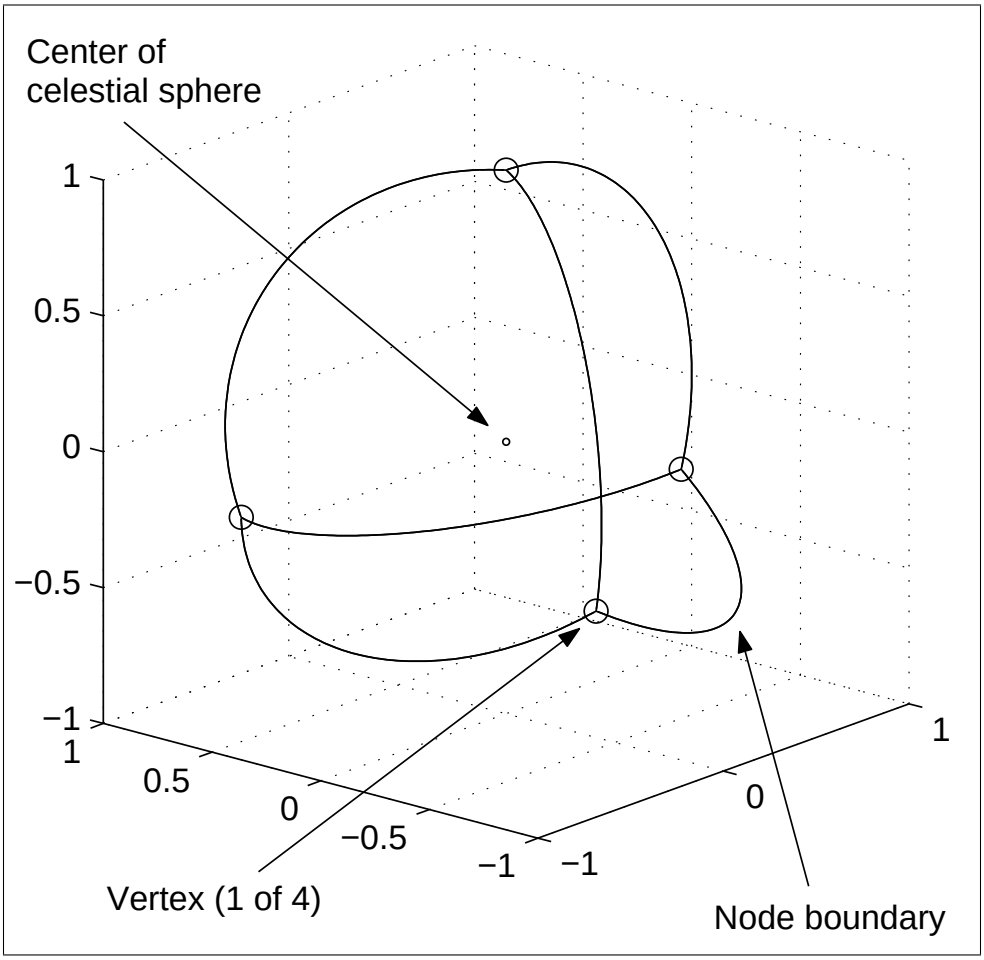

Figure 3.8.: First Level of Spherical Quad-Tree

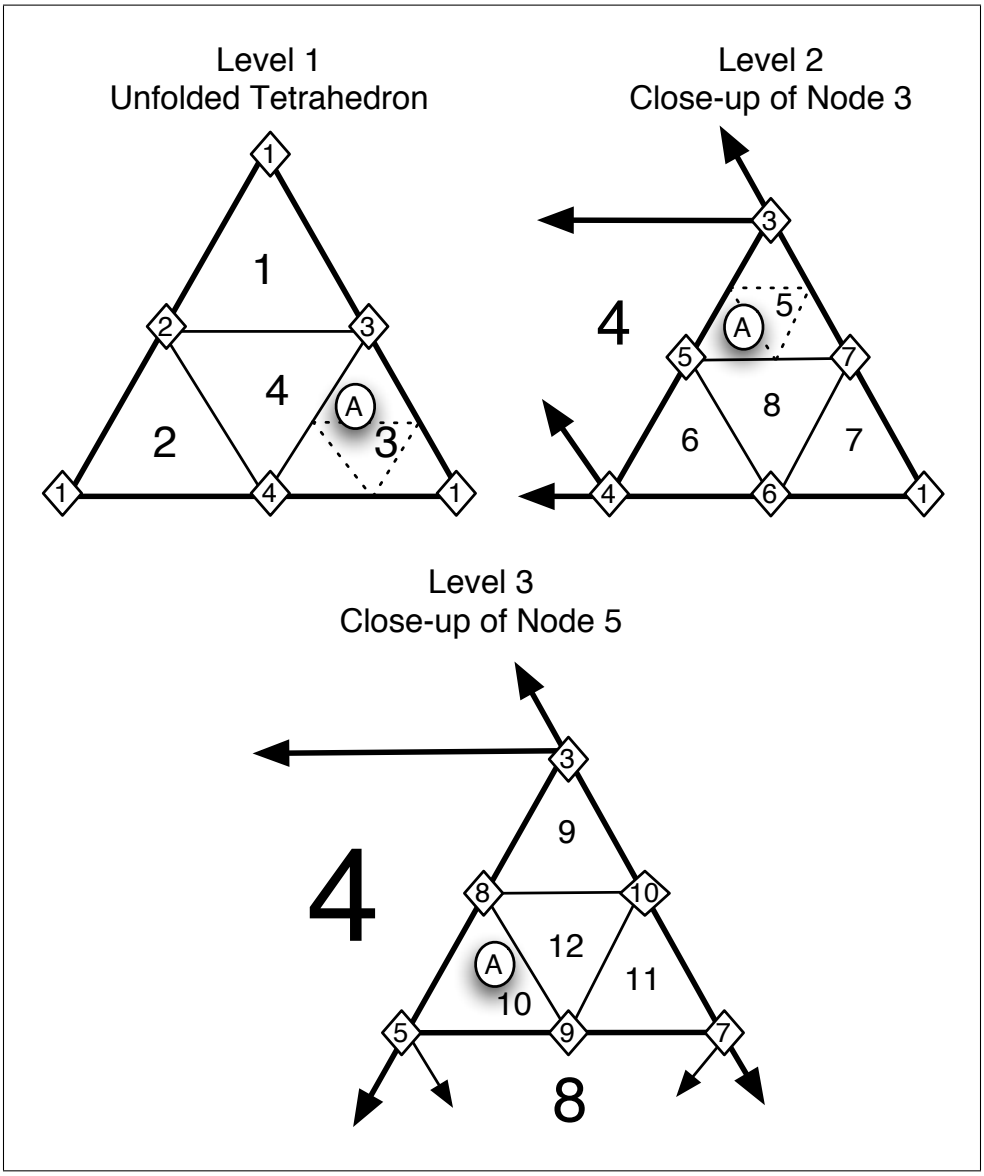

Figure 3.9.: Object A's Insertion into a Triangular Quad-Tree

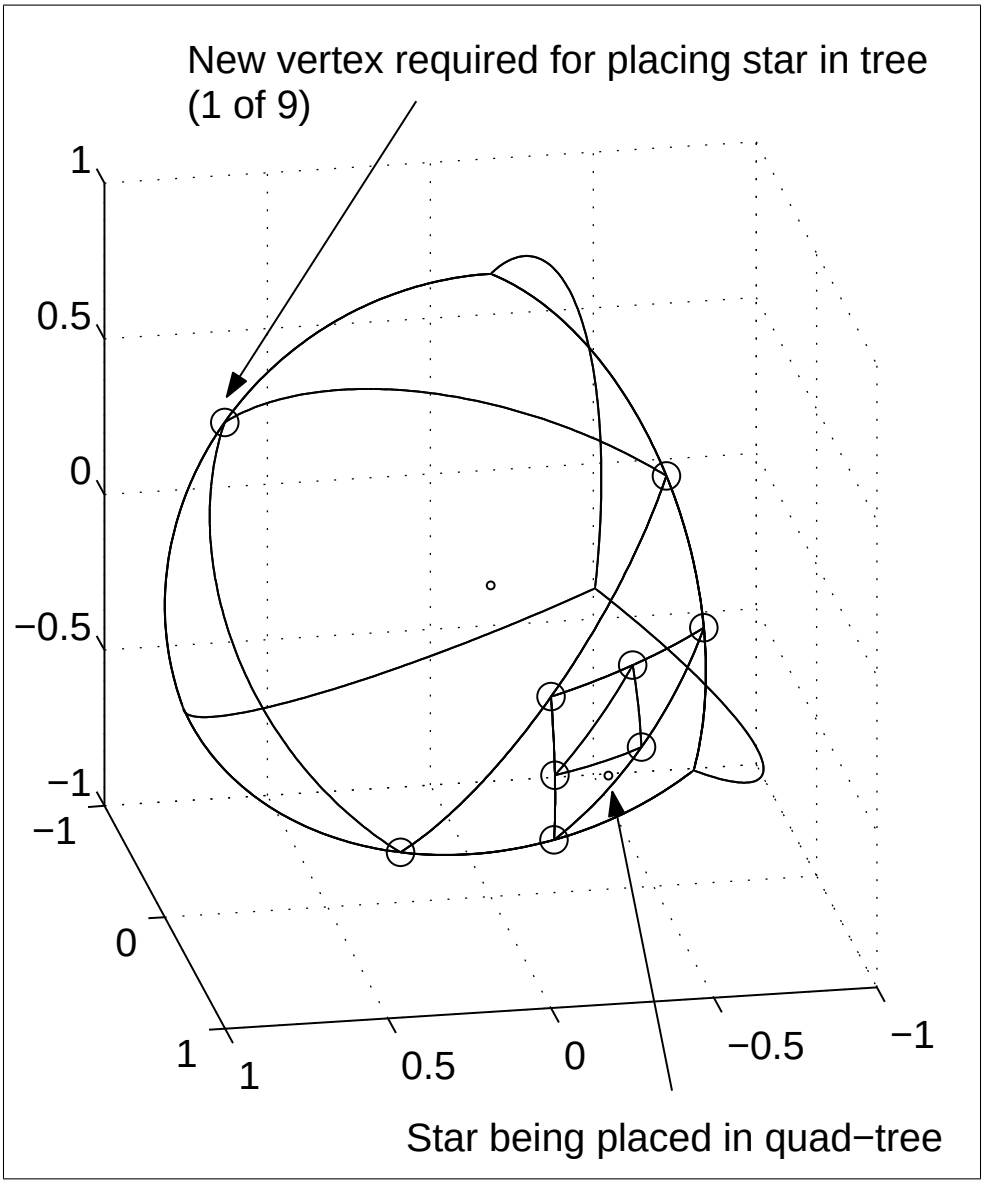

Figure 3.10.: Star inserted into Spherical Quad-Tree

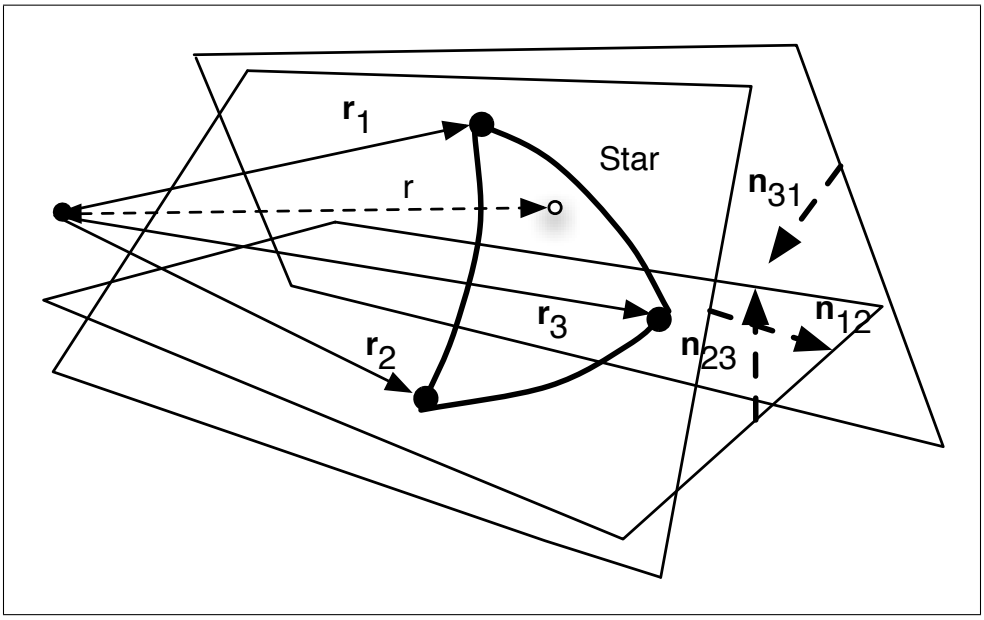

Figure 3.11.: Method for Determining Star Presence Within Node

shows a similar operation in the actual spherical triangle quad-tree. The first thing that can be seen by examining the triangles in the quad-tree is that they are not all the same size and shape. This is due to the choice of selecting the first level of the quad-tree to be divided into only four spherical triangles. The arcs that join the midpoints of these spherical triangles are joined to form children lie on a plane than passes through the center of the celestial sphere. An unfortunate result is that the center triangle is larger than its three siblings and different in shape.

3.3.3. Method for Determining Node

How it is determined which node a star lies can be seen in Figure 3.11. Three planes are constructed from pairs of the three vectors which define the node such that the normals all face toward the interior of the node. If the position of the star lies on the positive side of all three planes, that star is located within the boundaries of the node.

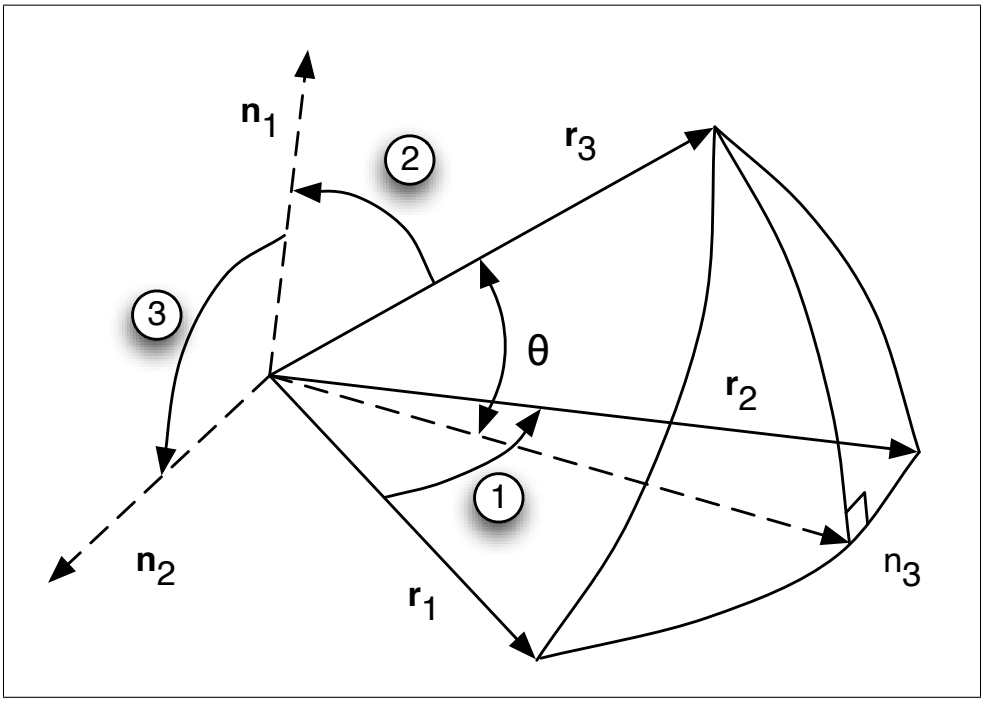

Figure 3.12.: Method for Determining Maximum Search Distance

3.3.4. Spherical Quad-Tree Search Distance

From the 2-D quadrant example, it was found that the level to which the quad-tree is constructed determines how far from an object neighbors can certainly be found. The size and shape of the quadrants at each level are the same shape and size, and as a result the maximum distance that can be searched does not change.

Since the spherical-triangle based quad-tree nodes are not necessarily the same size and shape, the smallest height of each node is calculated when the node is created, as shown in Figure 3.12. The height is found by first finding the longest side of the spherical triangle, in this case $\mathbf{r}_1$ and $\mathbf{r}_2$. The following cross-products need to be determined:

$$\bar{\mathbf{n}}_1 = \mathbf{r}_1 \times \mathbf{r}_2 \tag{3.2}$$

Table 3.2.: Search Distance as a Function of Spherical Quad-Tree Depth

Depth (level)	Search Area (deg)
3	14.5
4	6.94
5	3.43

$$\bar{\mathbf{n}}_2 = \mathbf{r}_3 \times \bar{\mathbf{n}}_1 \tag{3.3}$$

$$\bar{\mathbf{n}}_3 = \bar{\mathbf{n}}_1 \times \bar{\mathbf{n}}_2 \tag{3.4}$$

The height of the triangle, in degrees is then calculated:

$$\theta = \cos^{-1}(\bar{\mathbf{n}}_3 \cdot \mathbf{r}_3) \tag{3.5}$$

If the search radius from an object is limited to the smallest height of all the triangles in the target level, it is certain that no objects inside of that search distance will be overlooked. This distance is used to determine how deep to construct the quad-tree. It should be noted that the deeper the quad-tree is constructed, the less difference there is between triangles at the target level. The levels required for cataloging angles and triangles for the star pattern recognition algorithm do show significant differences however.

The effect that quad-tree depth has the maximum allowable search area is shown in Table 3.2. If the entire 8 deg field-of-view of a star tracker is to be utilized, the quad-tree can be only three levels deep, since it has a maximum allowable search distance of 14.51 degrees, greater than the required search distance of 8 degrees. While the computational effort will be greatly reduced as compared to testing all possible combinations of three stars,

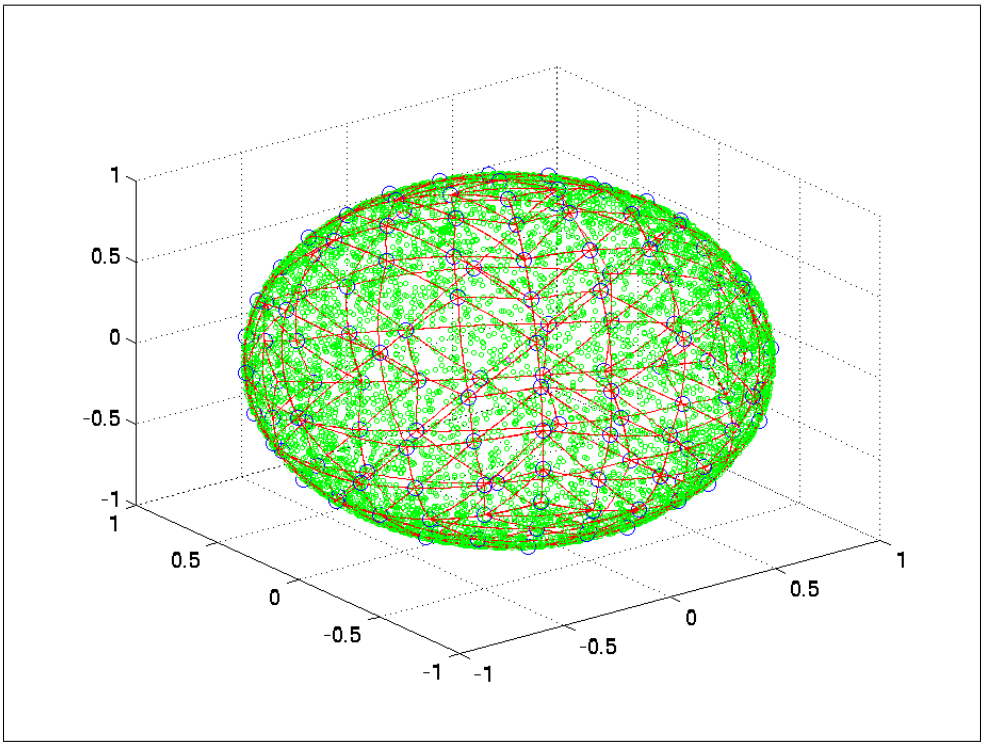

Figure 3.13.: Final Four-level Quad-Tree for Stars Magnitude 6.0 and Brighter

there will be a lot of computation effort wasted, since the search area will be much larger than required.

An interesting alternative is to limit the search distance less than 6.94 degrees, so that the quad-tree can be extended to a forth level. Although the star tracker's entire 8 degree field of view would not be utilized to its greatest advantage, the catalog loss will be minimal, and the computation time required to catalog the angles and triangles will be greatly reduced. Extending the quad-tree to a fifth level cuts the search distance to 3.43 degrees, far too small a search area for a star tracker with an 8 degree field-of-view.

The results of creating a fourth-level quad-tree for all stars in the catalog magnitude 6.0 or brighter is shown in Figure 3.13. It requires 1,364 nodes and 513 vertices.

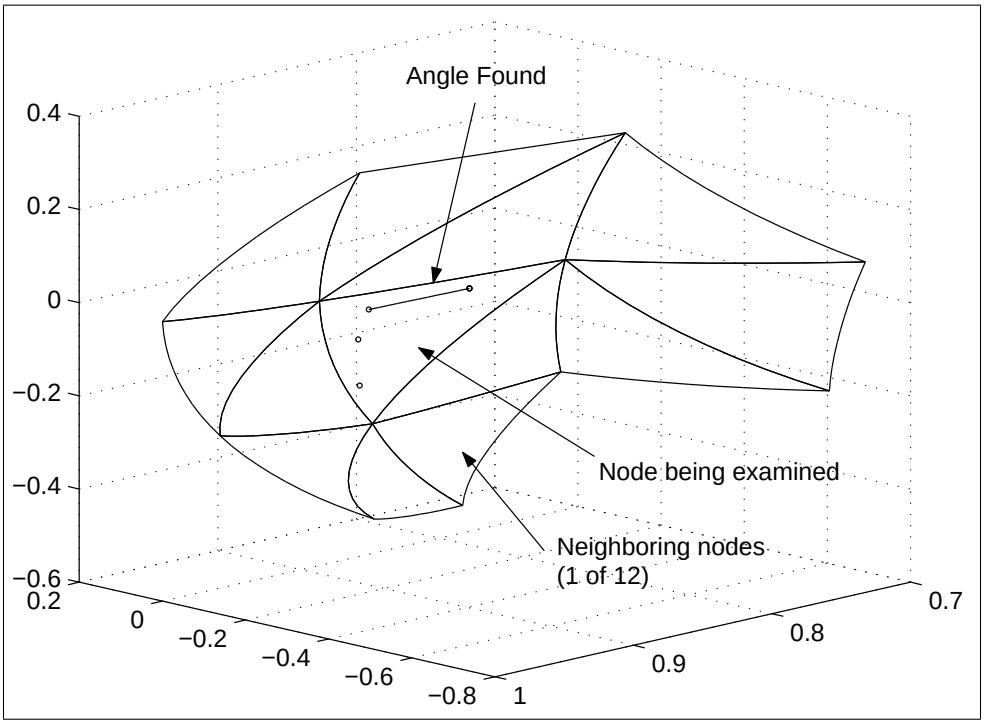

Figure 3.14.: Cataloging Intra-Node Angles

3.4. Cataloging Angles

The quad-tree is searched node by node to see what angles will fit within a 6.94 deg field of view. Because a quad-tree is being used, there are two kinds of angles that have to be searched: single-node angles and two-node angles.

3.4.1. Intra-node Search for Angles

The algorithm will start by examining the first node in the quad-tree that is of the target level, in this case, level 4. Once found, the routine will go through all the possible combinations of two stars that exist within the node, and determine if the angle they make are less than 6.94 degrees as shown in Figure 3.14. Angles that are less than 6.94 degrees are

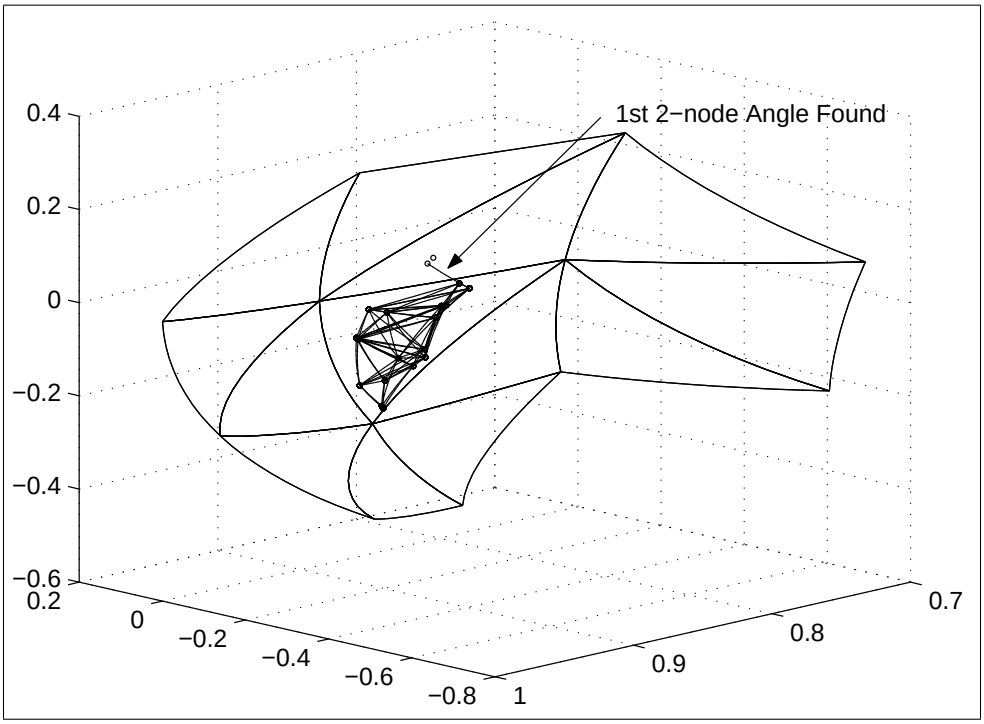

Figure 3.15.: Cataloging Two-node Angles

added to the catalog.

3.4.2. Two-node Search for Angles

After the intra-node search is done, the algorithm goes through all the neighboring nodes, looking for combinations of stars that have an angle less than 6.94 degrees. This can be seen in Figure 3.15. These are called two-node angles.

If any are found, like before, the angle is cataloged along with the stars that make it up. The difference here is, when a neighboring node is later examined for angles, it must somehow be kept from trying to catalog the same two-node angles previously found. After a neighboring node is tested for two-node angles, the neighboring node keeps a record of having been tested by the target node. When the neighboring node attempts to search for

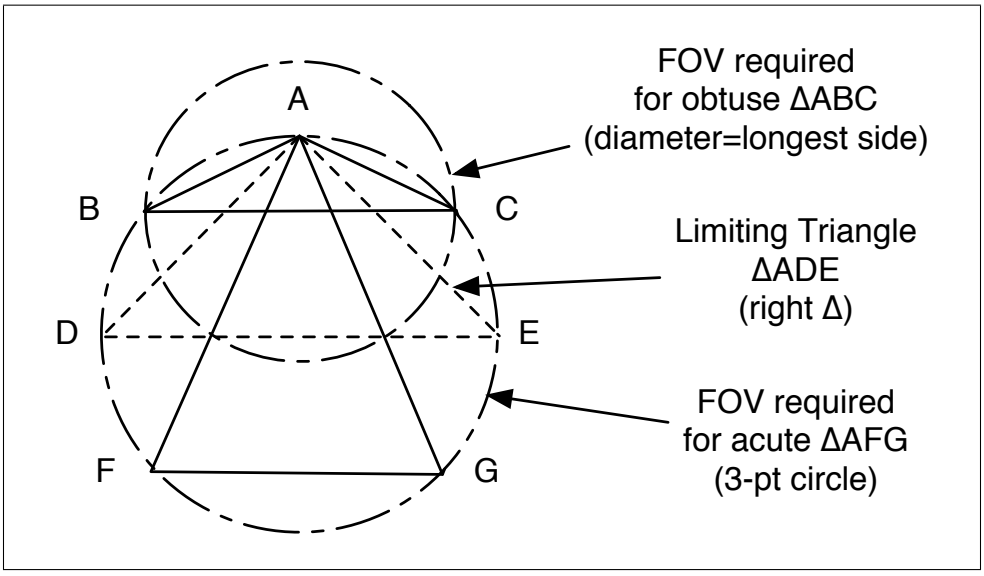

Figure 3.16.: Cataloging Intra-Node Spherical Triangles

two-node triangles, it will see that one of its neighboring nodes has already tested it for two-node angles, and will not search there.

3.5. Cataloging Spherical Triangles

3.5.1. Determining a Spherical Triangle's Field of View

Determining whether a triangle will fit into the field of view of the star tracker is done by looking at the planar triangle that connects the vertices of the spherical triangle. Depending on the whether the triangle is obtuse or acute determines how the field of view it occupies is determied

For instance, in Figure 3.16, if the triangle is acute, such as $\triangle ABC$ the field of view requires is merely the arc distance across its longest side. If the triangle is acute, such as $\triangle AFG$, the equation for determining the diameter of a circumscribed triangle has to be

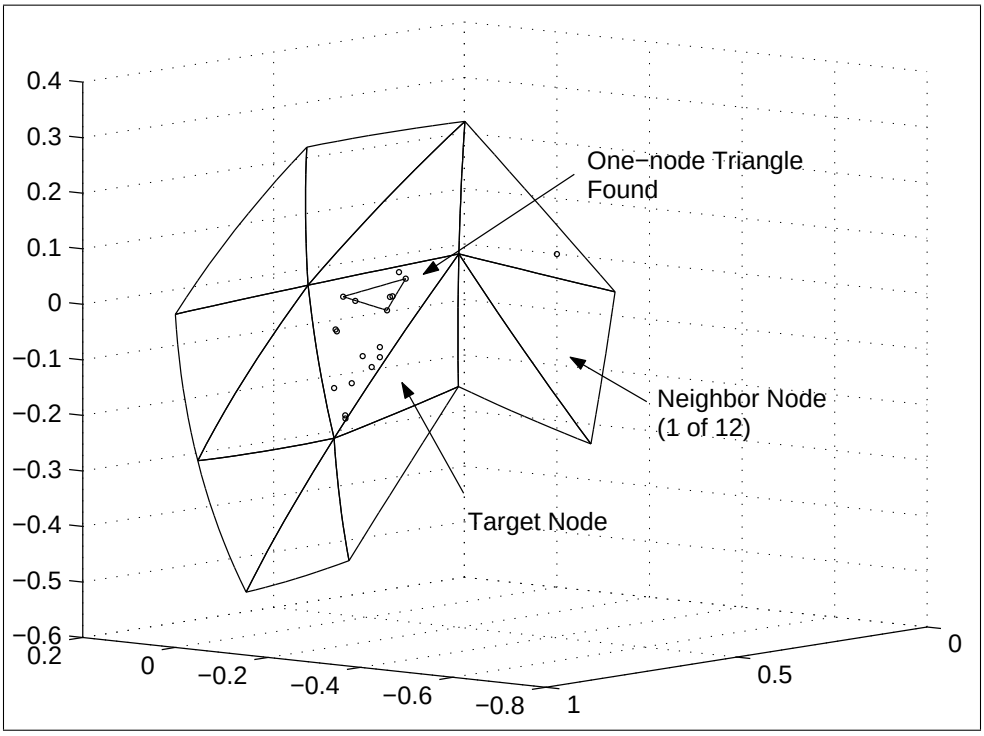

Figure 3.17.: Cataloging Intra-Node Spherical Triangles

used.

3.5.2. Intra-node Search for Triangles

The algorithm will start by examining the first node in the quad-tree that is of the target level, in this case, level 4. Once found, the routine will go through all the possible combinations of three stars that exist within the node, and determine if they any will fit within a 6.94 degree diameter circle, as shown in Figure 3.17. Spherical triangles that do are cataloged along with the stars that make it up.

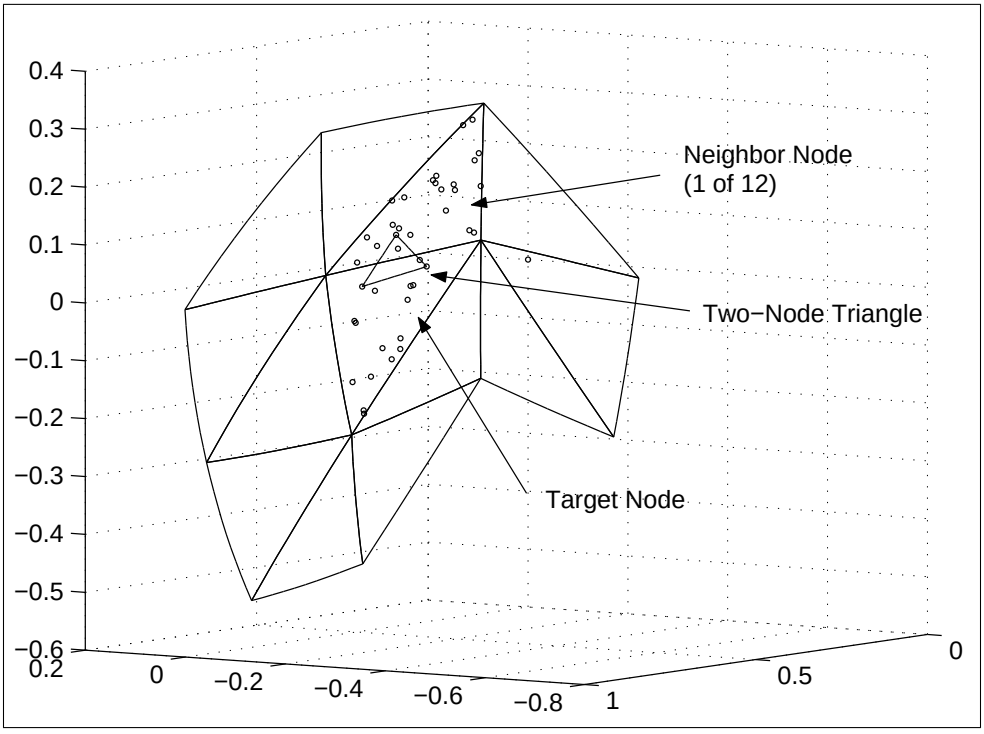

Figure 3.18.: Cataloging Two-node Spherical Triangles

3.5.3. Two-node Search for Triangles

After the intra-node search is done, the algorithm goes through all the neighboring nodes, looking for combinations of spherical triangles that can be fit into a 6.94 degree field of view using only the target and one other node. This can be seen in Figure 3.18. These are called two-node spherical triangles.

If any are found, like before, the triangle is cataloged along with the stars that make it up. The difference here is, when a neighboring node is later examined for triangles, it must somehow be kept from trying to catalog the same two-node spherical triangles previously found. After a neighboring node is tested for two-node spherical triangles, the neighboring node keeps a record of having been tested by the target node, and the target node keeps a record of having tested the neighboring node. When the neighboring node attempts to search for two-node triangles, it will see that one of its neighboring nodes has already tested it for two-node triangles, and will not search there.

3.5.4. Three-node Search for Triangles

Once the two-node spherical triangles are found, the algorithm goes through all the combinations of two nodes that can be made of its neighboring nodes. With each combination of two neighboring nodes, the algorithm tests every combination of three stars that can be made using one star from the target node and one star from each of the two neighboring nodes as shown in Figure 3.19. The spherical triangles that are found to fit within the required 6.94 degree diameter circle are called three-node triangles, since each star that makes it up comes from a different node.

Like the two-node triangles, it is important to prevent neighboring nodes from trying to catalog the same triangles more than once. When each combination of nodes is tested, each node keeps a record of having tested for three-node triangles with the other two nodes.

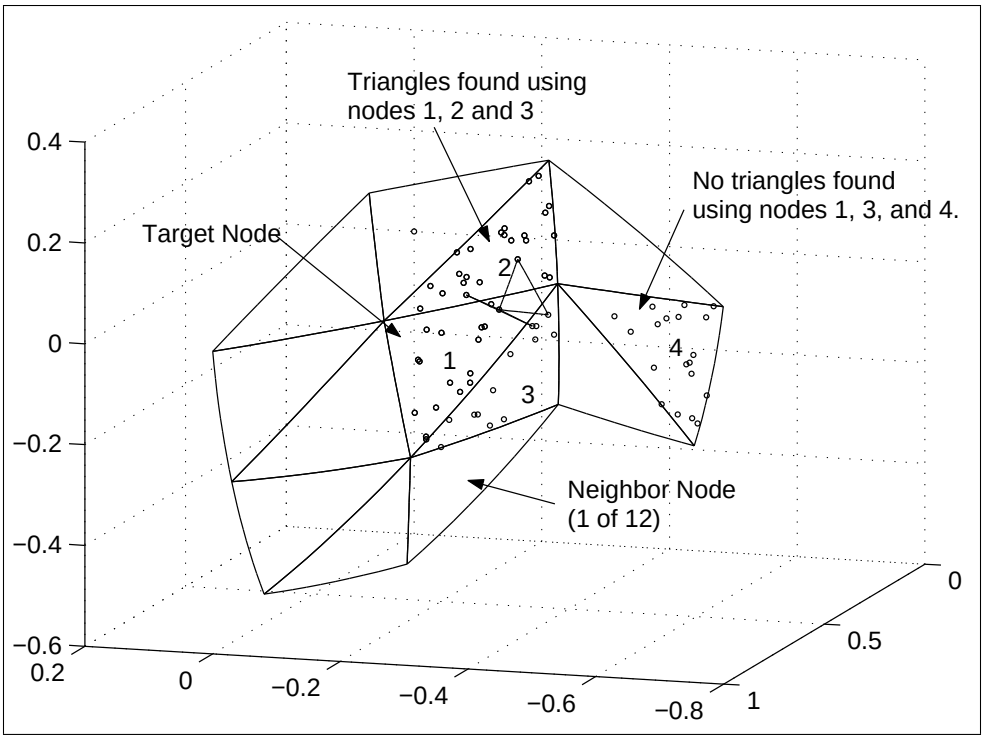

Figure 3.19.: Cataloging Three-node Spherical Triangles

When a neighboring node becomes a target node and attempts a three-node spherical triangle search, it will be aware that it has already attempted three-node triangle searches with some of the neighboring triangles, and won't test those combinations of neighboring nodes.

Once all the single-node, double-node and triple node triangles have been cataloged, the algorithm advances to the next target level node and continues until all the nodes have been tested.

3.6. Catalog Results

The angle catalog algorithm determined, for magnitude 6.0 stars or brighter, limited to a field-of-view of 6.94 degrees, that there are 106,308 angles. It took the hardware used a few hours to catalog them all.

The spherical triangle catalog algorithm determine for the same conditions, that there are 662,799 spherical triangles that can be formed. It took the hardware used about four hrs to catalog them all. This is a long time, but far far shorter than would have been required by merely testing all the combinations of three stars.

3.7. Sorting The Angles And Spherical Triangles

To locate the angles and triangles in the catalog as fast as possible, angles and area of the spherical triangles must be sorted so that a mathematical equation can be fit to it. The individual elements (angles or spherical triangles) of the catalog could be shuffled, but throughout the design of the star pattern recognition algorithm, it was not always clear if the catalog elements would need to be sorted by more than one property. If, for instance, the spherical triangle catalog had to be sorted according to area and polar moment, two

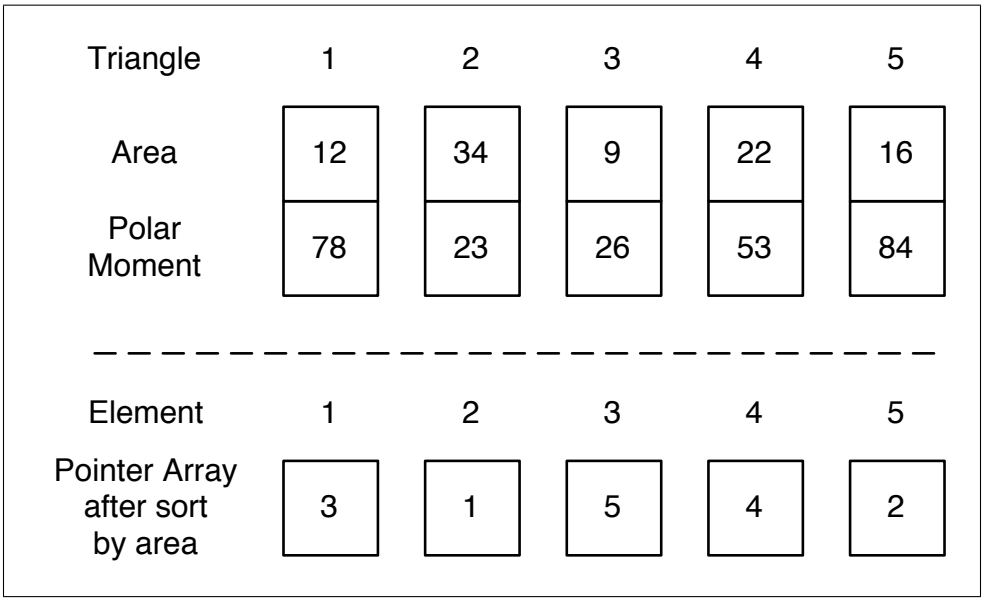

Figure 3.20.: Sorting Pointers by Area Instead of the Triangles

separate sorted catalogs would have been required. To keep from having to physically moving the elements of the catalog around, a pointer array was used instead, as shown in Figure 3.20. As the triangles are sorted, a pointer array is created which points to the triangles in order of area. The first element of the pointer array will point to the triangle with the smallest area, the second to the triangle with the next-smallest area and so forth. Matlab does not directly support pointers, and could be implemented more easily using C or Pascal. [17]

3.7.1. Binary Tree for Sorting

The sorted pointer array is created by using a structure called a binary tree such as shown in Figure 3.21. It is similar to the quad-tree structure, except that each node in the tree only splits in two directions, left and right.

To add items to the binary tree, a root node must first be created; it will be the first

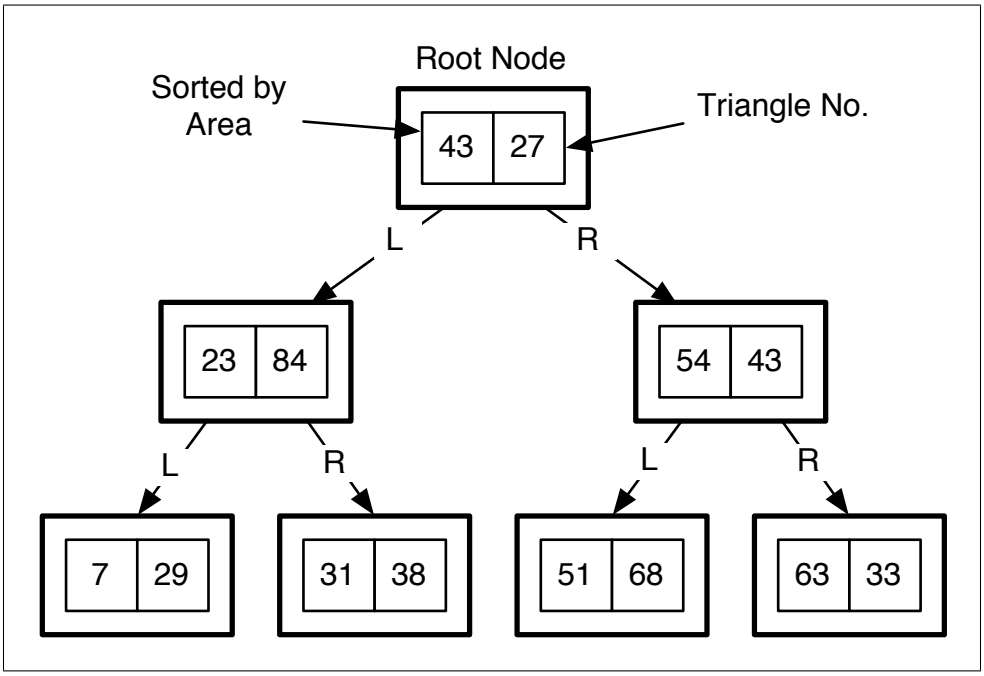

Figure 3.21.: Binary Tree of Triangle Pointers Sorted by Area

item to be stored in the tree. If the next item to be stored is less than the root, it is stored in a node that branches to the left side of the root. If the item is greater, it is stored in a node that branches to the right side of the root. As items are added to the lists, they move down the tree, left and right depending on their value relative to the value in the nodes they encounter. When the bottom of the tree is reached, the items are added to it.

In the case for spherical triangles, the triangle numbers as well as the area by which they are sorted are added to the tree. If the triangle areas in the catalog are stored in a random fashion, a relatively "even" binary tree will result, whose depth can be approximated by the following equation:

$$2^n = x$$

where n = number of levels and x = number of elements

There are 662,799 triangles which need to be sorted, which means the binary tree will be about 19 levels deep. To find any spherical triangle with a particular area in the binary tree will not require more than 19 comparisons, which for such a large amount of data is very efficient.

One the triangle numbers are stored in the binary tree they can be recalled in order of spherical area using a recursive algorithm. This algorithm traverses the binary tree in order of area, starting at the left-most branch. As the tree is traversed the pointer array is constructed. When finished, the elements of the pointer array will point to the triangles in order by area.

3.8. K-Vector Construction

A binary tree is a very efficient method for searching through large amounts of data, but can be improved upon. If a mathematical equation can be fit to the sorted elements of the

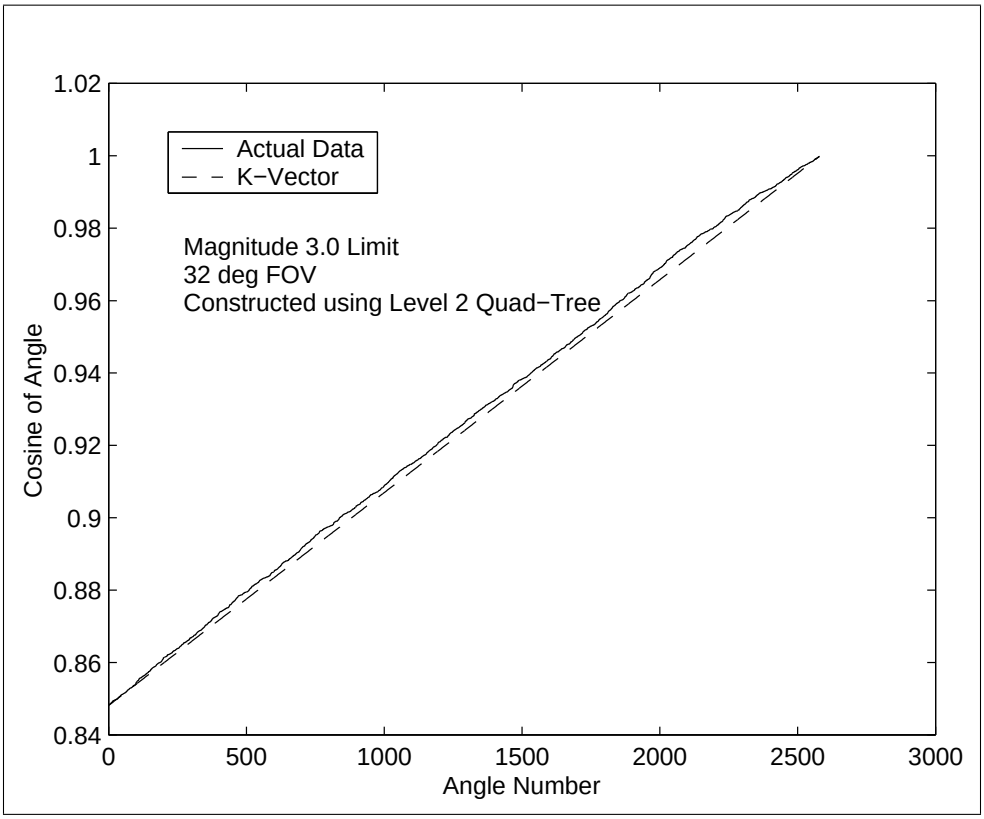

Figure 3.22.: Cosine of Angle Plotted Against Angle No. is Almost Linear

catalog, searching for items that fall within a specific range can be reduced to a single test. This method was using by Mortari [18] to find specific angles between stars for the angle method, and will be employed here for both the angle and spherical triangle method.

3.8.1. Linear K-Vector for Angles

Mortari sorted the catalog of angles that would fit within a 32 deg field of view star tracker sensitive to magnitude 3.0 stars, the cosine of each angle against its location in the catalog. The result is nearly linear, as shown in Figure 3.22. A line can be drawn from the first point to the last point, its slope and intercept calculated:

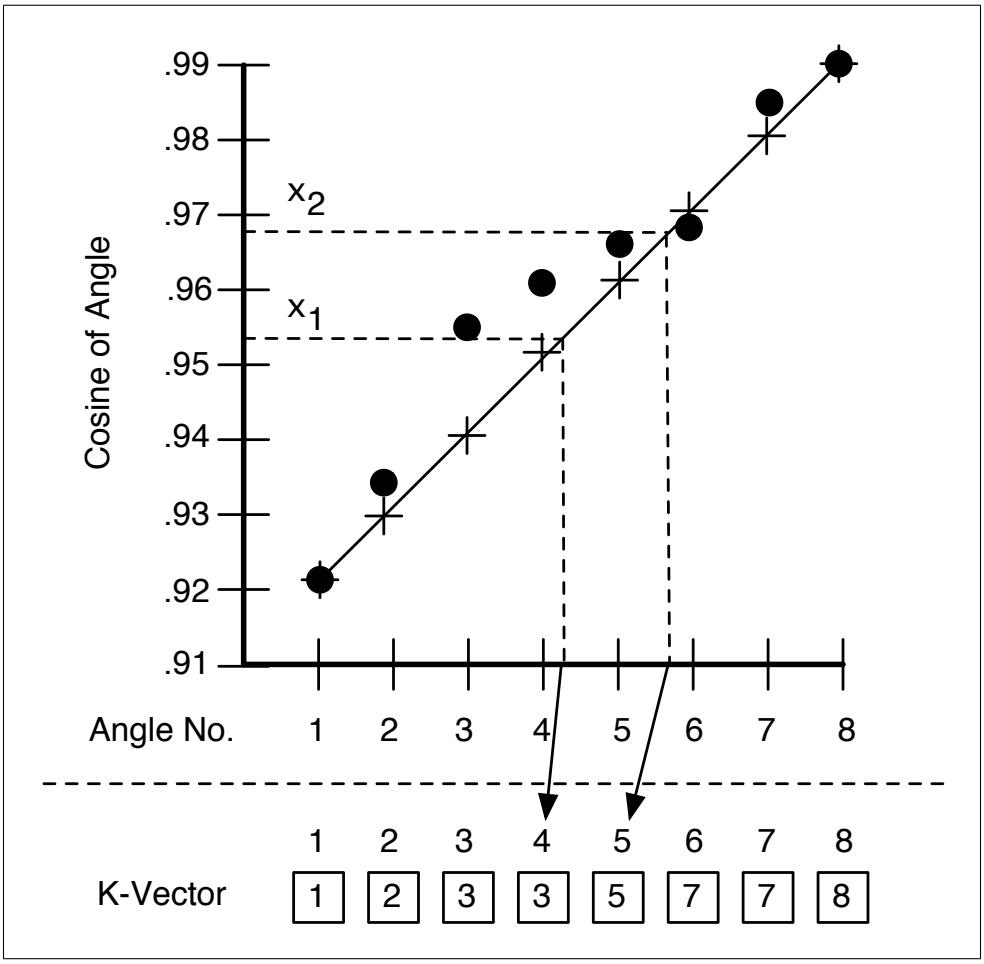

Figure 3.23.: Demonstration of K-Vector Construction

Once the line is determined, an array called the K-Vector is created. Each element of the K-Vector will point to the first angle that has a cosine greater than the cosine calculated from the line connecting the first and last points. However, the first and last elements of the K-Vector will always point toward the first and last spherical angles, respectively. This is demonstrated in Figure 3.23.

The first element of the K-Vector points to angle number 1, which is set as a rule. The second element of the K-Vector points to angle number 2, because it is the first angle whose cosine has a value above what would result from the equation of the line. The third and fourth elements both point to angle number 3, because the cosine of angle 3 has is above both points above K-Vector element 3 and 4 on the line. And so it goes until the K-Vector is filled.

The K-Vector becomes useful when all the angles with a range of cosines needs to be found. In this example, all the angle with cosines between x_1 and x_2 need to be found. Using the equation for the line, the K-Vector index can be found. Rounding downward, it can be seen that x_1 refers to K-Vector element 4, which then points to angle number 3. x_2 leads to K-Vector element 5 which points to angle number 5. The angles that have cosines between x_1 and x_2 are triangle numbers 3 through 5. With little searching, the triangles for a range of areas have been found.

For a star tracker with a field of view of 8 degrees, sensitive to magnitude 6, Figure 3.24 shows the cosine of the angle as a function of angle number. The result is still nearly linear, and the same method as employed by Mortari is employed here with one exception. Rather than using the angle numbers themselves, the angle pointer is used. This is because the angles in the catalog haven't been sorted, only the angle pointer array has. So when the triangles that fall between two bounds need to be found, the line that connects the first and last points is used to determine the range of angle pointers. The range of angle pointers

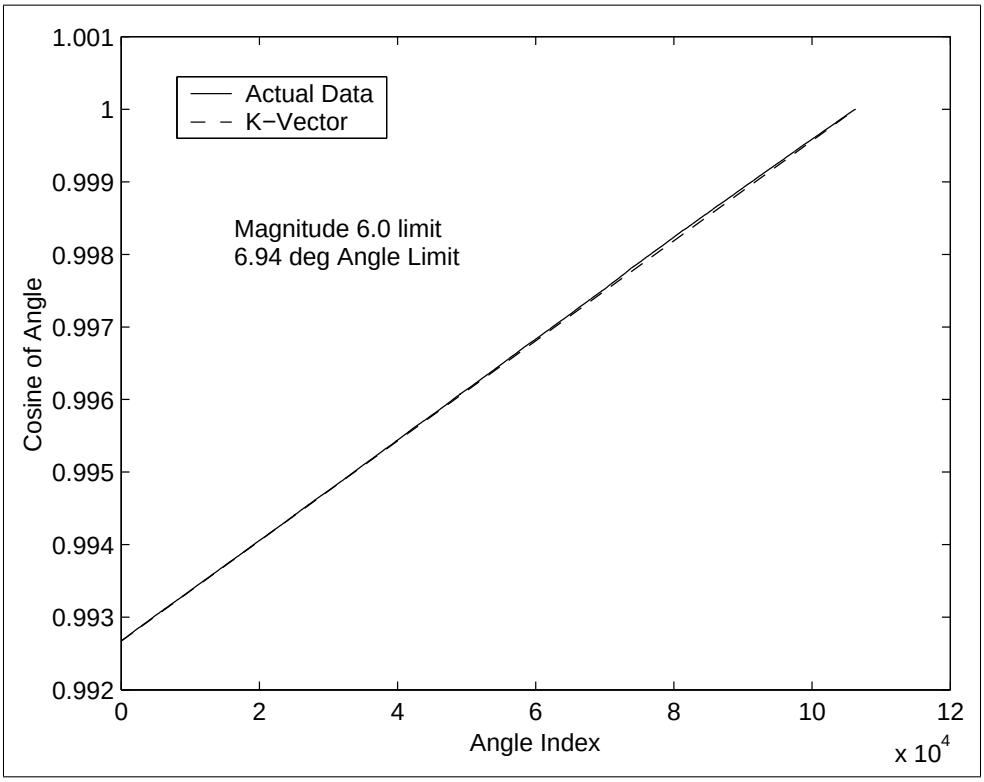

Figure 3.24.: Linear K-Vector for Angle Method

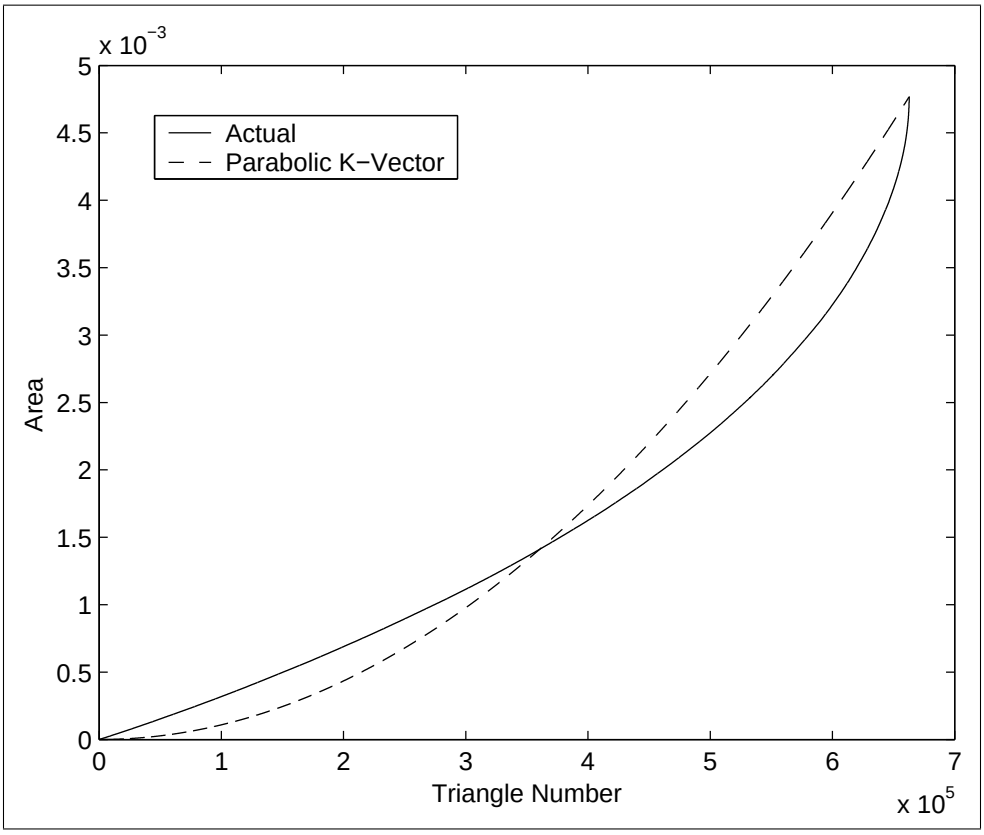

Figure 3.25.: Parabolic K-Vector for Spherical Triangle Method

then point to the individual angles that are being examined.

3.8.2. Parabolic K-Vector For Spherical Triangle Method

For the K-Vector Method to work efficiently requires the mathematical representation of the sorted data to fit closely to the actual data. If the sorted spherical triangle area is plotted against the triangle number, the resulting curve is shaped more like a parabola than a line, and so the equation used is parabolic. This can be seen in Figure 3.25. Although the fit is not nearly as precise as the linear K-Vector is for the angle catalog, it works well enough to not cause any significant change in results. Even if a linear K-Vector were constructed,

the effect would only be to either include a few spherical triangles that don't belong in the range, or to exclude a few that do. Since a 3σ range has been chosen, missing a few spherical triangle out of the hundreds that will lie within the range will not significantly affect the results.

As with the angle K-Vector, the spherical triangle K-Vector will be used to determine ranges of spherical triangle pointers. These pointers will then used to find the actual spherical triangle that has an area which falls into the area range being examined.

3.9. Running the Matlab Code

All of the Matlab code necessary to demonstrate both methods is presented in Appendix E, and the order for operation is presented in Figure E.1.

The file "CreateStarQuadTree.m," in Appendix E.17 is used to create the quad-tree of stars necessary to catalog the angles and triangles. The level and magnitude limit can be set at the top of the code, as well as the level of graphics output. The code, as set up, will create a four-level quad-tree of stars magnitude 6.0 and outputs a file called "QTM60L4.mat"

Once the quad-tree is constructed, the angles and spherical triangles can be cataloged. To create the angle catalog, the file "CatalogAngles.m," in Appendix E.11, should be run. As configured, it will catalog all the angles 6.94 degrees or less, and output the file "AngM60L4.mat." To create the K-Vector of the angles then requires running "SortAngles.m," in Appendix E.43.

To create the spherical triangle catalog requires running two pieces of code. First, "CatalogSphTris.m," in Appendix E.12 needs to be run. It will catalog all the spherical triangles that will fit within a 6.94 degree field of view and output the file "SphTriM60L4.mat." To add the area and polar moment measurements, the file "AddSphTriProps," in Appendix E.3 has to be run. It will output the file "SphTri2M60L4.mat." The K-Vector can now be cre-

ated running “SortSphTris.m,” in Appendix E.44. It will output ”SphTriPtrM60L4.mat.”

3.10. Summary

Two things are necessary for either the angle method and spherical triangle method: a catalog and a K-Vector for the catalog. For the angle method, the angles found in the field of view inside of the star tracker will be compared to the angles in the catalog. The K-Vector will be used to quickly determine which angles fit within the 3σ bound being placed of the measurement.

For the spherical triangle method, the spherical triangles found in the field of view inside of the star tracker will be compared to the spherical triangles in the catalog using the area and polar moment of the spherical triangle. The K-Vector will be used to quickly determine which spherical triangles have an area which fit within the 3σ bound being placed on the measurement. The polar moment will be used to quickly reduce the number of solutions that lie within the 3σ bound.

4. Testing the Angle & Spherical Triangle Method

4.1. Introduction

Once the catalog of angles and spherical triangles and their appropriate K-Vector arrays are prepared, the star pattern recognition algorithms can be tested. To test them, a random star tracker attitude is generated 1,000 times, and each star pattern recognition algorithm tested to see if it can positively identify at least one angle or spherical triangle in the field of view. If an angle or spherical triangle is identified, then the stars that make it up are known, and the attitude of the star tracker can be found. The star pattern recognition algorithm does not attempt to determine final attitude; it only tries to identify the stars that make up the angles or triangles.

If all of the stars determined to be in the star tracker's field of view by the star pattern recognition algorithm actually are in the field of view, it is considered a correct result. If any of the stars output by the algorithm are not in the field of view, the result is considered a failure. If the algorithm cannot identify any of the stars in the field of view, it is recorded as an inconclusive result.

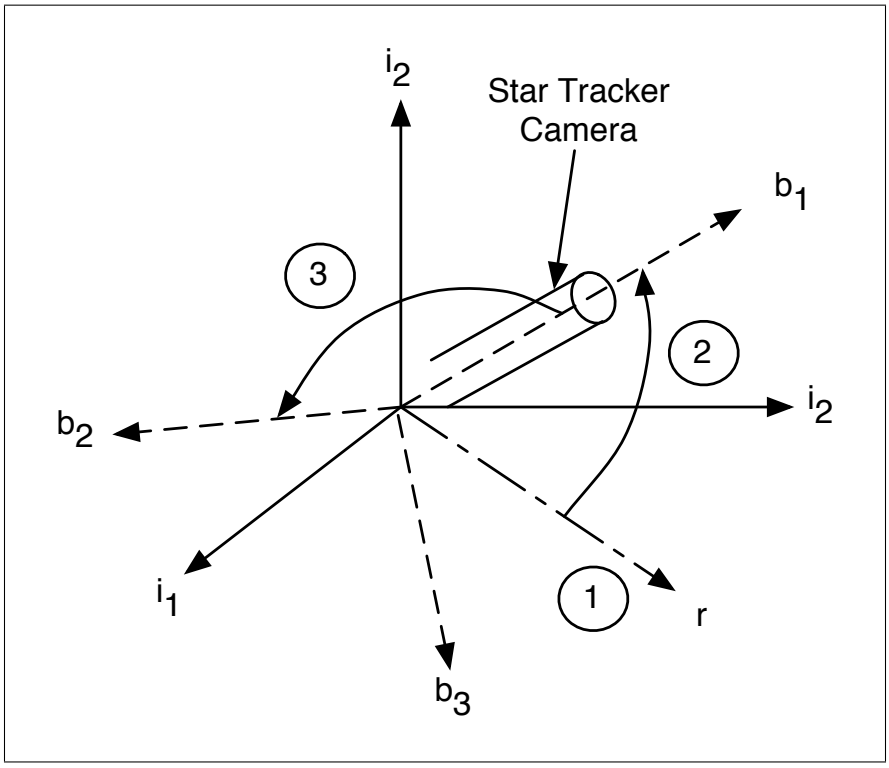

Figure 4.1.: Three Steps to Create a Random Body Frame

4.2. Star Tracker Random Attitude

Each time a star pattern recognition algorithm is to be tested, the imagined star-tracker must be set in a random attitude. Random unit vector $\mathbf{b}_1$, which represents the direction in space the star tracker camera is facing, is created. Two random angles are created and determine the direction of the vector in spherical coordinates. The azimuth is a random number ranging from 0 to 2π; the altitude is a random number ranging from $-\pi$ to π. These spherical coordinates are then converted to cartesian coordinates.

Once a random direction is established, a random frame based on it can be constructed. Figure 3.4 shows the procedure for creating a random frame for the star tracker given the

star tracker camera direction. First, a second random unit vector $\mathbf{b_r}$, is created in a manner similar to the first vector.

First a unit vector orthogonal to $\mathbf{b_1}$ must be created:

$$\bar{\mathbf{b}}_2 = \mathbf{b}_r \times \mathbf{b}_1 \tag{4.1}$$

$$\mathbf{b}_2 = \frac{\bar{b}_2}{|\bar{b}_2|} \tag{4.2}$$

Random unit vector $\mathbf{b_r}$ can now be discarded. The last axis, orthogonal to both $\mathbf{b_1}$ and $\mathbf{b_2}$ is created by crossing both:

$$\bar{\mathbf{b}}_3 = \mathbf{b}_1 \times \mathbf{b}_2 \tag{4.3}$$

$$\mathbf{b}_3 = \frac{\bar{b}_3}{|\bar{b}_3|} \tag{4.4}$$

Figure 4.2 shows a random star tracker frame and its 8 deg field of view in the sky. The quad-tree is then used to determine which stars need to be checked to determine whether or not they lie within 4 degrees (half of the field of view) of the star tracker's direction.

It is interesting to note, that since the distance being searched is 4 degrees, refereing back to Table 3.2, a four-level quad-tree is required, since a fifth-level quad-tree would only allow a search distance of 3.43 degree. Unfortunately, using a four-level quad-tree means searching a distance of 6.94 degrees, which is much larger than the 4 degrees necessary. Determining the stars within the field of view is not time critical, however, and is still much faster than scanning all the stars in the field of view. It is easy to see in Figure 4.3 that, although a large number of stars need to be checked, it is far less than would be required otherwise. Stars that are determined to be within the field of view are colored green; stars that are not it the field of view are colored red.

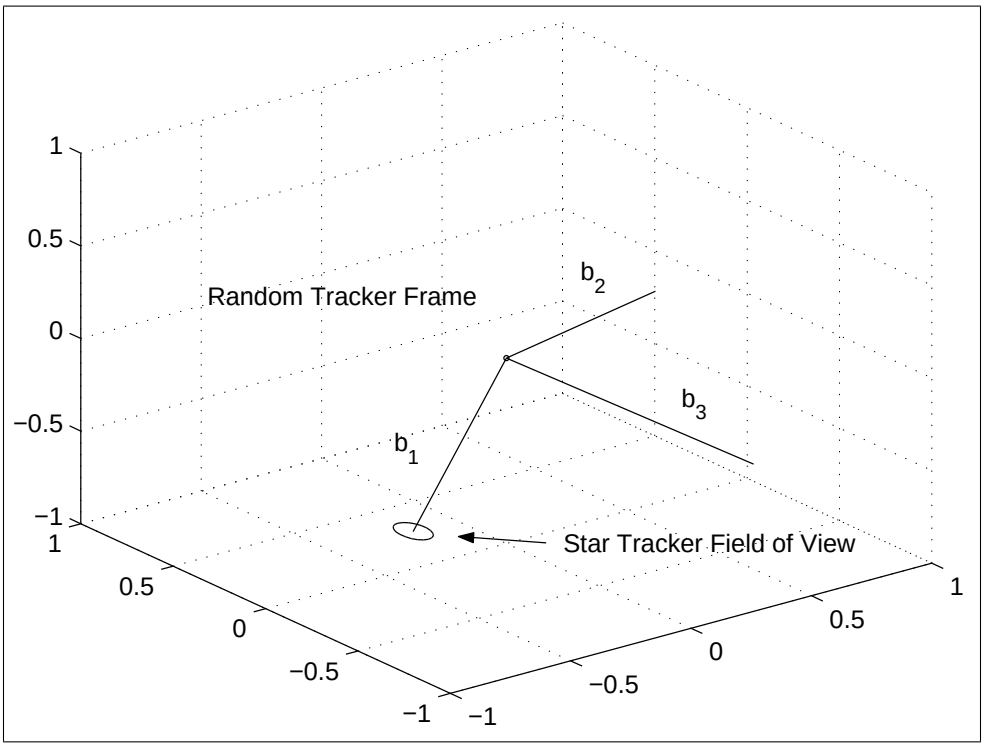

Figure 4.2.: Sample Random Star Tracker Frame

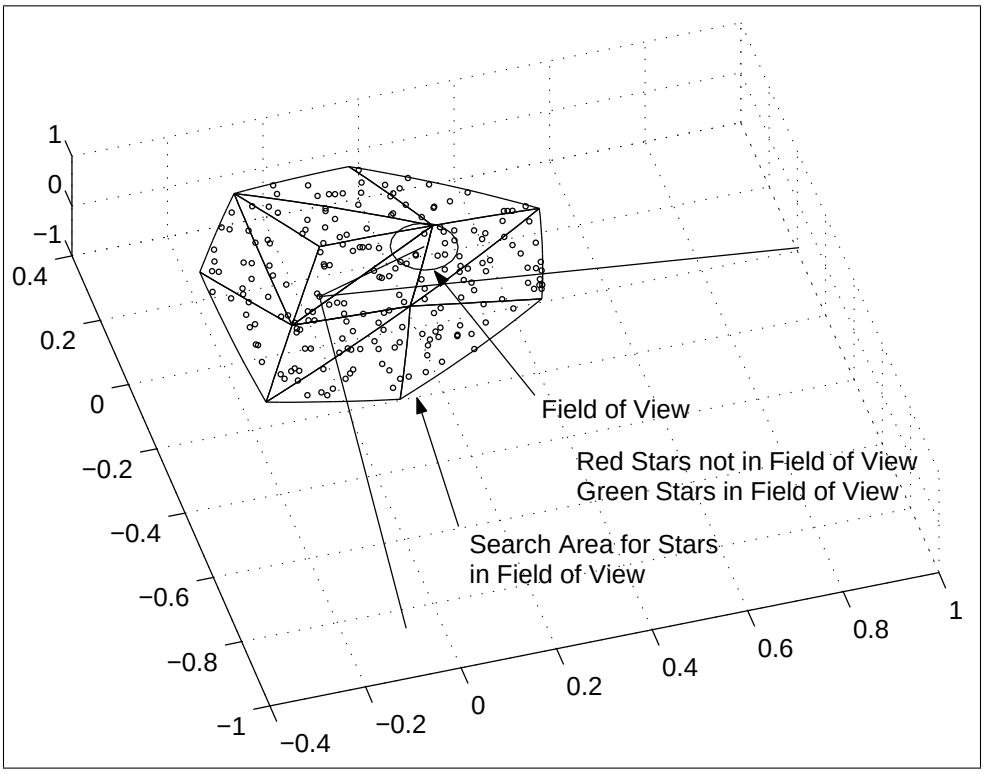

Figure 4.3.: Quad-Tree Used to Find Stars Within Star Tracker Field of View

4.3. Star Tracker Field of View

Once a the star tracker's random attitude has been determined and the stars with the field of view found, the positions of the stars have to be changed to the star tracker frame of reference. This is to to demonstrate that the properties of the triangles cataloged are not dependent on the attitude in which the triangles properties were calculated. Given the body axis in terms of the internal coordinate frame, the location of the stars in the ECI coordinate frame can be converted to the frame of the star tracker using the following coordinate transformation equation: [19]

$$\mathbf{v}_b = \begin{pmatrix} \mathbf{i}_x \cdot \mathbf{i}_\xi & \mathbf{i}_x \cdot \mathbf{i}_\eta & \mathbf{i}_x \cdot \mathbf{i}_\zeta \\ \mathbf{i}_y \cdot \mathbf{i}_\xi & \mathbf{i}_y \cdot \mathbf{i}_\eta & \mathbf{i}_y \cdot \mathbf{i}_\zeta \\ \mathbf{i}_z \cdot \mathbf{i}_\xi & \mathbf{i}_z \cdot \mathbf{i}_\eta & \mathbf{i}_z \cdot \mathbf{i}_\zeta \end{pmatrix} \mathbf{v}_i \tag{4.5}$$

The coordinates of the stars are so far still true, which would not be reported by a real star tracker. A positional error with a standard deviation of 87 microradians exists, and has to be added to the positions of each star within the star tracker's field of view. This is done by adding a normal random vector to the true vector, then making the resultant a unit vector. This result distribution of measurements is not perfectly normal but sufficient for these tests.

$$\tilde{\mathbf{v}} = \mathbf{v} + 3 \cdot \text{randn}(3,1) \cdot \sigma \tag{4.6}$$

then

$$\tilde{v} = \frac{\tilde{\mathbf{v}}}{|\tilde{\mathbf{v}}|} \tag{4.7}$$

Figure 4.4 shows the field of view of the star tracker. The stars, which are positioned three-dimensions are projected against a plane and their two-dimensional coordinates used to plot it. Circles indicate the true position of the stars, and crosses indicate the positions

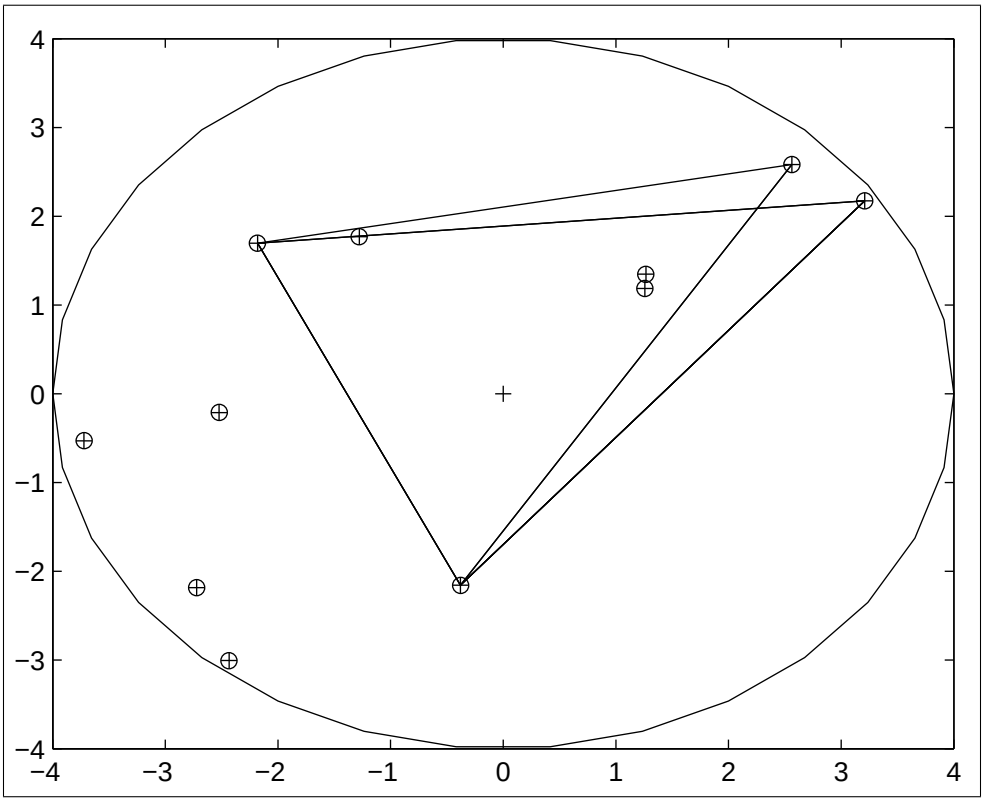

Figure 4.4.: Stars Within Star Tracker's Field of View

with error added to the measurement.

These values are now given to the star pattern recognition algorithms for testing.

4.4. Running the Matlab Code

To test either the angle method or spherical triangle method requires running "RandAttTest.m," in Appedix E.41. How the method variable is set will determine which method is used. The number of tests is currently set to 1,000, but can be easily changed. Pivot and possible solutions limits can be changed, as well as the level of graphics output. The program will output the results of the tests to a file called "Results.mat."

4.5. Conclusion

The code required to test the testing algorithms is almost as complex as creating the angle and spherical triangle algorithms themselves. The importance of thoroughly testing the testing method cannot be understated. Many smaller pieces of code, not presented here, were used to test each section of the testing routines to be sure the results of testing can be trusted.

5. Star Pattern Recognition Algorithms

5.1. Introduction

In this chapter, the data structures and programming methods for implementing the angle and spherical triangles methods are discussed. Since there are many similarities between the angle and spherical triangle method, the code written for each method was deliberately kept as similar as possible, to make time differences between the two a function of their methods, not their code.

5.2. Angle Method

5.2.1. Introduction

The angle method is presented so that the spherical triangle recognition algorithm can be compared to it. The entire algorithm can be divided into three distinct sections: Angle listing, pivot ordering and angle matching.

5.2.2. Listing the Angles in The Field of View

The first thing the angle recognition algorithm has to make a list of all the angles that can be made from the stars within the field of view of the star tracker. Since the star tracker has a field of view of 8 degrees and the angle catalog only contains angles to 6.94 degrees,

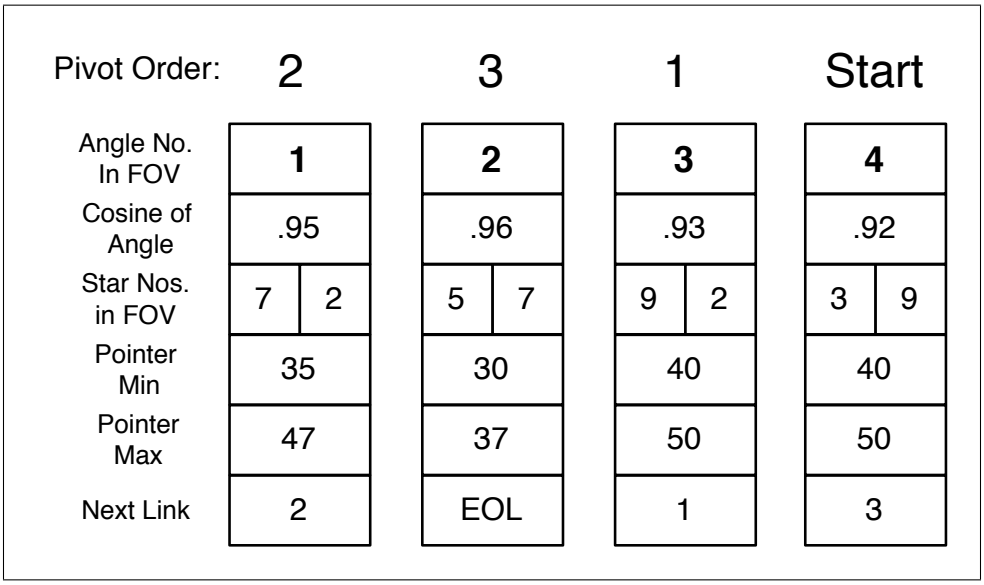

Figure 5.1.: Stars Within Star Tracker's Field of View

and angles larger than 6.94 degrees in the field of view are rejected. As angles are added to the list, the 3σ bounds of the angle measurement are calculated. Using the K-Vector made of the angles, a range of angle pointers is found and stored with that angle on this list.

5.2.3. Determining the Order of Pivots

To reach a result in the most efficient manner, the largest angle cataloged in the field of view is the first to be examined, since it will have the fewest number of possible solutions. The remaining angles are then ordered according to two rules: First the angle must share one star in common with the previous angle. Second, of all the angles that meet the first rule, the largest angle should be chosen. This list continues until either the limit on pivots is encountered, or there are no more angles meet the above two requirements.

To create the pivot order, the list of angles in the field of view is linked a shown in Figure 5.1. The largest angle is angle 4 (which has smallest cosine), is linked to angle 3,

which shares one star in common. Angle 3 can either pivot to angle 4 or 1, since each share one star in common with angle 3. Angle 1 has a larger angle, and so angle 3 will pivot to angle 1. Angle 1 then finally pivots to angle 2. The order for pivoting in this case is 4, 3, 1 and 2. Note that "EOL" indicates "end of list."

This can be a time-consuming operation if no pivot limits are established. For an 8 degree field of view star tracker sensitive to magnitude 6.0 stars, there can be hundreds of possible angles to examine and sort. It will be seen later that the number of pivots can be limited to improve the CPU time required without significantly impacting the probability of reaching a solution.

5.2.4. Pivoting Toward a Solution

Assuming there exists more than one solution to the first triangle, a pivots will have to be made to reach a single solution. All the possible solutions for the first angle are considered "finalists," and will be compared to the possible solutions to the second angle, the "pivot angle". When an angle from the finalists is found to have a star in common with a solution to the pivot angle, it is put into a new list of finalists. After all the comparisons are made, if there is still more than one finalist, another pivot has to be made.

An example is shown in Figure 5.2. Here it can be seen that between the finalists and the pivot angle, only three possible pairs of angles share a star in common: pairs 10 & 27, 46 & 54 and 66 & 54. The new finalists, 10, 27 and 56 carry along the previous finalists, 27 and 54 (twice) because when a single solution is reached, all the angles that were used to reach the solution will be known. Since there are still three new finalists, another pivot will be made, and angles 10, 46 and 66 will be compared to possible solutions to that pivot angle, and the process continued.

Note that Figure 5.2 lists the actual angle numbers in the catalog. If the angle pointers

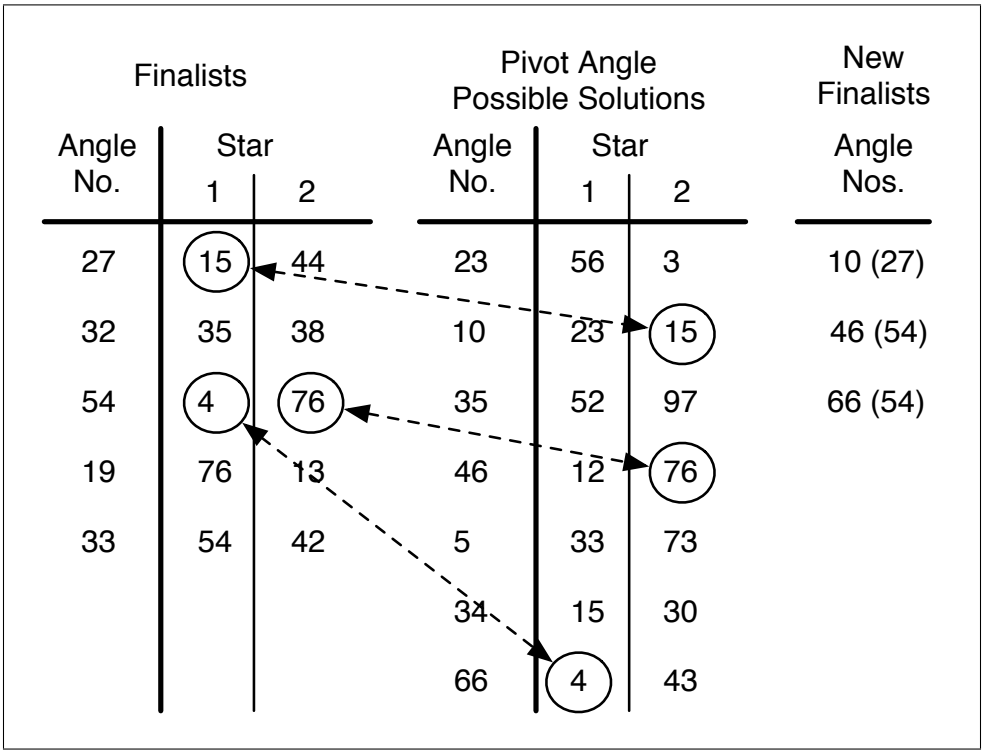

Figure 5.2.: Example Of Angle Pivoting To Approach A Solution

as calculated from the K-Vector were listed, they would have been sequential. For instance, the angle pointer for the finalists might have ranged from 41 to 45, and they would have pointed to triangles 27, 32, 54, 19 and 33 respectively.

5.2.5. Using Binary Trees for Angle Comparisons

The easiest way to do the comparisons between the finalists and pivot angle possible solutions is to compare each finalist to each of the pivot angle possible solutions. If there are x finalists, and y pivot angle possible solutions, there are $x \times y$ combinations to test. It is likely that there will be hundreds of finalists and pivot angle possible solutions, so there will be a very large number of combinations to test, slowing the algorithm greatly.

Instead, two binary trees are constructed. The first tree contains all the possible solutions to the pivot angle, sorted by the first star that makes it up; the second tree contains all the possible solutions to the pivot angle sorted by the second star. Each of the binary trees are queried to see if any of the angles stored in it have a star that matches the stars which make up each finalist. Any that do are recorded to the new finalists.

Although a lot of CPU "overhead" is required to construct the trees, the number of tests that need to be performed is greatly reduced. For instance, if there are 150 pivot angle possible solutions, the binary tree would be only 7 levels deep. If there are 100 finalists, there would only be 700 tests to be made, rather than 15,000. These numbers are conservative; there are often hundreds of finalists and pivot angle possible solutions.

5.2.6. Angle Method Output

If a single solution is reached, the angle recognition algorithm compiles a list of stars which make up the angles solved other wise it reports that a single solution was not reached. If a solution is returned to the random attitude testing algorithm, it checks to make sure every

star reported as found actually exists in the field of view. If all the stars reported by the algorithm actually are in the field of view, the result is called a correct result. If any star reported as being in the field of view is not, the test is considered a failure. If the algorithm does not report a solution, it is either because there are too few stars in the field of view, or it could not reach a single solution, and is recorded by the testing algorithm as such.

5.3. Spherical Triangle Recognition Algorithm

5.3.1. Introduction

The spherical triangle recognition algorithm is very similar to the angle recognition algorithm with the exception that two properties, area and polar moment, are being used to approach a solution instead of only the cosine of an angle. This method too can be divided into three distinct sections: spherical triangle listing, pivot ordering, and triangle comparison.

5.3.2. Listing the Spherical Triangles in the Field of View

This section of the star pattern recognition algorithm creates a list of all the possible combinations of triangles that can be created from the stars within the field of view of the star tracker. Spherical triangles that occupy a field of view greater than 6.94 degrees are rejected, since the catalog will only contain spherical triangles that size or smaller. As the list is created, the area of the spherical triangle is calculated and a note is made of the spherical triangle with the greatest area in the field of view.

Entering the upper and lower bounds of area into the K-Vector equation results in a range of triangles pointers, which point to triangles, one of which will hopefully be the correct triangle. The upper and lower end of the triangle pointers are stored with the area and will be needed for pivoting.

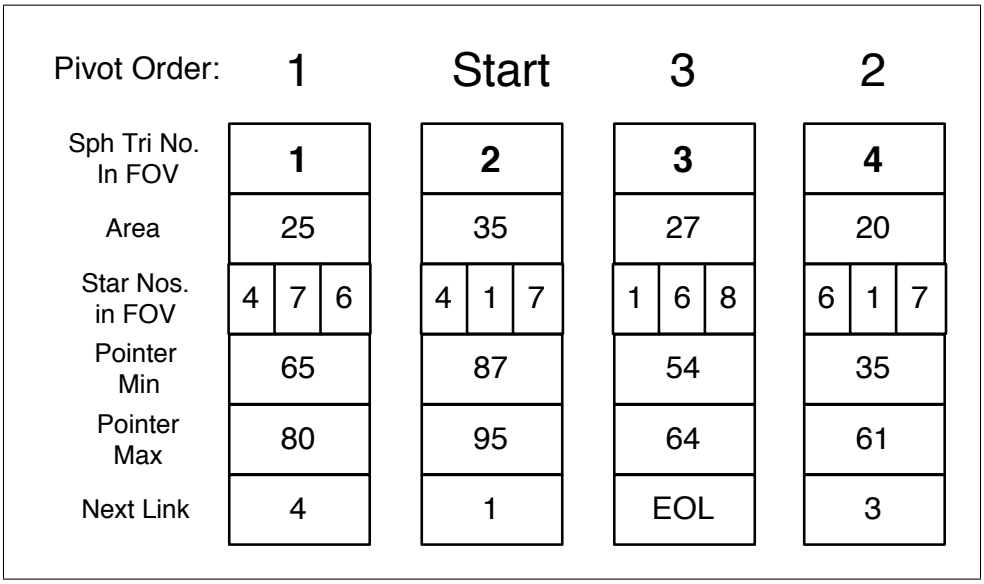

Figure 5.3.: Pivot Order for Spherical Triangles

5.3.3. Pivot Sorting

The first triangle to be examined will be the triangle with the largest area. The triangles with the largest area tend to have the smallest range of possible solutions, which helps to reach a singular solution using the least number of pivots. The list of triangles in the field of view are linked in order of pivots, similar to how is done with the angle method.

To pivot, from one triangle to another requires they share two stars in common. If there are more than one triangle that qualify, the one with the largest area is selected. The linked listed continues, until either the limit on pivots is encountered, or there are no more triangles to which can be pivoted. No triangle can be used more than once.

An example of this is shown in Figure 5.3. It can be seen that triangle 2 has the largest area, and so the comparisons start with this triangle. Triangle 2 could be linked to either triangle 1 or triangle 4, since both share two stars in common with the triangle 1, but triangle 1 has a greater area. Triangle 1 then gets linked to triangle 2 which also happens

to share two stars in common with triangle 1 and then triangle 2 is linked to triangle 3. The order for pivoting is then: 2, 1, 4 and 3.

It is interesting to note that triangle 3 is last despite having a greater area, and thus fewer combinations to examine, than triangles 1 and 2. It is unfortunate that this triangle could not be better utilized, but there is no other way to pivot to it if the first triangle used is triangle 2.

5.3.4. Spherical Triangle Comparison

Starting with the first triangle in the linked list, if it turns out there exists only one spherical triangle in the catalog that has an area that fits between the upper and lower limits of the bounded area, that spherical triangle is likely to be the solution, and the star pattern recognition algorithm can stop. With a 3σ bound, there are often hundreds if not thousands of possible solutions for every triangle in the field of view, and the algorithm will never end here.

This is where the polar moment helps to further reduce the number of solutions. The polar moment of the first triangle in the linked list is calculated. Its 3σ bound is calculated and a lower bound on the polar moment established. A new list, called "finalists" is created, and any of the possible solutions for the first triangle which has a polar moment within the bounds established is added to the list. Then if there is only one "finalist" triangle, the solution is assumed to have beenfound and the star pattern recognition algorithm can stop here without any pivoting. It will be seen later that this still does not occur, and at least one pivot is always required. What is surprising though, is how few triangles do become a finalist. Starting with hundreds or thousands of possible triangles that fall within the area bounds established for the target triangle, the number of triangles that become finalists may only be a hundred or less. It will have a great impact on the CPU time required to

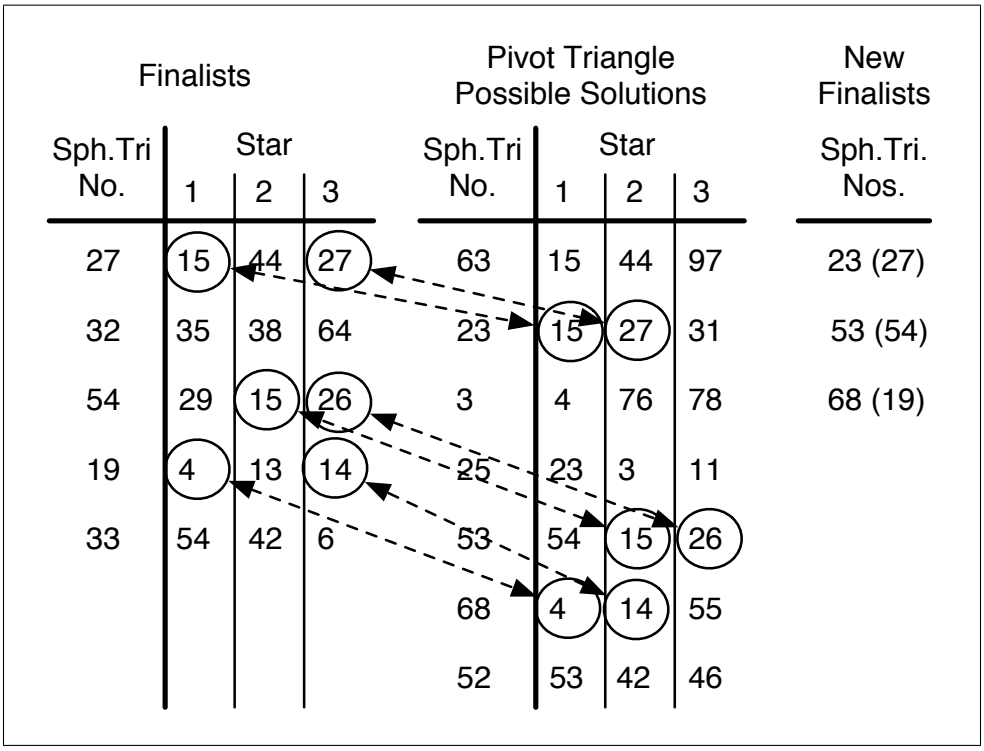

Figure 5.4.: Spherical Triangle Pivot to Approach a Solution

reach a solution.

5.3.5. Pivoting Toward a Solution

If the area and polar moment of one spherical triangle is not enough to establish its identity, a pivot must be done to further reduce the number of possible solutions. Following the linked-list for pivoting, the next "pivot" triangle is selected. Like the first spherical triangle, its area, polar moment and range of possible solution triangles is determined. A list of finalists for the pivot triangle is created, where only the possible solutions that have polar moments within the range established by the pivot triangle are kept.

What is important here, is that the first and second triangles share two stars, and so

must the correct solution. Only the finalist and pivot triangle possible solutions that share at least two stars are kept, as shown in Figure 5.4.

5.3.5.1. Binary Tree Star Comparison

Instead of testing every finalist against every possible solution for the pivot triangle, binary trees are used. While the angle method uses two trees, the spherical triangle method uses three. The first tree contains all the possible solutions to the pivot triangle, sorted by the first star that makes it up. The second tree contains all the possible solutions to the pivot triangle sorted by the second star, and the third tree sorted by the third star. Each of the finalist's stars are tested among the three binary trees to see if any triangles between the two lists share two stars.

5.3.6. Spherical Triangle Method Output

The possible results of the spherical triangle method are similar to the angle method. If a single solution is reached, the spherical triangle recognition algorithm compiles a list of stars which make up the spherical triangles solved other wise it reports that a single solution was not reached. If a solution is returned to the random attitude testing algorithm, it checks to make sure every star reported as found actually exists in the field of view. If all the stars reported by the algorithm actually are in the field of view, the result is called a correct result. If any star reported as being in the field of view is not, the test is considered a failure. If the algorithm does not report a solution, it is either because there are too few stars in the field of view, or it could not reach a single solution, and is recorded by the testing algorithm as such.

5.4. Conclusions

It can be seen that the spherical triangle method, like many other star pattern recognition algorithms, has its roots based on the angle method and relies on pivoting to reach a solution. Because of this, the structures used in both methods are quite similar and will help make time comparisons between the two methods more useful in determining their worth. All of the code is presented in Appedix E.

6. Results

6.1. Introduction

In this chapter both the angle method and the spherical triangle results are compiled. The success and failure rates will be determined and the time required for both methods measured.

The CPU times stated are based on the hardware and software used, shown in Table 6.1. All graphics output from the routines were tuned off, and care was taken to ensure there were no other significant processes running in the background during testing. Each method was reduced to a single subroutine called from the random attitude testing routine. The time is measured from the line of code directly before calling the routine to the time it returns from it. The Matlab variable "CPUTIME" is used.

Both the angle and spherical triangle method star pattern recongnition algorithms were run with only one limit: if the number of possible solutions to any particular angle or

Table 6.1.: Hardware and Software Used for CPU Time Measurements

Item
Apple Powerbook G4, 500 MHz, 512 MB RAM
Apple OS X, Version 10.3.2
Apple X11, Version 1.0
Matlab Version 6.5.0.1951, Release 13

triangle exceeded 10,000, the angle or triangle is rejected. Despite the speed using binary tree structures for pivot comparisons offers, at this point, the time required for a solution becomes impractical and generally do not contribute to successful results.

6.2. Running the Matlab Code

The charts and results presented here are constructed from the "Results.mat" file using the "CompileResults.m" file, listed in Appendix E.13. The axis limits and and other details of the graph output can be controlled at the top of the code.

6.3. Angle Method Results

All the graphs output by Matlab for the angle method without pivot limits are given in Appendix A. All the results for the angle method with a 9-pivot limit are given in Appendix B.

6.3.1. Overall Success Rate

Refering to Figure 6.1 is can be seen that the angle method is successful in identifying at least one angle in the field of view only 55% of the time and cannot positively identify any of the angles 45% of the time, even when allowed virtually unlimited pivots. Less than one percent of the attitudes tested contained less than two stars within the field of view, for which the angle method could not be tested and less than one percent of the attitudes were incorrectly identified.

Since every possible triangle has been cataloged, the only way an incorrect result can occur is if the correct solution lies outside the 3σ bound placed on the angle recognition algorithm. Since the probability of the angle being within a 3σ bound is 99.7%, the errors

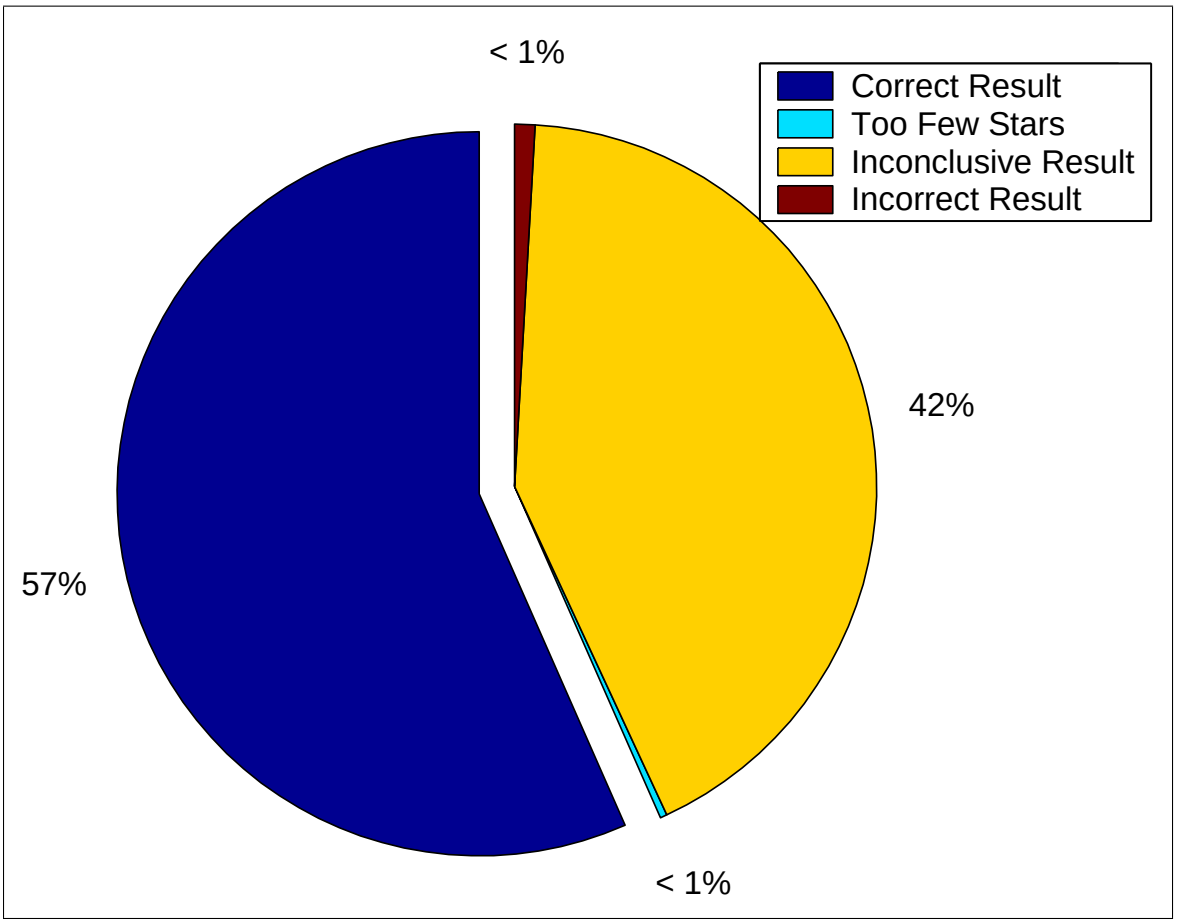

Figure 6.1.: Overall Results for Angle Method without Pivot Limit

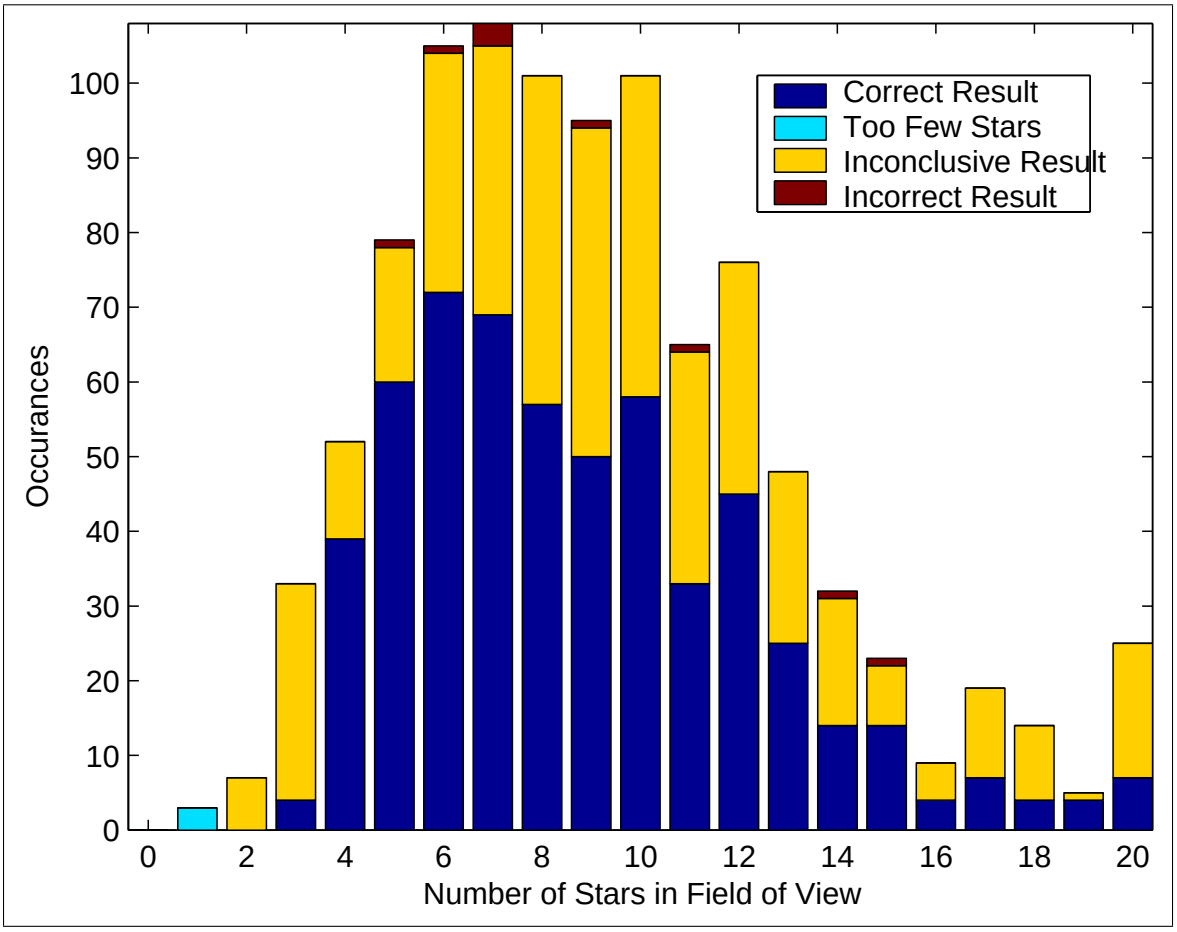

Figure 6.2.: Distribution of Results for Angle Method without Pivot Limit

encountered are not completely unexpected.

Refering to Figure 6.2, it can be seen that the angle does not work if there are less than 3 stars in the field of view. This implies that at least one pivot is required before any attitudes could be identified. Even with three stars in the field of view, the angle method could only positively identify at least one angle only a small fraction of the time.

Once four stars are present in the field of view, the angle method begins to show more success. If four stars are present, the angle method is successful 75% of the time, and

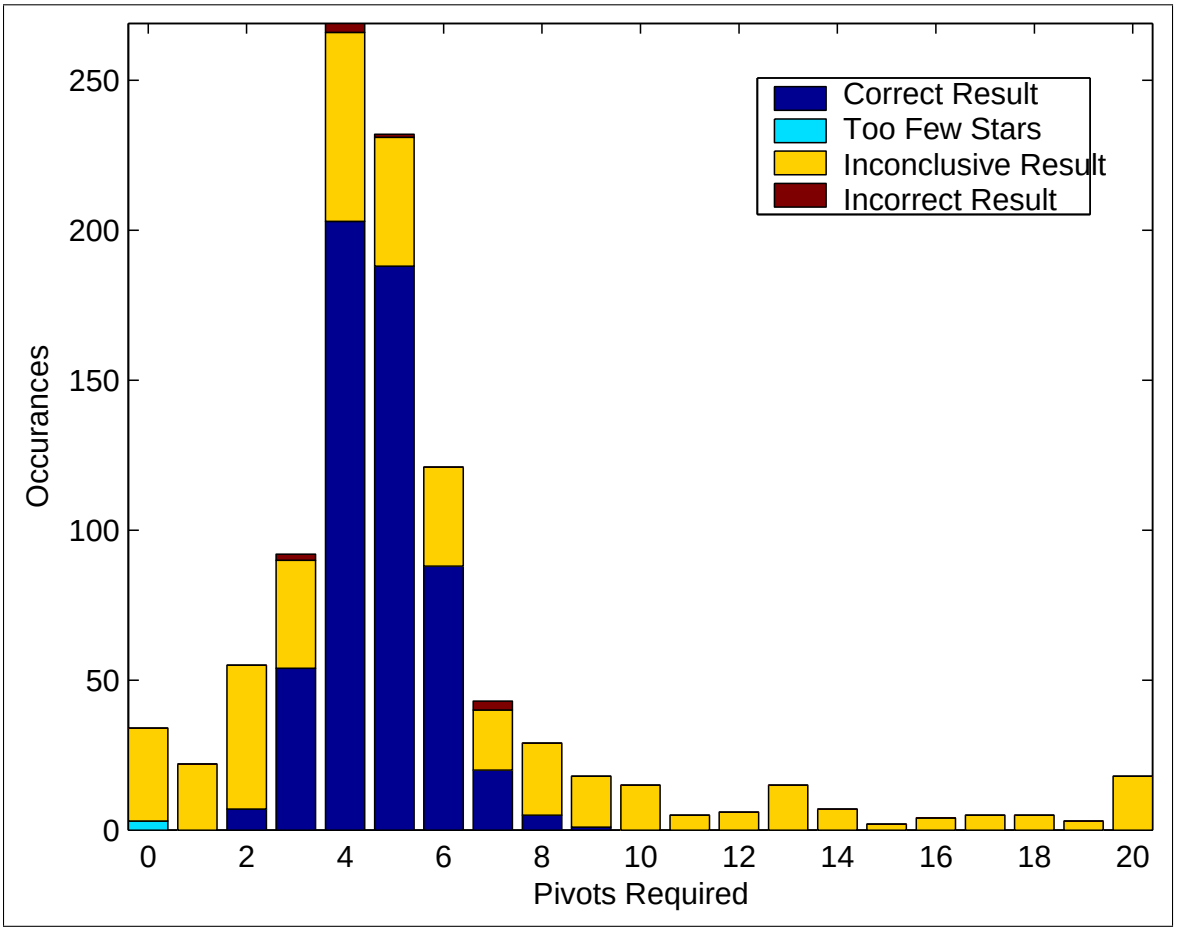

Figure 6.3.: Pivots Required for Angle Method without Pivot Limit

becomes more successful as more stars enter the field of view. But the rate of success drops again after 7 stars.

6.3.2. Pivots Required

Refering to Figure 6.3, the number of pivots required by the angle method is shown. The average number of pivots is 5.48, with a standard deviation of 4.41, including the short spike at the left side of the chart. This spike is a result that no pivoting occurs if there are

less than 3 stars in the field of view.

It is interesting to note that any attitude that required nine or more pivots would ultimately not be solvable. The angle method attempts pivots from largest angle to smallest angle, since the largest angles tend to have the smallest number of possible solutions. After a certain number of pivots, the number of possible solutions for an angle becomes so great, the pivoting no longer reduces the possible number of solutions. In some cases, the number of solutions can begin to rise.

In addition, the angle method will not yield a correct result if it has not achieved a correct result within nine pivots. The time efficiency of the method may be improved by limiting the maximum number of pivots to nine. The result is discussed in the next section.

6.3.3. CPU time required

The CPU time required to execute the angle method is shown in Figure 6.4. The average time required is 7.67 seconds, with a standard deviation of 3.60 seconds. As with the pivot chart, this chart shows a short spike at the left side, a result of requiring at least 3 stars in the field of view before a pivot can be made. Since no pivots can be made in those situations, little CPU time is required. Another spike exists at the right side of the chart. All the attitudes requiring more than 20 seconds are grouped together to make the chart more compact.

Since a significant amount of time is spent by the angle recognition algorithm determining the order for pivoting, the time results can be improved by limiting the number of pivots which the algorithm can take. It was shown in the last section that if an attitude requires more than nine pivots, the solution would not be found, so the algorithm has been tested limiting the number of pivots to nine. The resulting graphs look similar to testing without pivots and so are not included here, but are available in Appendix B. The overall success

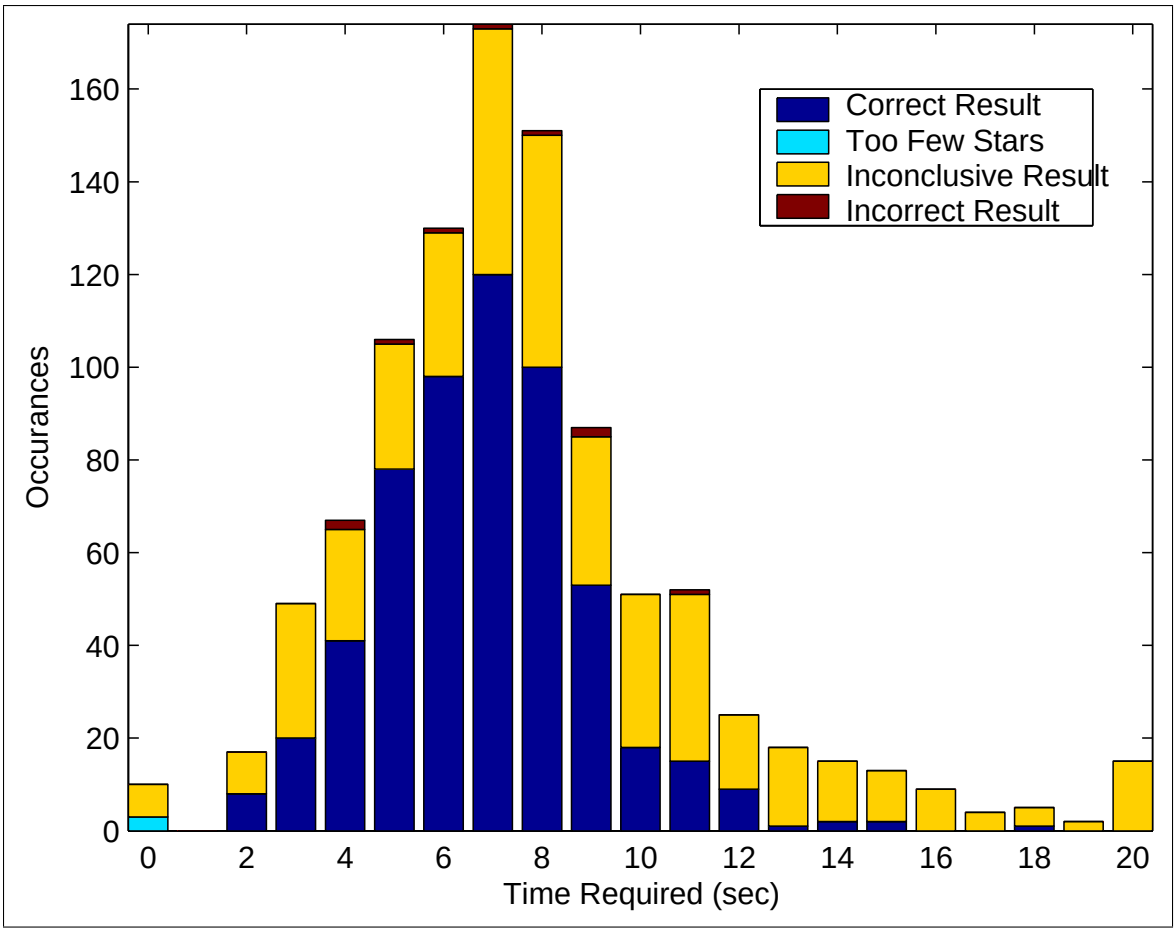

Figure 6.4.: CPU Time Required for Angle Method without Pivot Limit

rate remains 55%, but the average time required per attitude dropped to 6.89 seconds, a 10% improvement. The standard deviation also drops to 2.37 seconds.

6.4. Spherical Triangle Method Results

All the results for the spherical triangle method without pivot limits are given in Appendix C. The results for the spherical triangle method with a 3-pivot limit are given in Appendix D.

6.4.1. Overall Success Rate

The overall results for the spherical triangle method is shown in Figure 6.5. This method is able to obtain a successful result 92% of the time and is not able to positive identify any spherical triangles 7% of the time. Less than one percent of the attitudes contained less than 3 stars in the field of view, and had little impact overall. In addition, less than one percent of the attitudes are incorrectly identified, which is to be expected due to the 3σ bound placed on the spherical triangle search.

The overall result as a function of the number of stars present in the field of view is shown in Figure 6.6. This method cannot be used when there are less than three stars in the field of view, and is never able to determine any of the triangles in the field of view with less than four stars, meaning that at least one pivot was always required.

Once there are four stars in the field of view, the spherical triangle method is capable of recognizing at least one of the triangles 73% of the time. If there are at least four stars in the field of view, the spherical triangle method will arrive at the correct solution 92% of the time; if there are five or more stars in the field of view, the correct solution is attained 95% of the time.

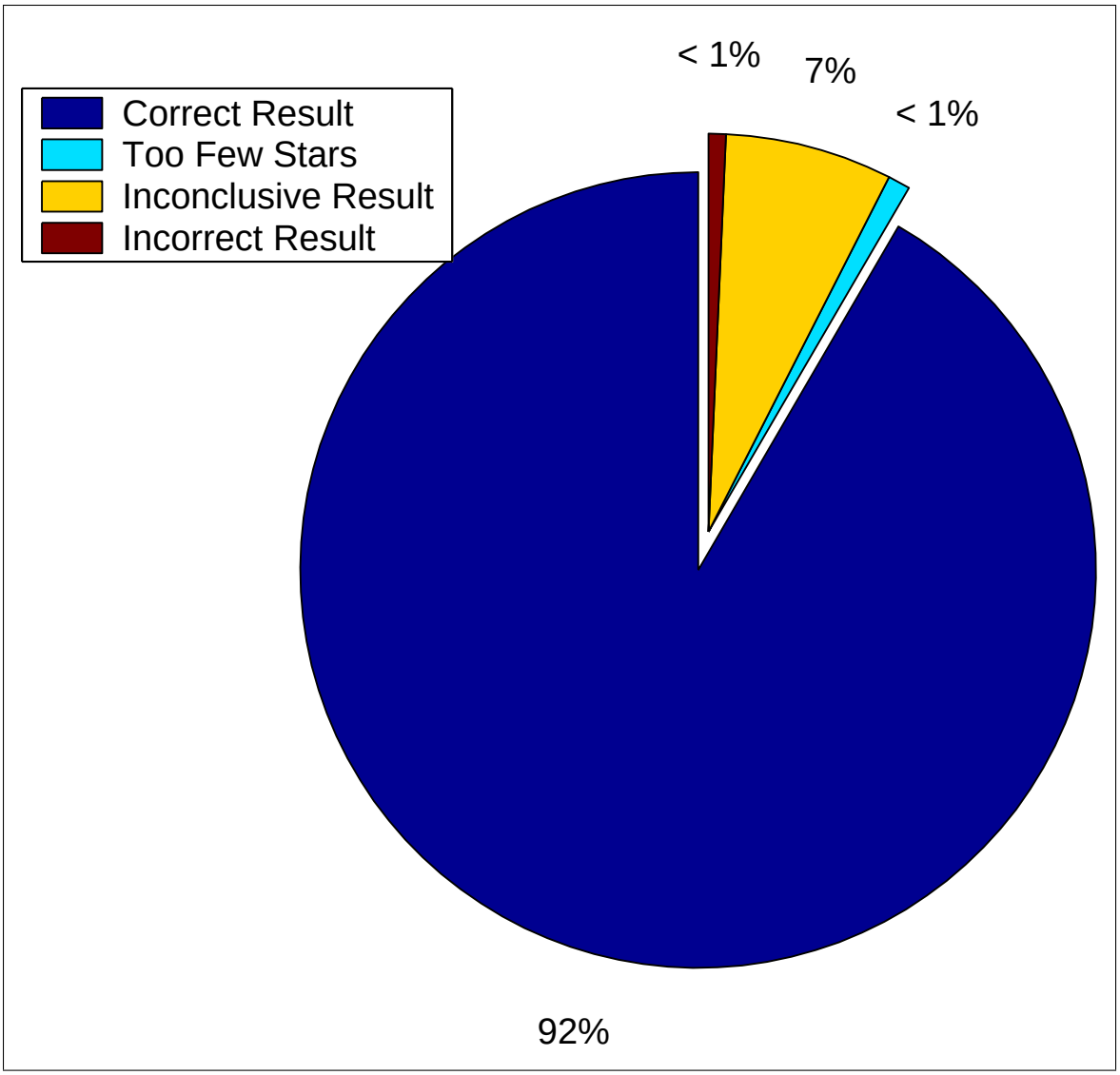

Figure 6.5.: Overall Results for Spherical Triangle Method without Pivot Limit

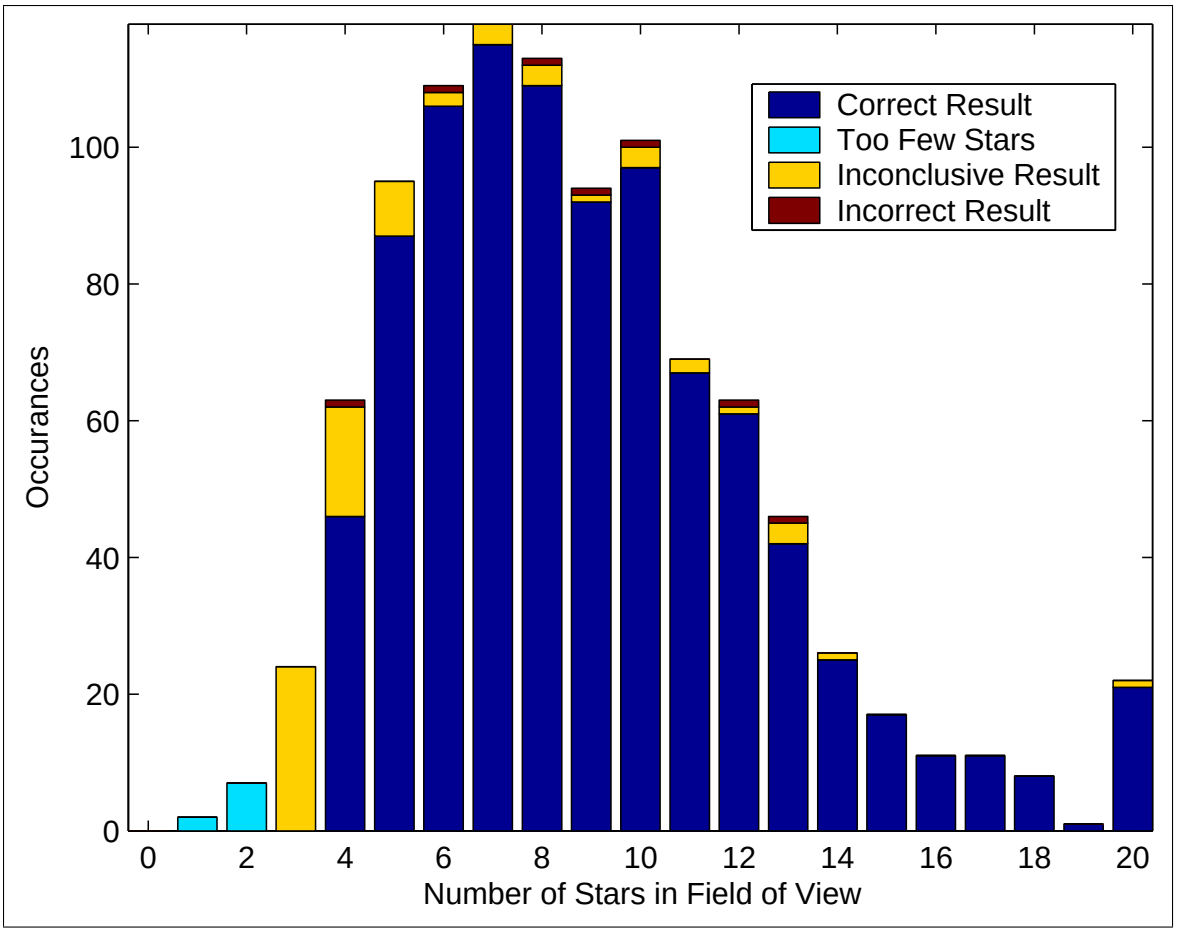

Figure 6.6.: Distribution of Results for Spherical Triangle Method without Pivot Limit

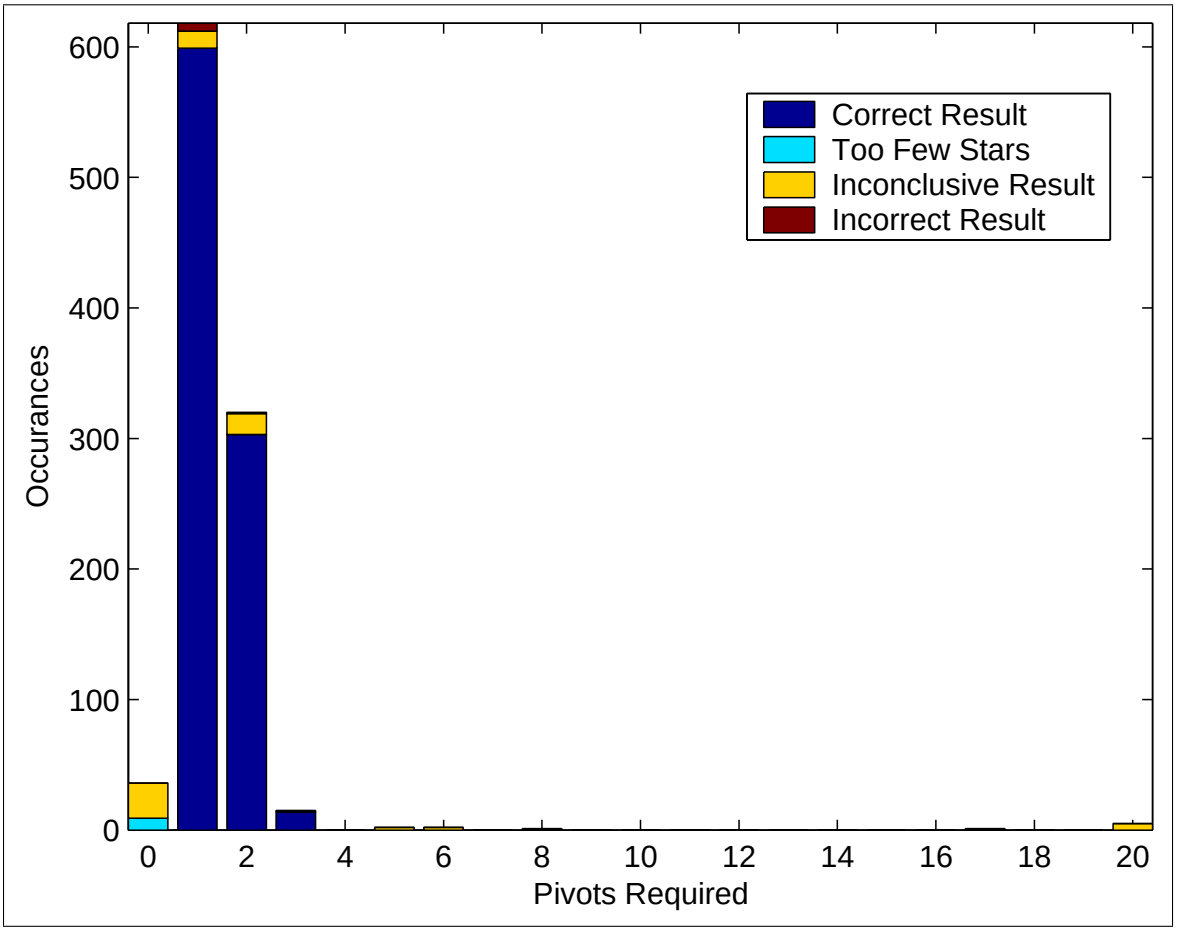

Figure 6.7.: Pivots Required for Spherical Triangle Method without Pivot Limit

6.4.2. Pivots Required

The number of pivots required by the spherical triangle method is shown in Figure 6.7. The mean number of pivots required is 1.73, and more than 2 pivots is rarely required. Since it was only a tiny number of attitudes required more than 4 pivots, the time efficiency of the method might be improved by limiting the number of pivots required to 3. The downside to this is that some of the attitudes requiring a great number of pivots are successfully identified, and as a result, the overall success of them method may drop slightly. The result

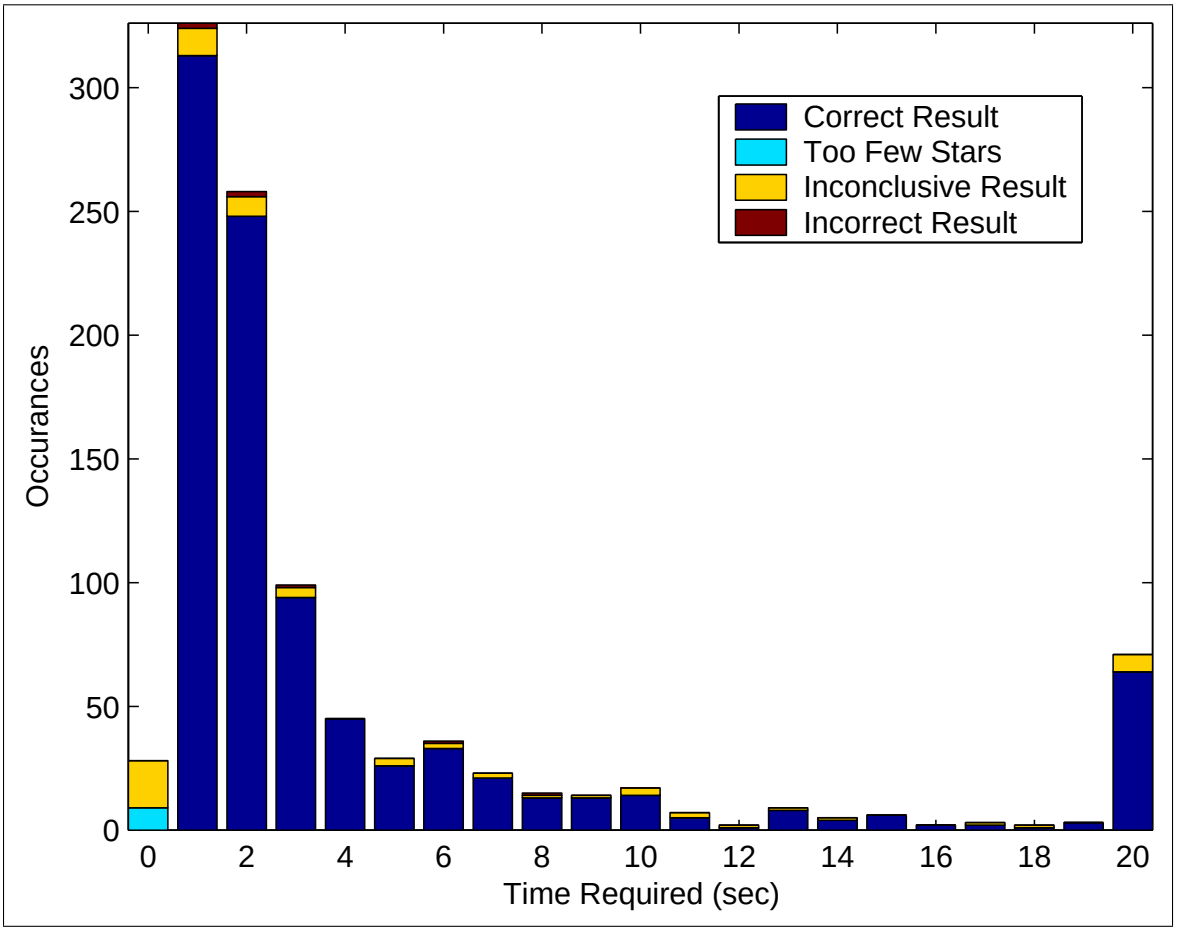

Figure 6.8.: CPU Time Required for Spherical Triangle Method without Pivot Limit

of limiting the number of pivots is discussed in the next section.

6.4.3. CPU Time Required

The CPU time required for the spherical triangle method is shown in Figure 6.8. It should be noted that solutions that take 20 seconds or more are grouped together into the 20 second slot. Some of the attitudes take as long a 400 seconds, resulting when more than 20 stars are within the field of view. Under these conditions, a thousand or more spherical

triangles exist in the field of view and have to be sorted for pivoting.

The average time required is 10.43 seconds. It is interesting to note that the median is 1.97 seconds and can be seen as the peak in the chart. For half of the attitudes tested, the result (correct, inconclusive, or otherwise) is given in 1.97 seconds or less, however, the other half took substantially longer, moving the average significantly to the right of the median. A surprising result is that a large percentage of attitudes that took 20 seconds or longer still resulted in a correct result; abandoning attitudes that are "taking too long" reduce the overall rate of success, and would not be a good method to improve the CPU rate.

An alternative suggested in the previous section that limiting the number of pivots to 3 may improve the time results. This has been tested, but the resulting graphs look very much like the results without pivot limits and are not included here. These graphs can be found in Appendix D. The overall success rate remained 92%, even though it has been that shown pivoting more than 3 times could result in a successful attitude determination. The average time drops dramatically to 1.65 seconds, an 84% improvement. The median moves only slightly to 1.33 seconds.

6.4.4. Angle Method and Spherical Triangle Method Comparison

A summary of the results between the angle method and spherical triangle method is shown in Table 6.2. The angle method is limited to nine pivots, and the spherical triangle method is limited to 3 pivots.

Although the angle method can sometimes determine attitude with as few as 3 stars in the field of view, when the star tracker is sensitive to magnitude 6 stars, it is a situation rarely encountered. Four stars are required for the spherical triangle method to begin working, but even that situation is rarely encountered.

Table 6.2.: Angle Method Vs. Spherical Triangle Method

Item	Angle	Sph. Triangle
Minimum Stars Required	2	3
Min. Stars Req'd for Success	3	4
Overall Success Rate	55%	92%
Too Few Stars To Operate	$< 1\%$	$< 1\%$
Inconclusive Result	45%	6%
Incorrect Result	$< 1\%$	$< 1\%$
Average Pivots Required	4.86	1.29
Average CPU Time Required	6.89 sec	1.65 sec

6.5. Conclusion

Overall, the spherical triangle method is far more successful than the angle method at recognizing stars, even though it will only work when there are more than three stars in the field of view. The angle method only requires 3 stars before it stars to work, but even then, the angle method could only determine the attitude a small percentage of the time. For a star tracker sensitive to magnitude 6.0 stars and has an 8 degree field of view, attitudes with less than four stars in the field of view rarely happen, and so is not a significant issue.

Both methods have similar amounts of incorrect results, due to the 3σ bound on the angle and spherical triangle searches. With such bounds, it is expected that 0.3% of the attitudes will be determined incorrectly, since, in those situations, the correct solutions lies outside the bounds of the search.

The spherical triangle method is computationally intensive, but can find a result with only a few pivots a very large percentage of the time. Determining the order for pivoting requires a large portion of the CPU time, particularly for the spherical triangle method because all the combinations of three stars have to examined as opposed to all the combinations of two stars as required by the angle method. Since the number of pivots for spherical triangle

method could be limited to three without significantly affecting the results, the average CPU time required dramatically dropped. The angle method could only be limited to 9 pivots before impacting the results, and so could not benefit as much.

The biggest problem with the spherical triangle method is certainly the size of the file required. The angle method required only 12 MB, the spherical triangle method required 167 MB.

7. Conclusions & Recommendations

Being able to match the stars in the celestial sphere to those seen the star tracker's field of view using more than one property can result in a very fast recognition algorithm with a high rate of success. For a star tracker with an 8 deg field of view, sensitive to magnitude 6.0 stars, the spherical triangle method is almost twice as likely to be successful in determining attitude than the angle method. With appropriate pivot limits on placed on both methods to minimize the response times without significantly sacrificing the rate of successful results, the spherical triangle method was 4 times faster to arrive at a solution.

The price of speed was certainly storage. Requiring 167 MB, the spherical triangle catalog, including the K-Vector array, is more than ten times larger that required for the angle method. Methods for data storage aboard spacecraft are constantly improving, and perhaps in the not-too-distant future, this size of database will not be a significant issue.

The biggest question that needs to be answered is how changing the *sigma* bounds on both methods will affect the overall results. Since inconclusive results are far preferable to incorrect solutions, a 3σ bound was chosen, meaning neither method should have incorrect results more than 0.3% of the time. This does appear to be the case, as incorrect results occur less than 1% of the time over 1,000 random attitudes. It is not yet known, however, how either method's overall rate of success will be affected significantly changing the σ bounds.

The time spent to develop efficient algorithms to create the angle and spherical triangle

catalogs will make testing the methods on different size fields of view and different magnitude sensitivities relatively easy. Outside of confirming the results of Mortari, whose model was a 32 degree field of view star tracker sensitive to magnitude 3.0 stars, little else has been examined. The biggest problem with creating the spherical triangle catalog is the polar moment calculation. An analytical solution would certainly reduce the time it takes to add the property to the spherical triangle catalog.

Another problem related to the polar moment is the graphical method employed to determine the standard deviation of a measurement. Depending on the shape of the spherical triangle, two spherical triangles with the same polar moment may have very different standard deviations. The results from the current method for determining the standard deviation are likely to be higher than the actual standard deviation for the shape spherical triangle being measured. The result is the bounds for the measured polar moment is greater than what they should be, and it is possible that triangles that should not be "finalists" are being included.

The spherical quad-tree can certainly be improved. The deeper to which the quad-tree can be constructed, the more efficient the search algorithm becomes, since more objects not located near the target object can be excluded. Since the distance of searching is a function of the size of the target-level node, if the nodes are irregularly shaped, the search distance has to limited to the size of the smallest node. To make the spherical quad-tree more regular, the basic shape might be a dodecahedron, with a pentagon as the shape to divide. Instead of a quad-tree, it would become a pent-tree. Still the quad-tree presented here worked well, taking only an hour to catalog the angles, and several hours for the spherical triangles.

The methods presented here tackle the worst-case scenario for a star tracker – no a priori information about its position, no other sensor information to assist and that the attitude in the field of view can be of any section of the celestial sphere. If some information hinting

at a direction the star tracker should focus on could be found, the spherical quad-tree may be a useful tool for limiting the focus of the star tracker to a particular section of the sky. It seems wasteful that the spherical quad-tree, for this thesis, is only used for creating the angle and spherical triangle catalog, and for determining the stars within the field of view of the star tracker for testing the angle and spherical triangle methods.

Bibliography

[1] D. M. Gottlieb, "Star identification techniques," *Spacecraft Attitude Determination and Control*, pp. 259–266, 1978.

[2] E. A. Ketchum and R. H. Tolson, "Onboard star identification without a priori attitude information," *Journal of Guidance, Control and Dynamics*, vol. 18, pp. 242–246, March-April 1995.

[3] J. C. Kosik, "Star pattern identification aboard an inertially stabilized spacecraft," *Journal of Guidance, Control and Dynamics*, vol. 14, pp. 230–235, 1991.

[4] P. Gambardella, "Algorithms for autonomous star identification," Tech. Rep. TM-84789, NASA, 1980.

[5] J. L. Junkins, C. C. White, and J. D. Turner, "Star pattern recognition for real time attitude determinationstar pattern recognition for real time attitude determination," *Journal of Astronautical Sciences*, vol. 25, pp. 251–270, November 1977.

[6] J. L. Junkins and T. E. Strikweerda, "Autonomous attitude estimation via star sensing and pattern recognition," in *Proceedings of the Flight Mechanics and Estimation Theory Symposium*, (Greenbelt, MD), pp. 127–147, NASA-Goddard Space Flight Center, 1978.

[7] T. E. Strikewerda, J. L. Junkins, and J. D. Turner, "Real-time spacecraft attitude determination by star pattern recognition: Further results," *AIAA Paper 79-0254*, January 1979.

[8] B. V. Sheela, C. Shekhar, P. Padmanabhan, and M. G. Chandrasekhar, "New star identification technique for attitude control," *Journal of Guidance, Control and Dynamics*, vol. 14, no. 2, pp. 477–480, 1991.

[9] K. E. Williams, T. E. Strikwerda, H. L. Fisher, K. Strohbehn, and T. G. Edwards, "Design study: Parallel architectures for autonomous star pattern identification and tracking," *AAS Paper 93-102*, February 1993.

[10] Ball Aerospace Systems Group, Electro-Optics Cryogenics Division, Boulder, CO, *Specification Sheet for Ball Aerospace CT-601 Star Tracker*.

[11] J. L. Crassidis, F. Markley, A. Kyle, and K. Kull, "Attitude determination improvements for goes," in *Proceedings of the Flight Mechanics/Estimation Theory Symposium*, (Greenbelt, MD), pp. 151–165, NASA-Goddard Space Flight Center, May 1996.

[12] M. D. Schuster, "Attitude determination from vector observations," *Journal of Guidance, Control and Dynamics*, vol. 4, no. 1, pp. 70–77, 1981.

[13] J. D. Jansen, "Use of spherical trigonometry to compute near-well flow through irregular grid block boundaries," *Compuational Geosciences*, vol. 6, pp. 195–206, 2002.

[14] J. L. Crassidis and J. L. Junkins, "Optimal estimation of dynamic systems.".

[15] Skymap.com, *Skymap of Stars, Magnitude 12.0 Limit*.

[16] R. Bauer, "Distribution of points on a sphere with application to star catalogs," *American Institude of Aeronautics and Astronautics*, no. AIAA-98-4330, 1998.

[17] N. Dale and S. C. Lilly, *Pascal Plus Data Structures, Algorithms And Advanced Programming*. D.C. Heath and Company, 1985.

[18] D. Mortari, "A fast on-board autonomous attitude determination system based on a new star-id technique for a wide fov star tracker," *AAS/AIAA Space Flight Mechanics Meeting*, pp. 1–11, 1996.

[19] R. H. Battin, *An Introduction to the Mathematics and Methods of Astrodynamics*. American Institute of Aeronautics and Astronautics, 1999.

A. Angle Method Without Pivot Limits

This section will show the results of the Angle Method with the settings as described in table A.1

Table A.1.: Angle Method Without Pivot Limit Testing Specfications

Specifications
1,000 Random Attitude
Magnitude 6.0 Limit
8 deg Star Tracker Field Of View
6.94 deg Angle Limit
10,000 Possible Solution Per Angle Limit
1,000 Pivot Limit (virtually no limit)

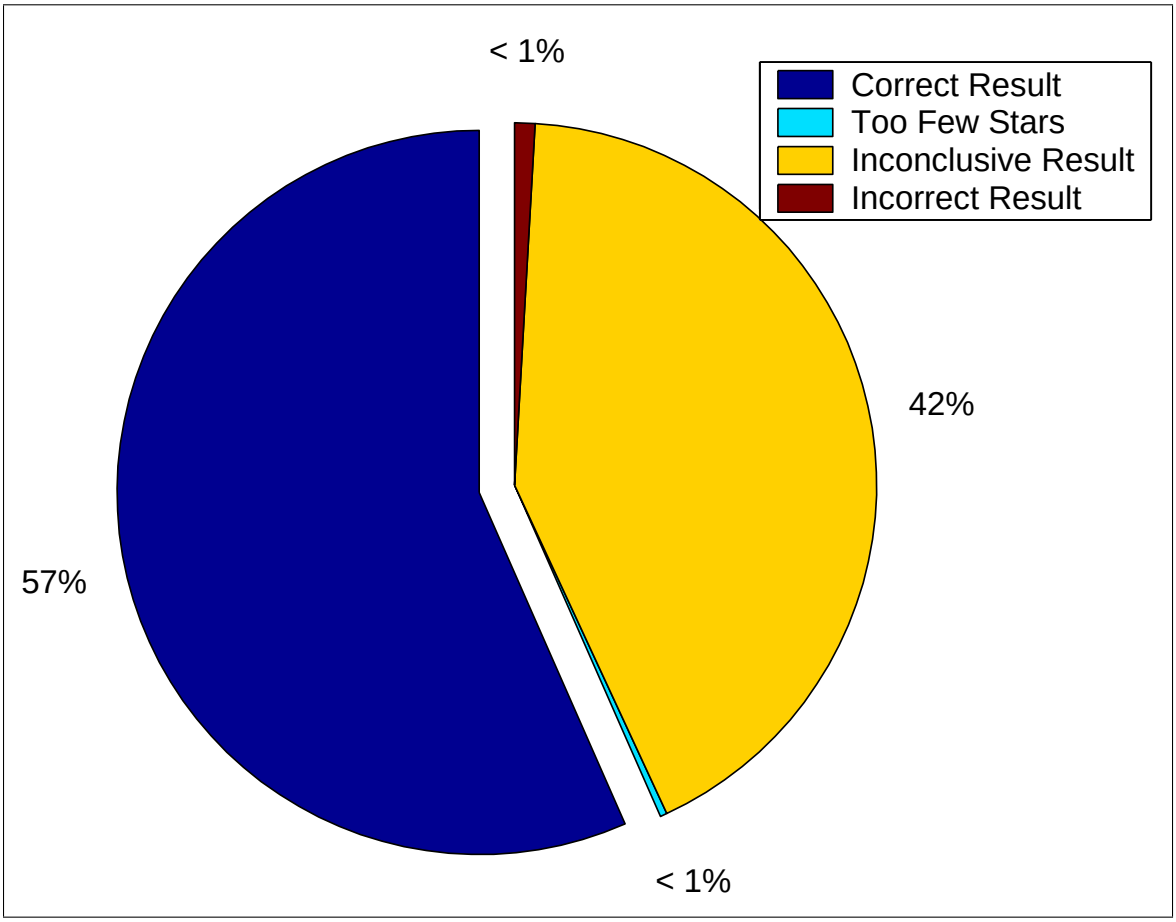

Figure A.1.: Overall Results For Angle Method With No Pivot Limit.

A.1. Overall Results

A.2. Distribution of Results

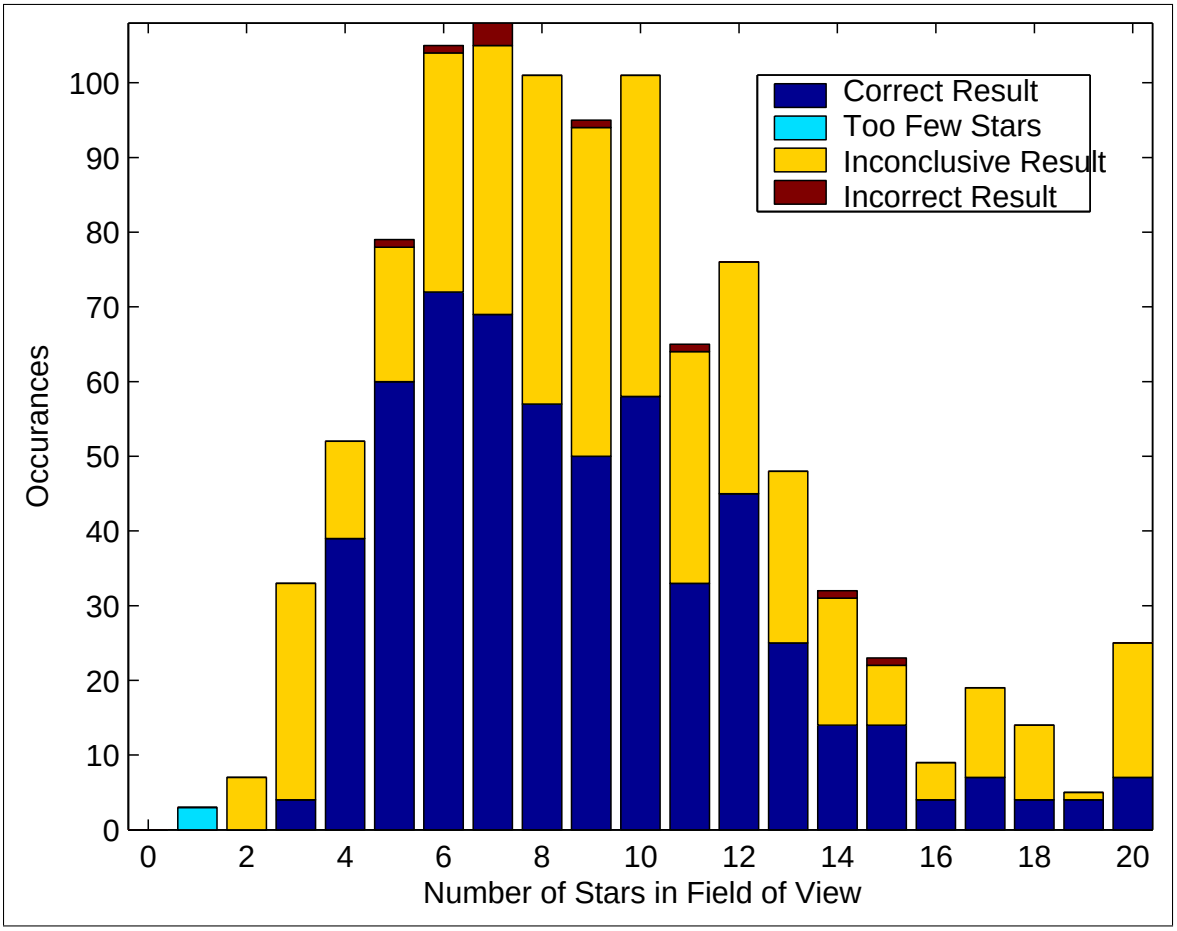

Figure A.2.: Distribution Of Results By Stars In FOV Using The Angle Method With No Pivot Limit.

A.3. Pivots Required

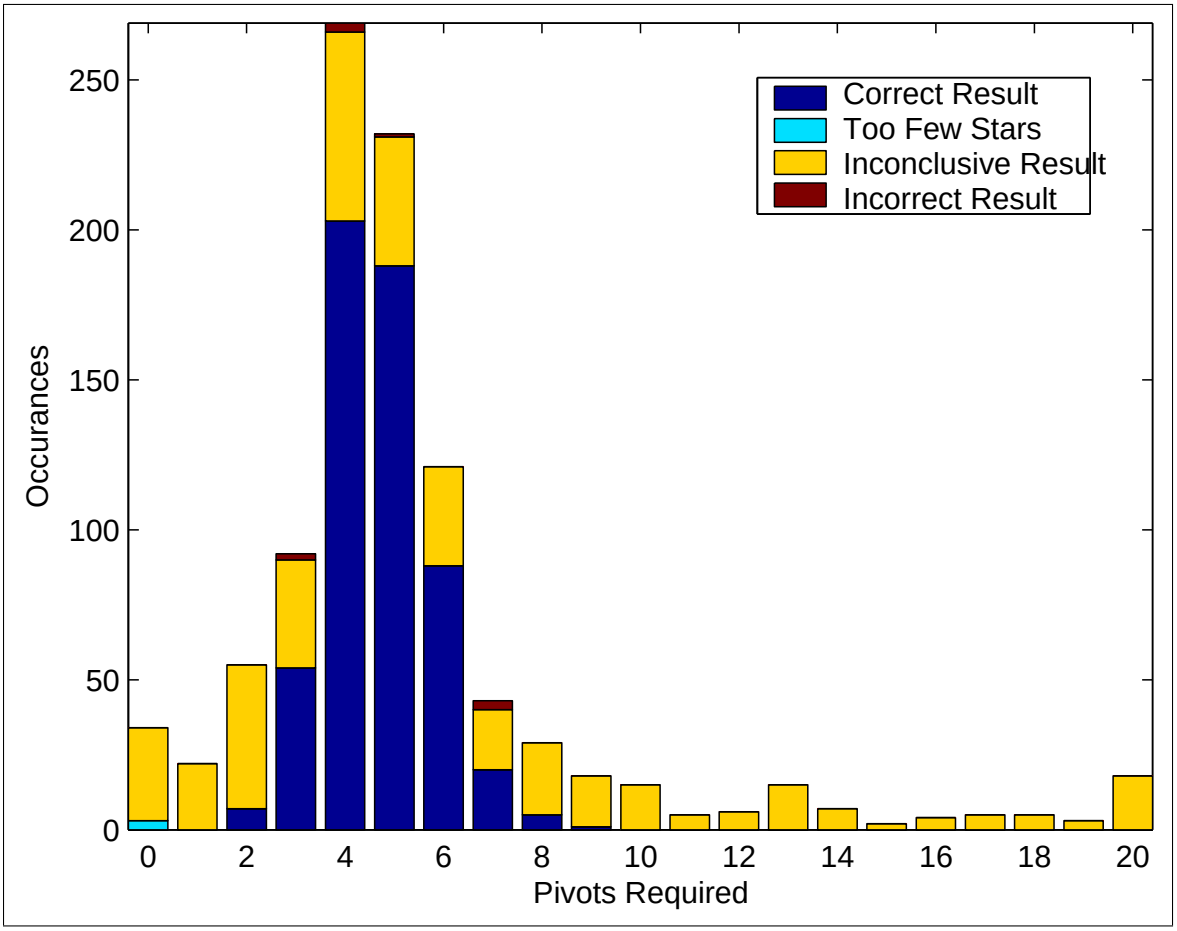

Figure A.3.: Distribution Of Pivots Required Using The Angle Method With No Pivot Limit.

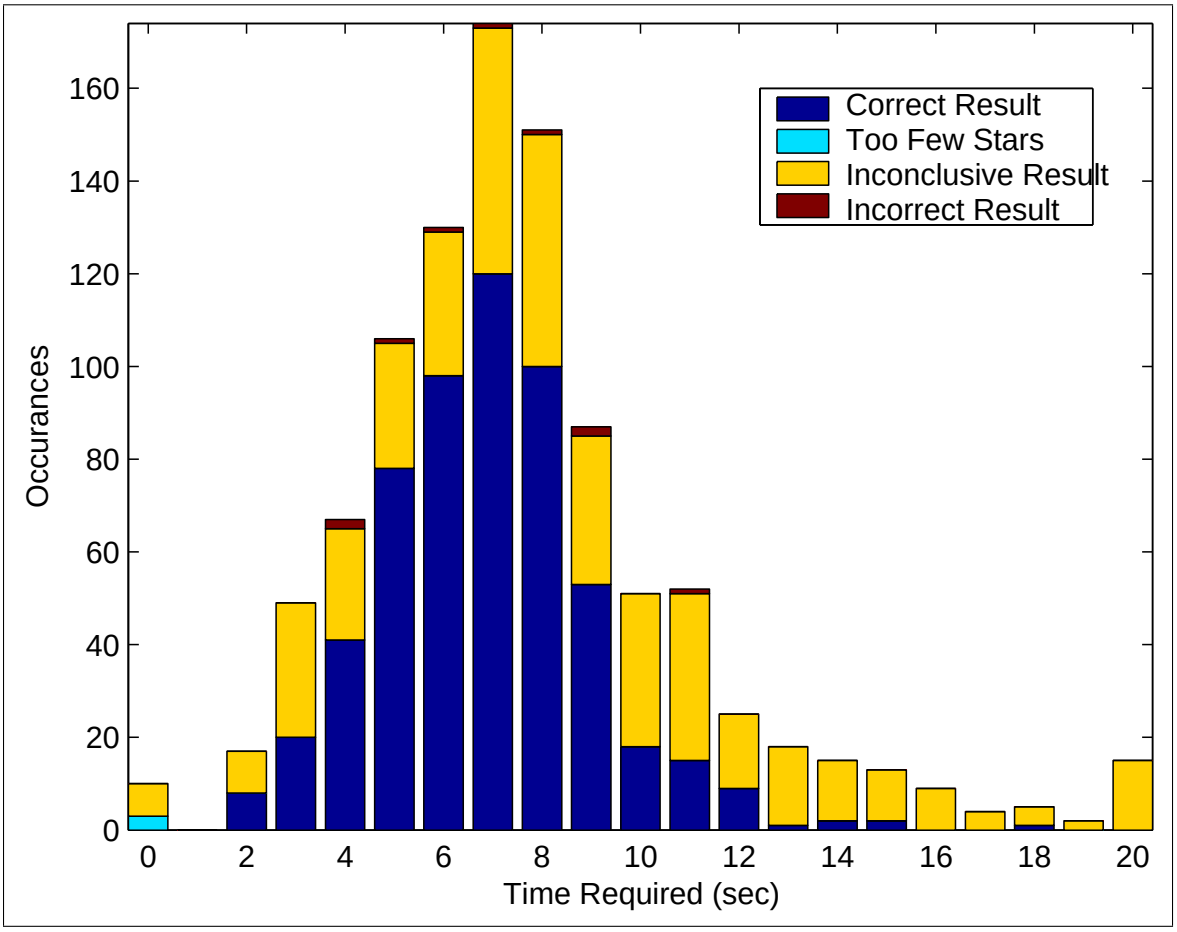

Figure A.4.: Distribution Of Time Required Using The Angle Method With No Pivot Limit.

A.4. CPU Time Required

B. Angle Method With 9-Pivot Limit

This section will show the results of the Angle Method with the settings as described in table B.1

Table B.1.: Angle Method With 9-Pivot Limit Testing Specfications

Specifications
1,000 Random Attitude
Magnitude 6.0 Limit
8 deg Star Tracker Field Of View
6.94 deg Angle Limit
10,000 Possible Solution Per Angle Limit
9 Pivot Limit

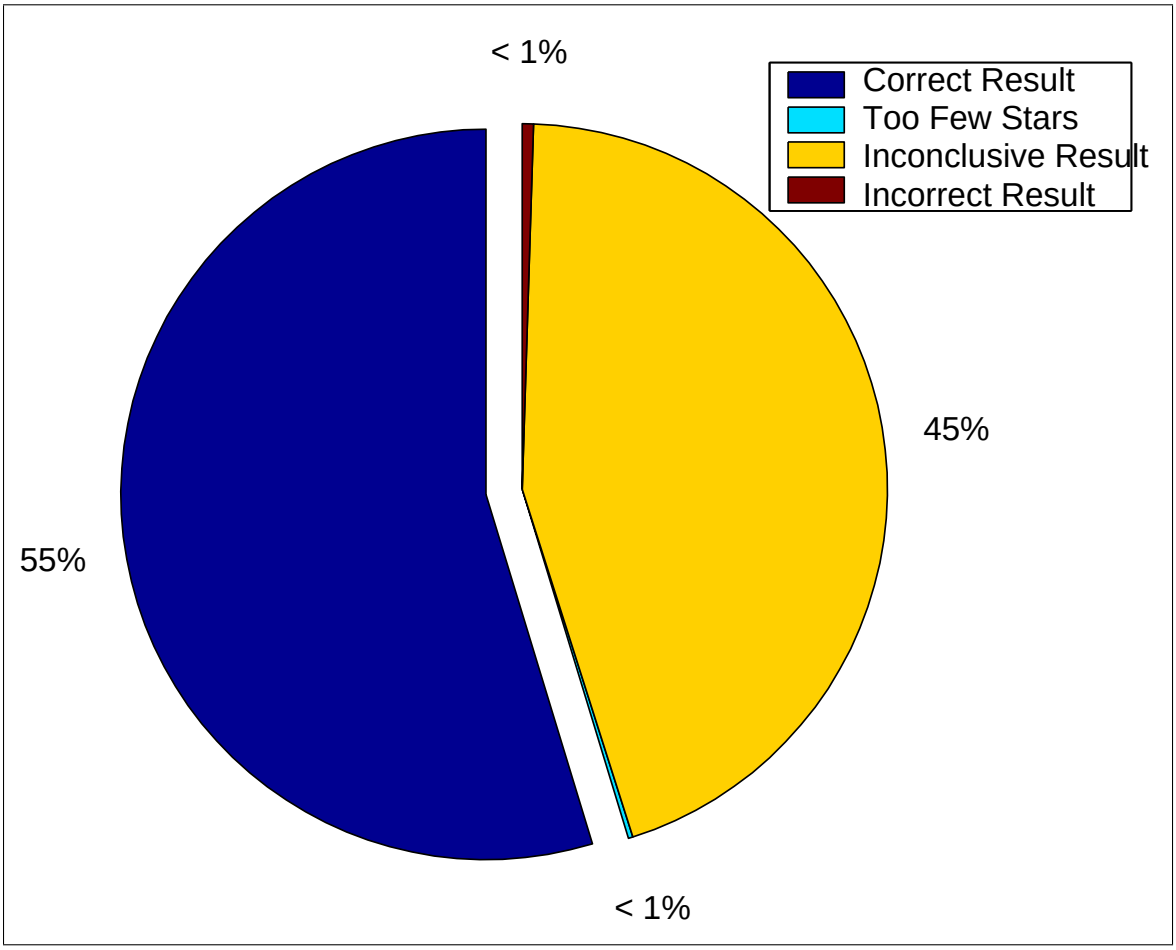

Figure B.1.: Overall Results For Angle Method With 9-Pivot Limit.

B.1. Overall Results

B.2. Distribution of Results

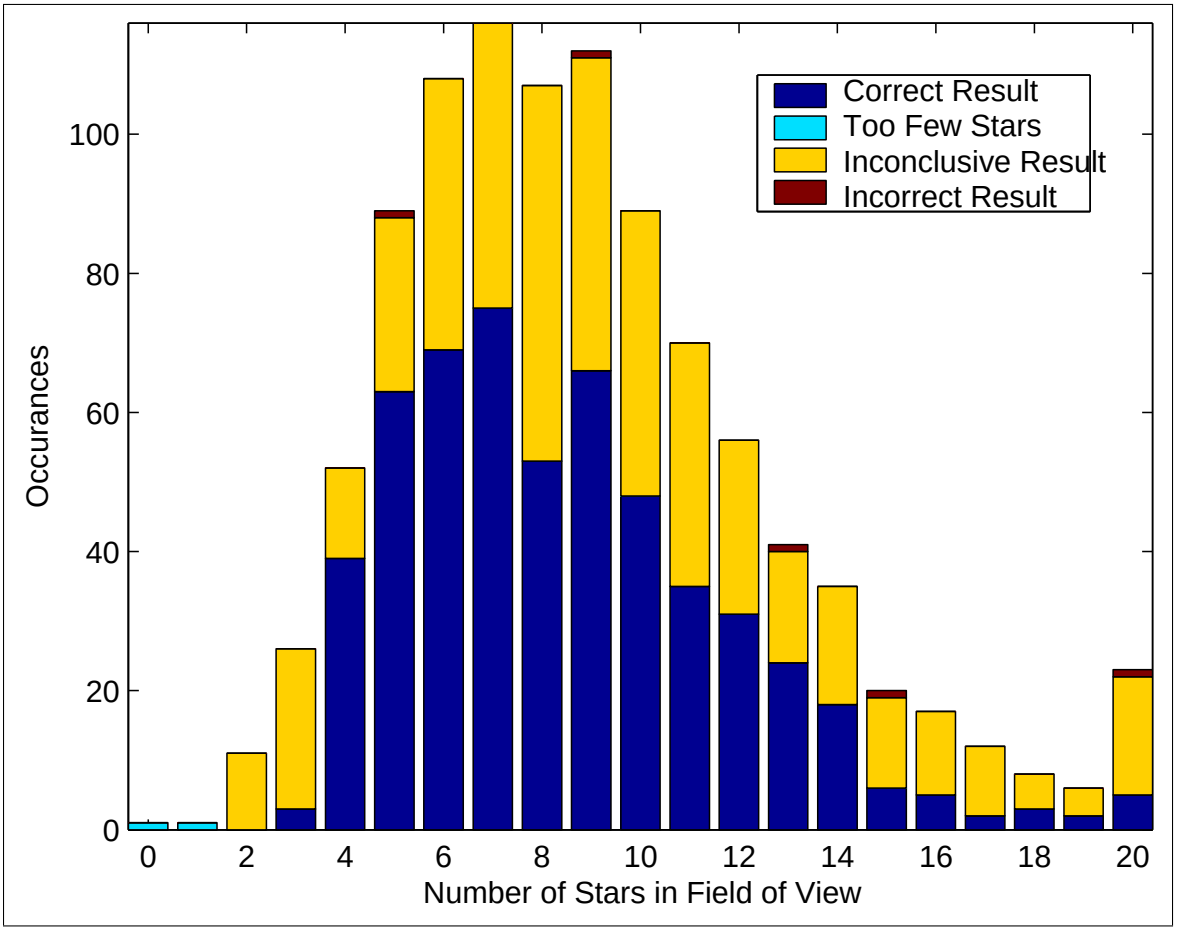

Figure B.2.: Distribution Of Results By Stars In FOV Using The Angle Method With 9-Pivot Limit

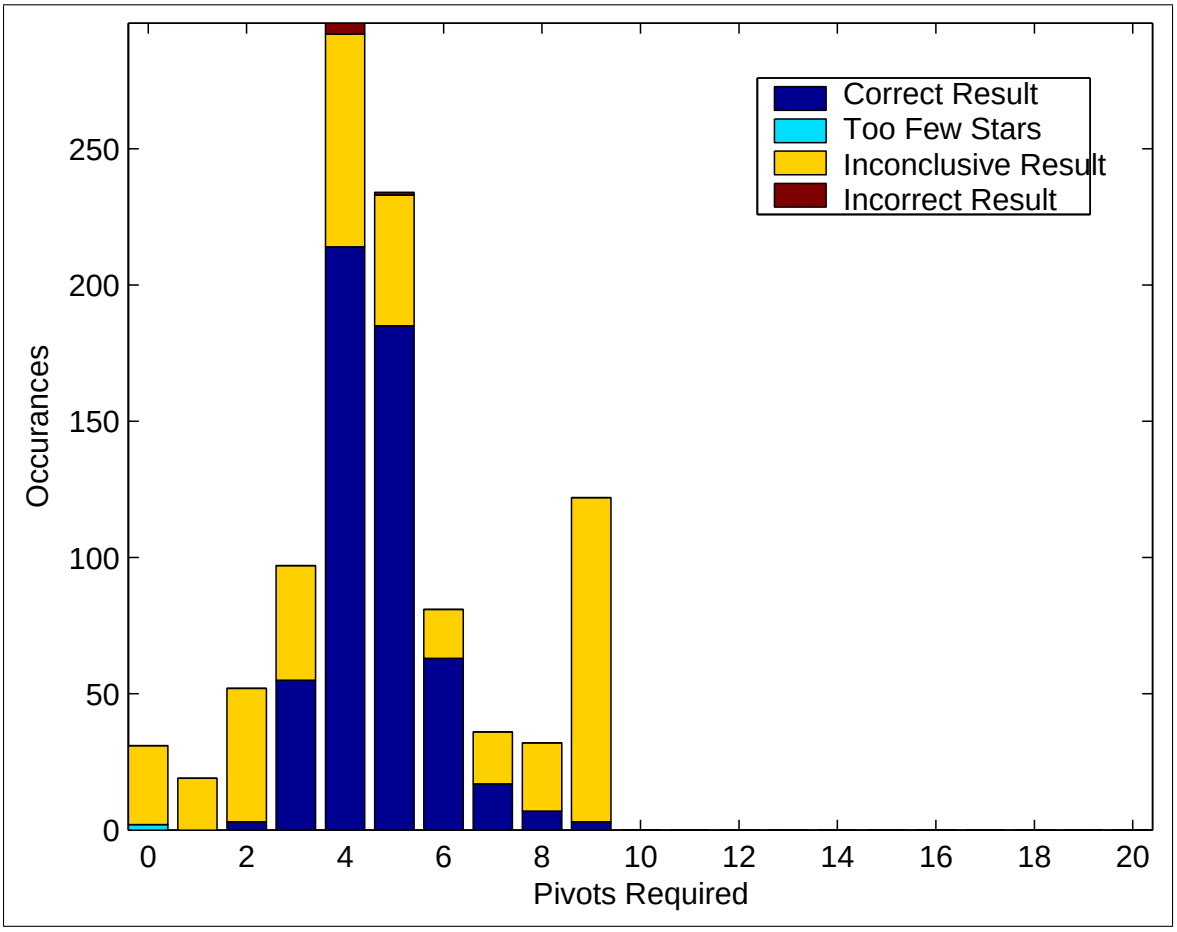

Figure B.3.: Distribution Of Pivots Required Using The Angle Method With 9-Pivot Limit.

B.3. Pivots Required

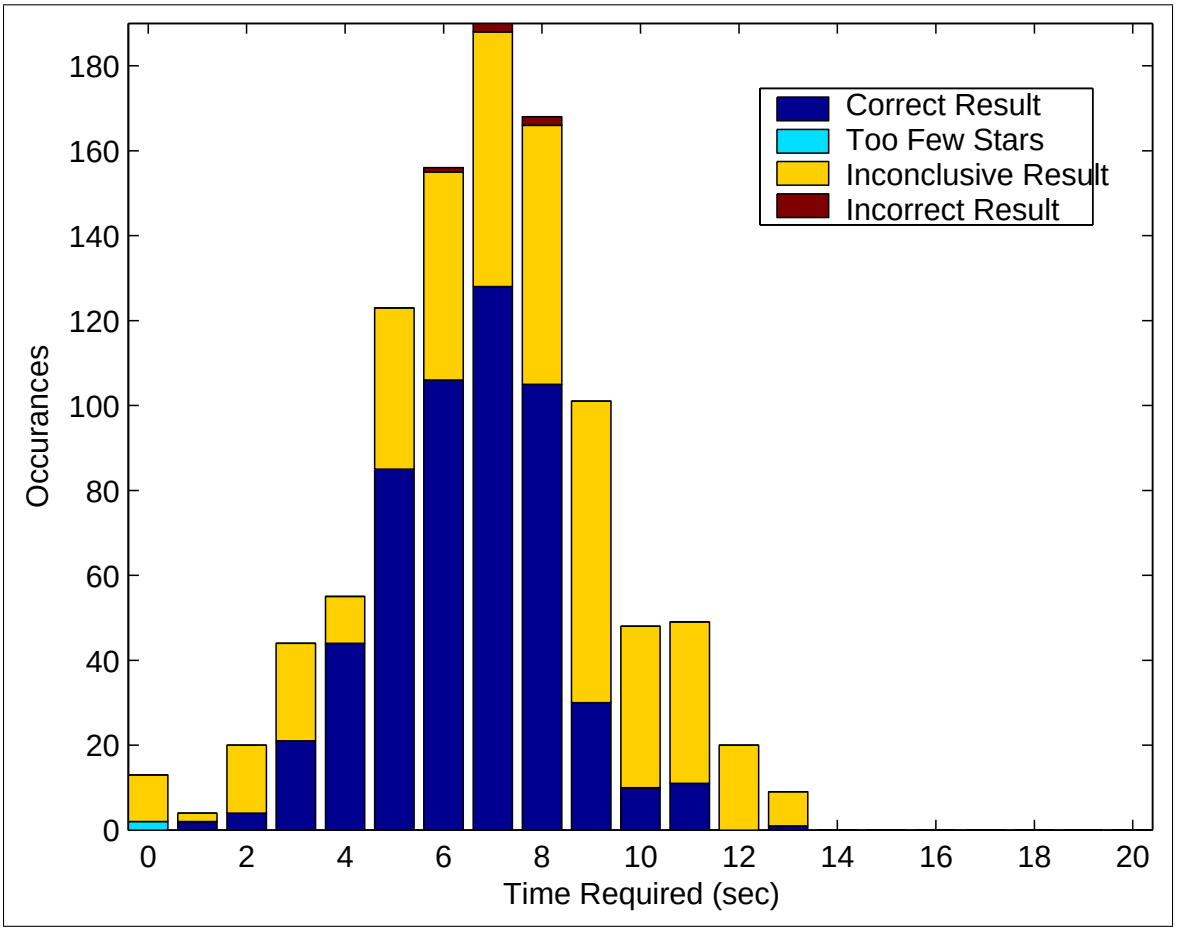

Figure B.4.: Distribution Of Time Required Using The Angle Method With 9-Pivot Limit.

B.4. CPU Time Required

C. Spherical Triangle Method Without Pivot Limit

This section will show the results of the Angle Method with the settings as described in table C.1

Table C.1.: Spherical Triangle Method Without Pivot Limit Testing Specifications

Specifications
1,000 Random Attitude
Magnitude 6.0 Limit
8 deg Star Tracker Field Of View
6.94 deg Triangle Field Of View Limit
10,000 Possible Solution Per Spherical Triangle Limit
1,000 Pivot Limit (virtually no limit)

C.1. Overall Results

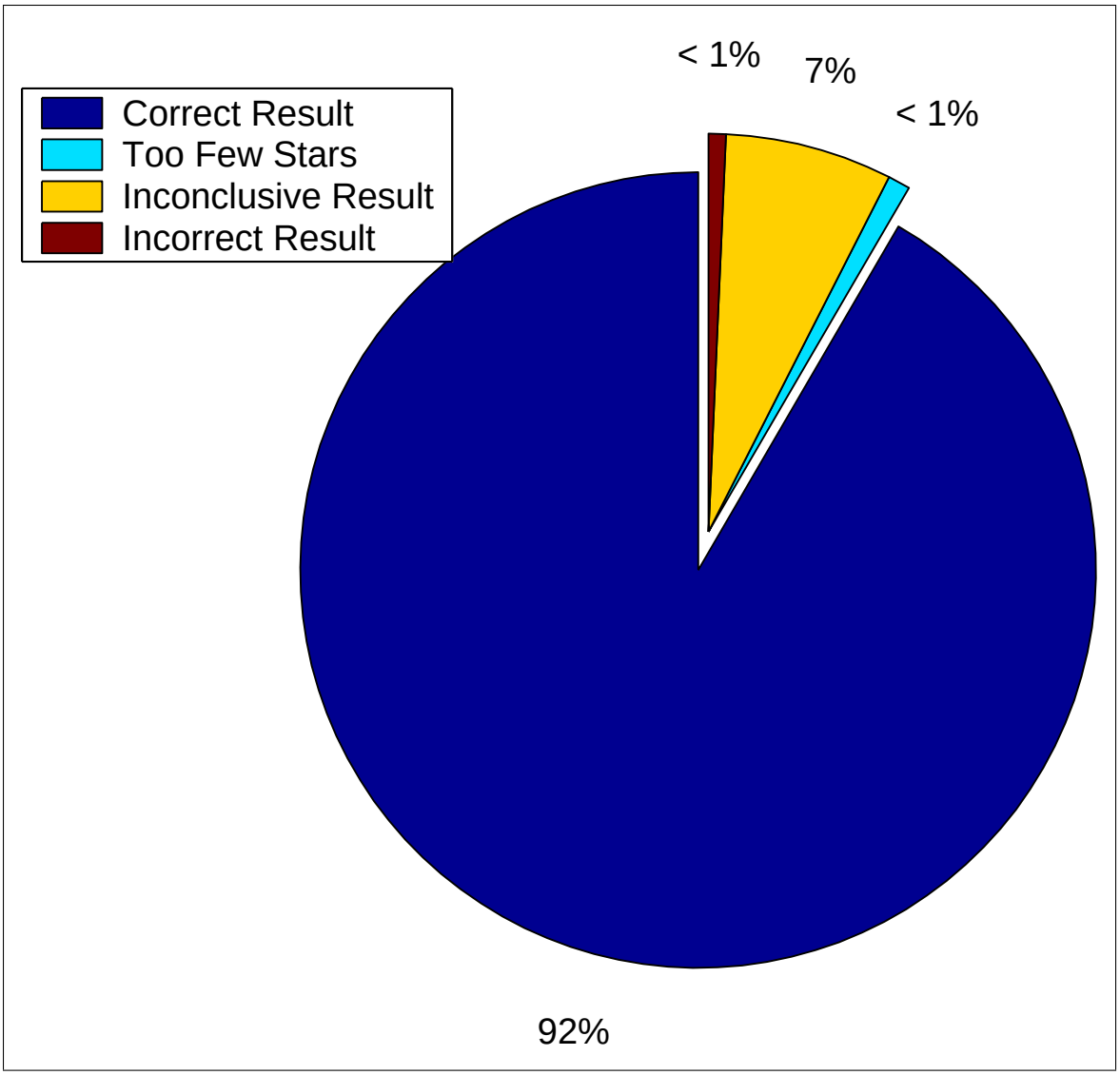

Figure C.1.: Overall results of Spherical Triangle Method Without Pivot Limits.

C.2. Distribution of Results

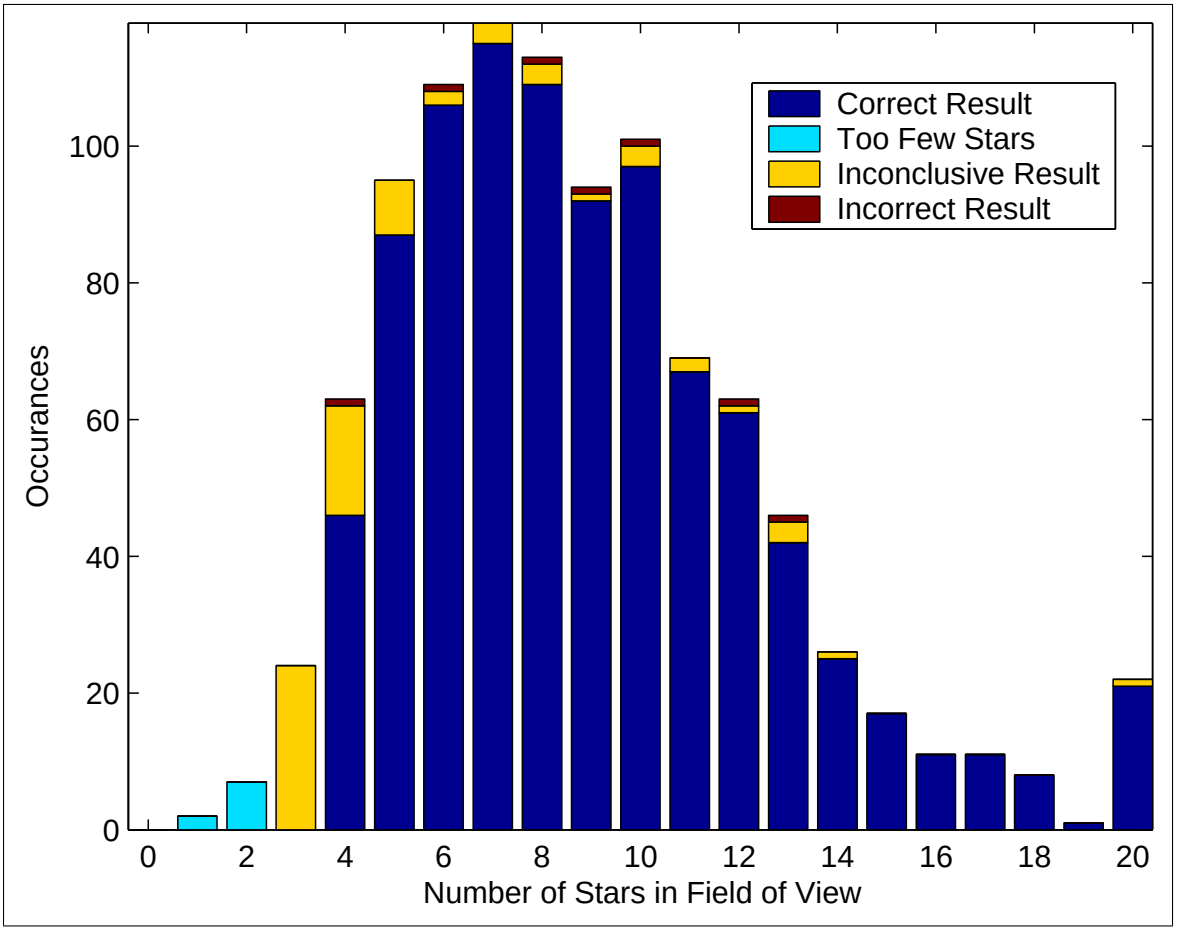

Figure C.2.: Distribution Of Results by Stars in FOV Using The Spherical Triangle Method Without Pivot Limit.

C.3. Pivots Required

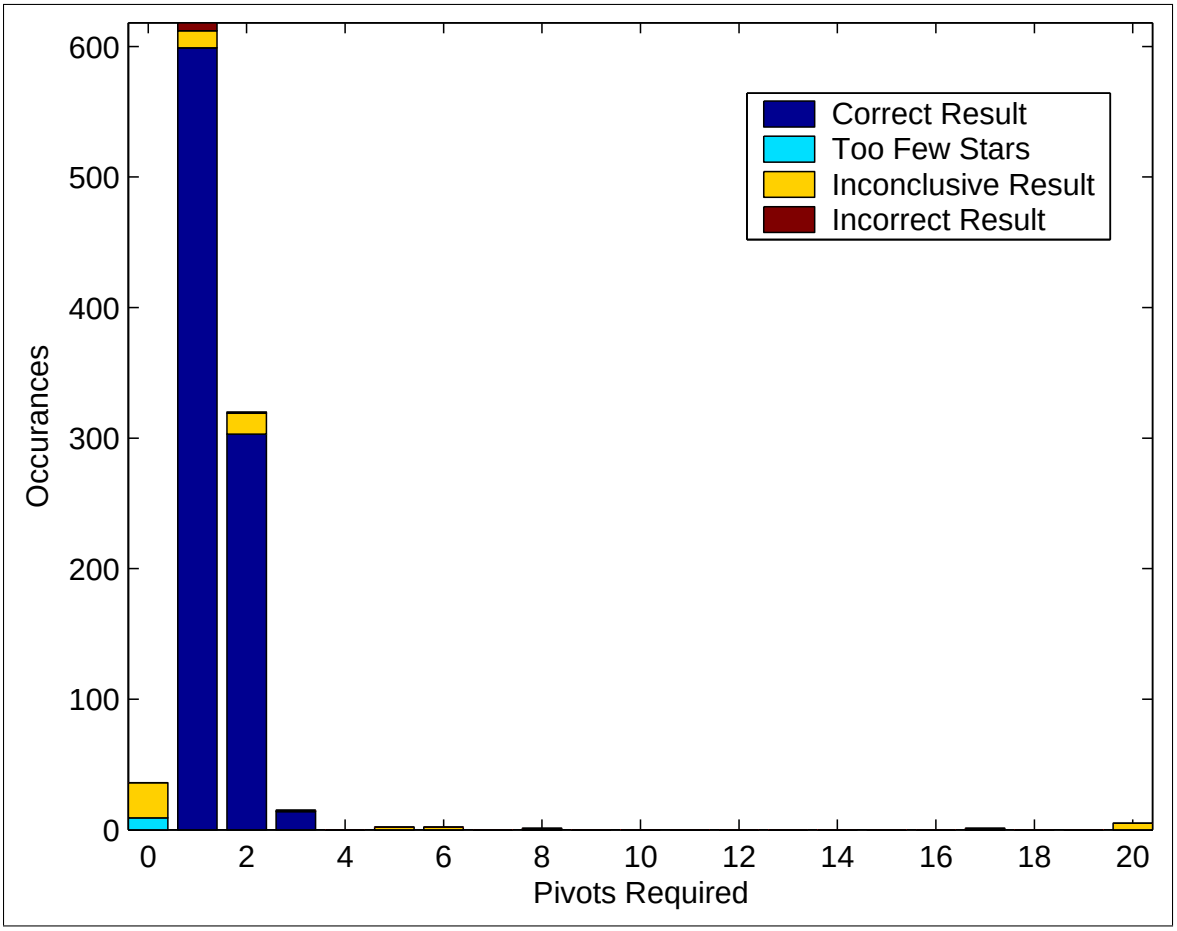

Figure C.3.: Distribution Of Pivots Required Using The Spherical Triangle Method Without Pivot Limit.

C.4. CPU Time Required

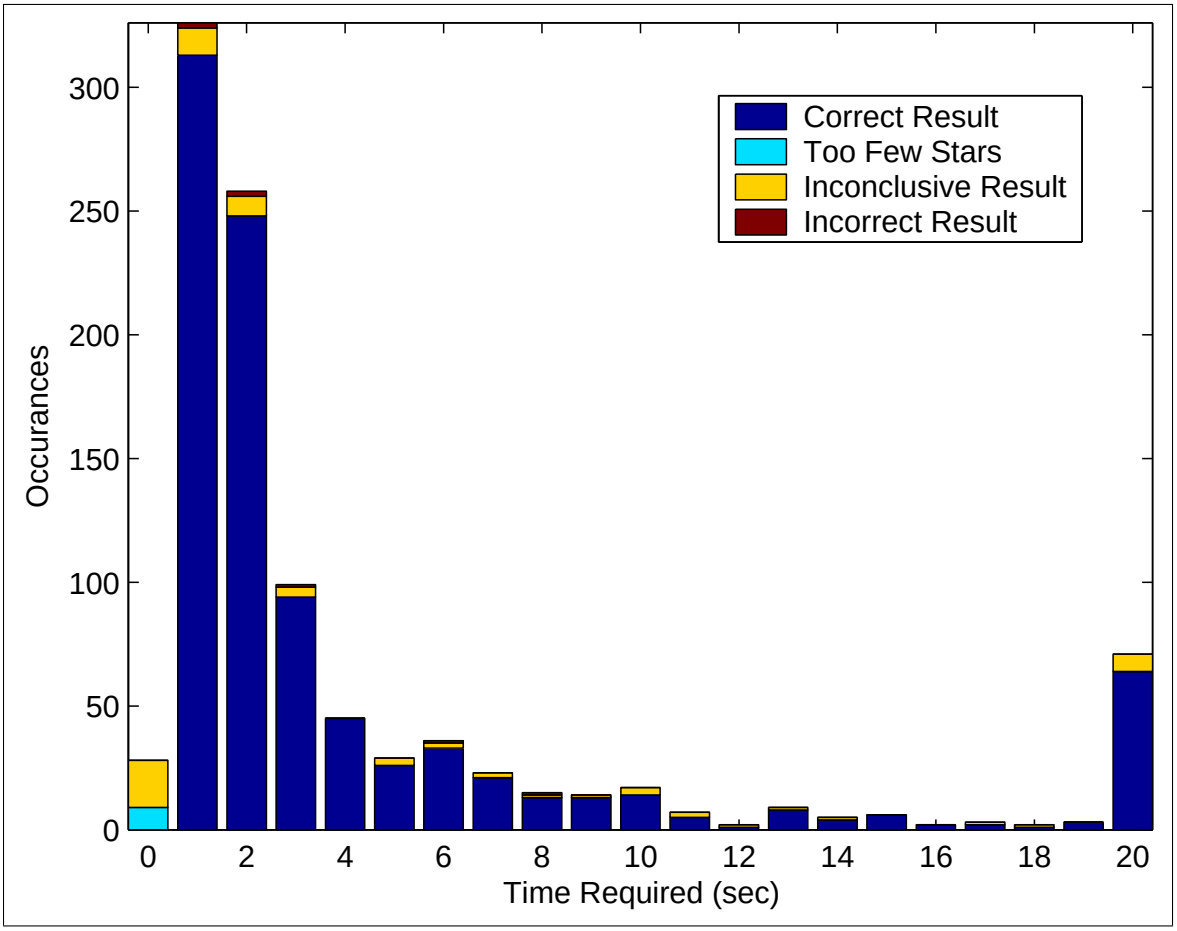

Figure C.4.: Distribution of Time Required Using The Spherical Triangle Method Without Pivot Limit.

D. Spherical Triangle Method With 3-Pivot Limit

This section will show the results of the Angle Method with the settings as described in table D.1

Table D.1.: Spherical Triangle Method With 3-Pivot Limit Testing Specifications

Specifications
1,000 Random Attitudes
Magnitude 6.0 Limit
8 deg Star Tracker Field Of View
6.94 deg Triangle Field Of View Limit
10,000 Possible Solution Per Spherical Triangle Limit
3 Pivot Limit

D.1. Overall Results

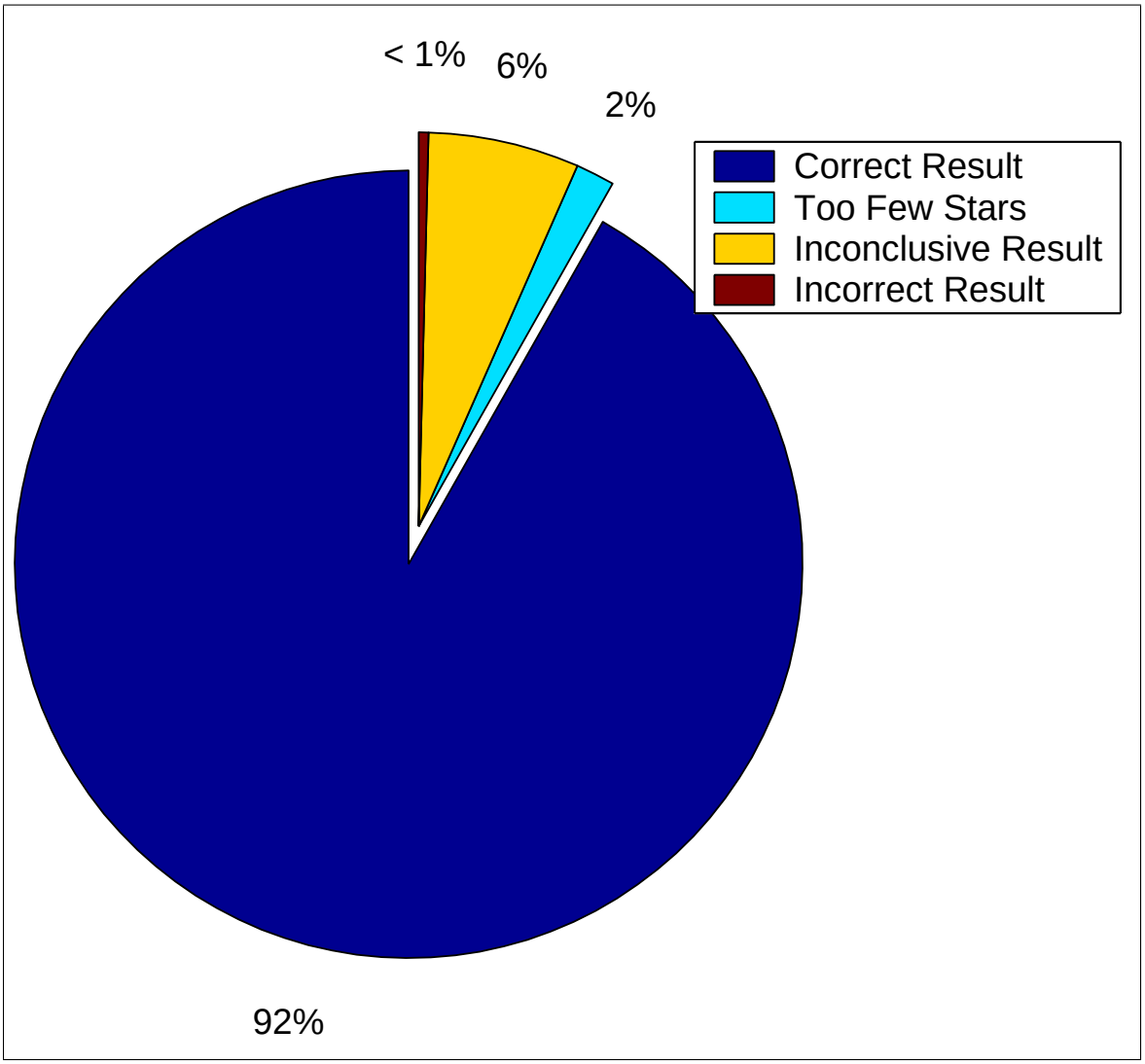

Figure D.1.: Overall Results Of Spherical Triangle Method With 3-Pivot Limit.

D.2. Distribution of Results

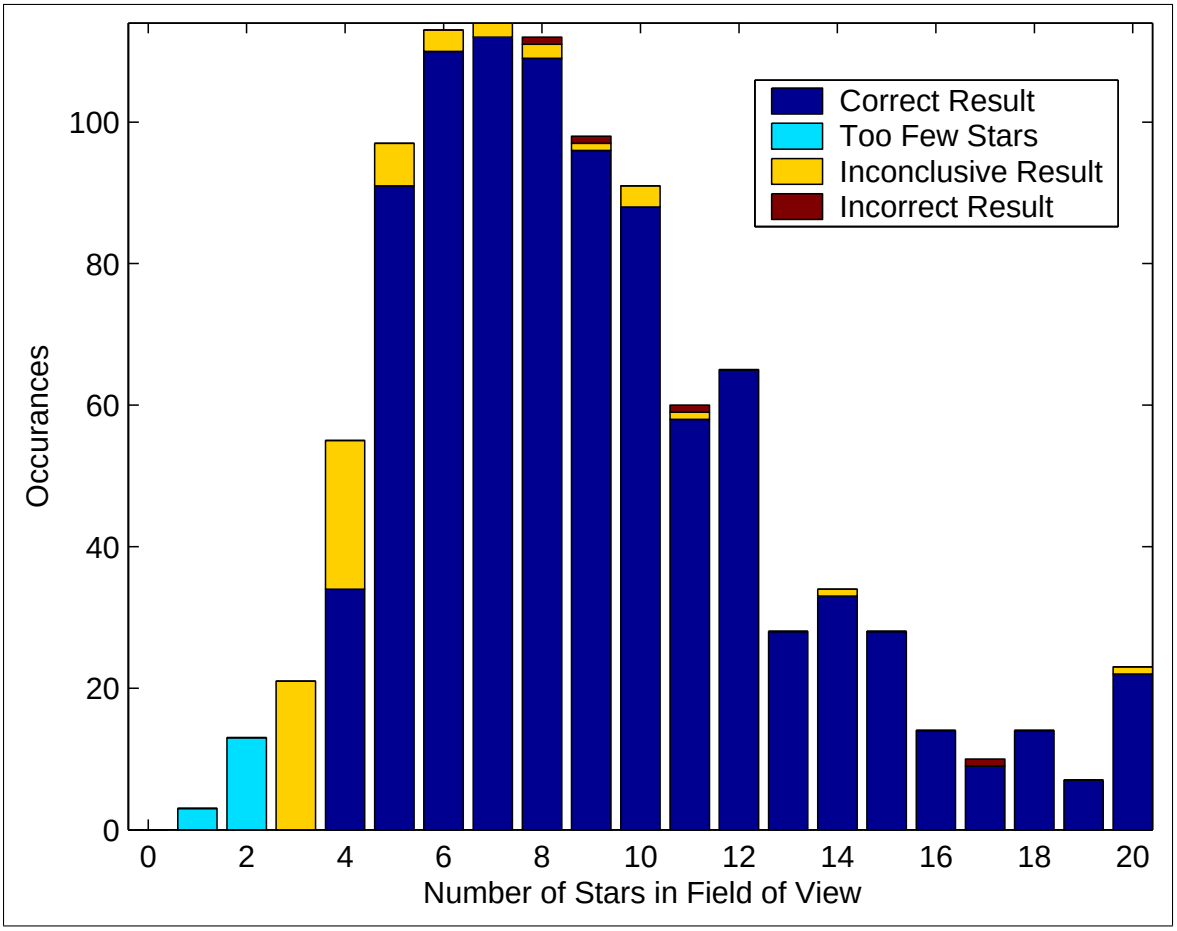

Figure D.2.: Distribution Of Results by Stars in FOV Using The Spherical Triangle Method With 3-Pivot Limit

D.3. Pivots Required

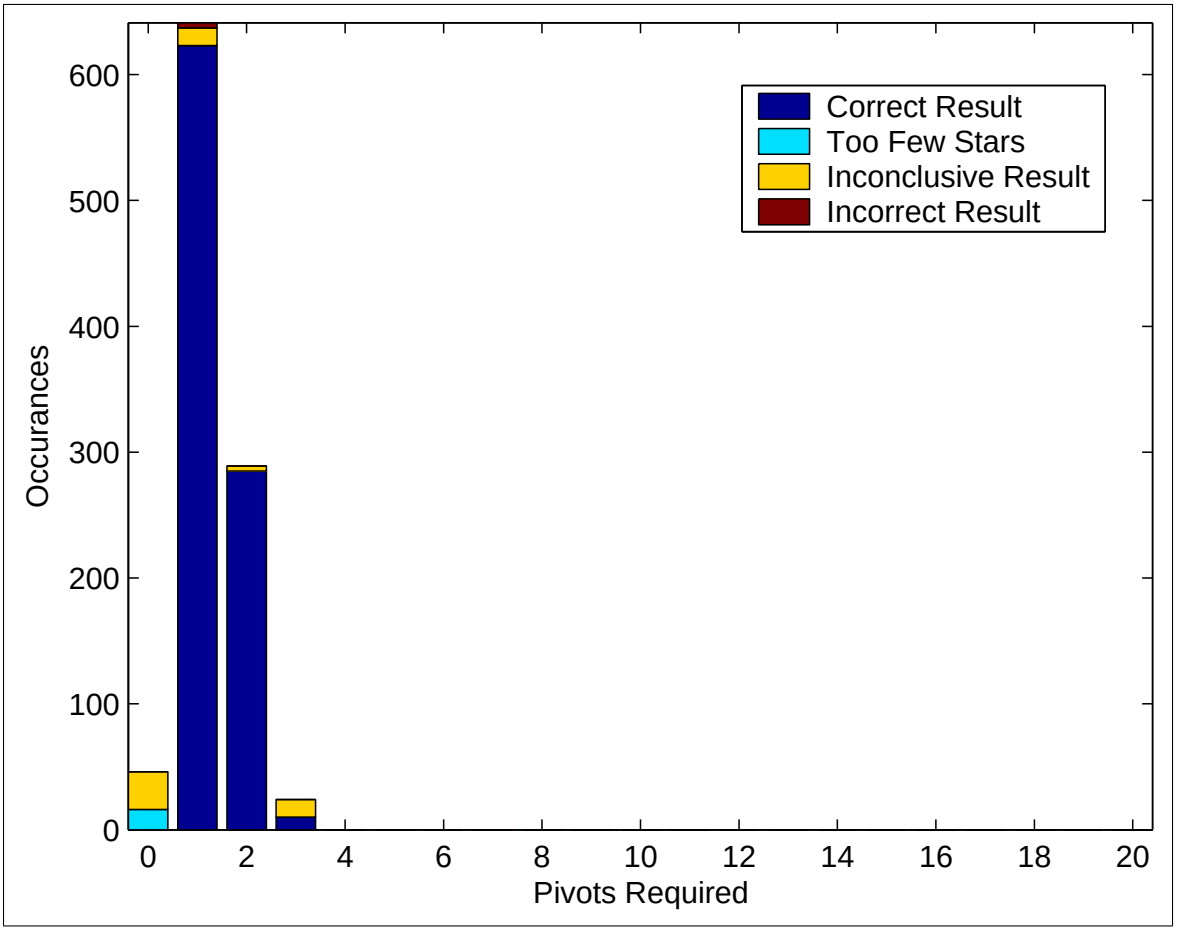

Figure D.3.: Distribution Of Pivots Required Using The Spherical Triangle Method With 3-Pivot Limit.

D.4. CPU Time Required

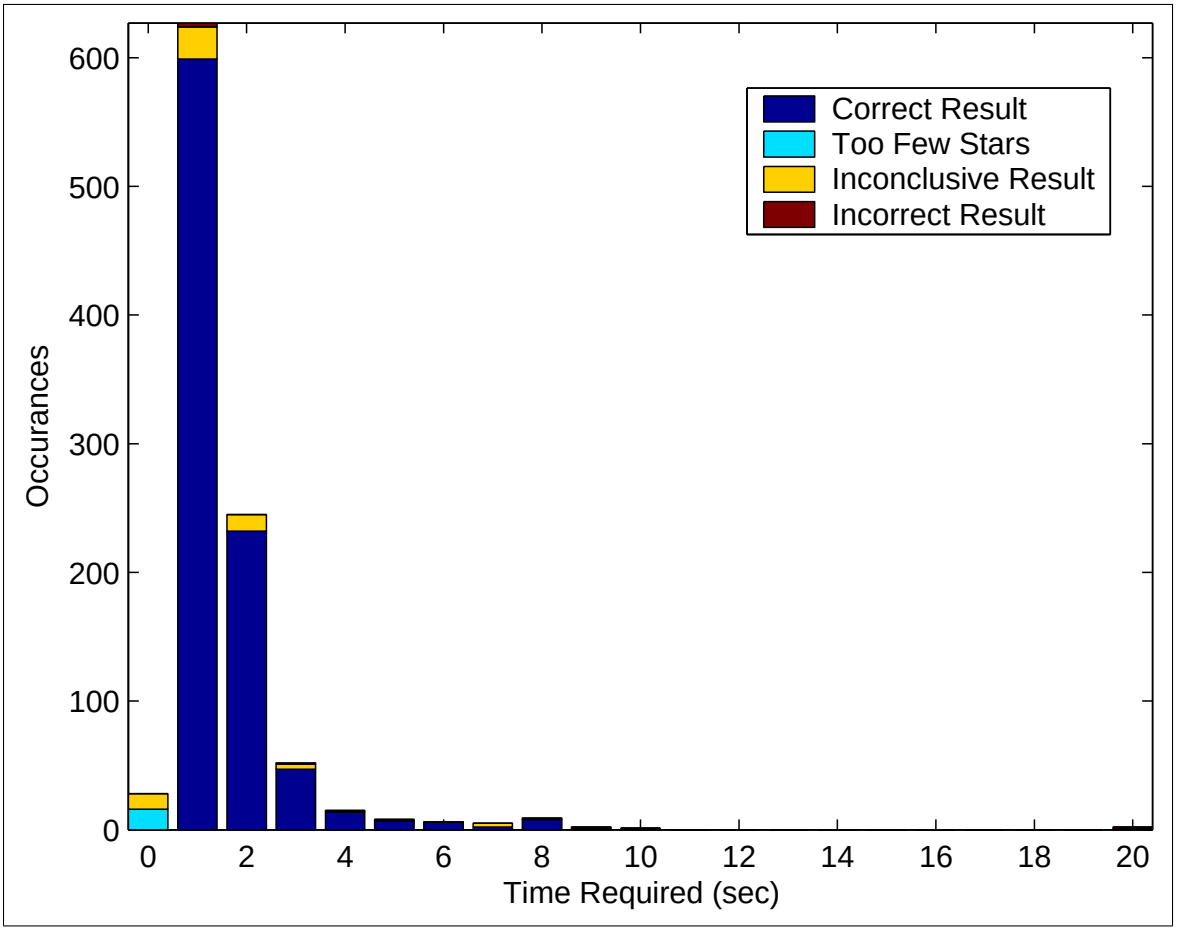

Figure D.4.: Distribution of Time Required Using The Spherical Triangle Method With 3-Pivot Limit.

E. Matlab Code

All the source code used to create the angle and spherical triangle catalog as well as the method to test the angle and spherical triangle method are listed here. Figure **??** shows the order in which the code needs to be executed.

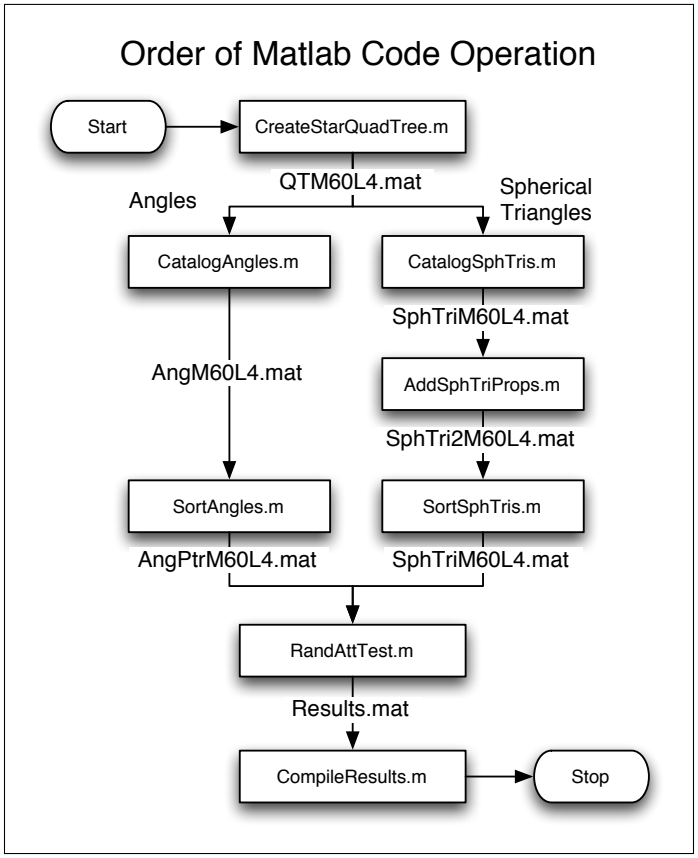

Figure E.1.: Order in Which to Operate Matlab Code

E.1. AddChildren.m

```
% ========================================================================
%
% AddChildren.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Creates four new nodes for parent node
%
% INPUTS:   n - parent node no.
%           level - quad-tree level of children
%
% OUTPUT:   none
%
% SUBROUTINES REQUIRED: GetVertex.m
%                       SearchDist.m
%
% ========================================================================

function AddChildren( n, level );

global Node nNodes Vertex nVertex MaxLevel gmode

n1 = nNodes + 1;
n2 = nNodes + 2;
n3 = nNodes + 3;
n4 = nNodes + 4;

%Point node n toward its new children

Node(n).Children = [n1 n2 n3 n4];

%Create new vertex or find existing vertex for each midpoint

Vx1 = Node(n).Vertex(1);
Vx2 = Node(n).Vertex(2);
Vx3 = Node(n).Vertex(3);
```

```
Vx1V = Vertex( Vx1 ).Vector;
Vx2V = Vertex( Vx2 ).Vector;
Vx3V = Vertex( Vx3 ).Vector;

Vx12 = GetVertex( Node(n).Vertex(1), Node(n).Vertex(2) );
Vx23 = GetVertex( Node(n).Vertex(2), Node(n).Vertex(3) );
Vx31 = GetVertex( Node(n).Vertex(3), Node(n).Vertex(1) );

Vx12V = Vertex( Vx12 ).Vector;
Vx23V = Vertex( Vx23 ).Vector;
Vx31V = Vertex( Vx31 ).Vector;

%Child #1

Node(n1).Parent = n;
Node(n1).Children = [];
Node(n1).Vertex = [ Vx1 Vx12 Vx31 ];
Node(n1).Stars = [];
Node(n1).Level = level;

if gmode ~= 0
    PlotNode( n1, 'r' )
end

Node(n1).SDist = SearchDist( Vx1V, Vx12V, Vx31V );

%Child #2

Node(n2).Parent = n;
Node(n2).Children = [];
Node(n2).Vertex = [ Vx2 Vx23 Vx12 ];
Node(n2).Stars = [];
Node(n2).Level = level;

if gmode ~= 0
    PlotNode( n2, 'r' )
end

Node(n2).SDist = SearchDist( Vx2V, Vx23V, Vx12V );

%Child #3
```

```
Node(n3).Parent = n;
Node(n3).Children = [];
Node(n3).Vertex = [ Vx3 Vx31 Vx23 ];
Node(n3).Stars = [];
Node(n3).Level = level;

if gmode ~= 0
    PlotNode( n3, 'r' )
end

Node(n3).SDist = SearchDist( Vx3V, Vx31V, Vx23V );

%Child #4

Node(n4).Parent = n;
Node(n4).Children = [];
Node(n4).Vertex = [ Vx12 Vx23 Vx31 ];
Node(n4).Stars = [];
Node(n4).Level = level;

if gmode ~= 0
    PlotNode( n4, 'r' )
end

Node(n4).SDist = SearchDist( Vx12V, Vx23V, Vx31V );

% Add nodes to verticies so all nodes connected to a vertex can be found.
% Only add if at desired depth

if level == MaxLevel

    Vertex(Vx1).Nodes = [ Vertex(Vx1).Nodes n1 ];
    Vertex(Vx2).Nodes = [ Vertex(Vx2).Nodes n2 ];
    Vertex(Vx3).Nodes = [ Vertex(Vx3).Nodes n3 ];

    Vertex(Vx12).Nodes = [ Vertex(Vx12).Nodes n1 n4 n2 ];
    Vertex(Vx23).Nodes = [ Vertex(Vx23).Nodes n2 n4 n3 ];
    Vertex(Vx31).Nodes = [ Vertex(Vx31).Nodes n3 n4 n1 ];

end
```

```
% Number of nodes increased by four

nNodes = nNodes + 4;
```

E.2. AddNoise.m

```
% ========================================================================
%
% AddNoise.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Adds measurement noise of a normal distribution to vector
%
% INPUTS:   vt - true vector
%           sigm - measurement standard deviation
%
% OUTPUT:   vm - measurement of vector
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function vm = AddNoise( vt, sigm );

vm = vt + sigm * randn( 3, 1 );
vm = vm / norm( vm );
```

E.3. AddSphTriProps.m

```
% ========================================================================
%
% AddSphTriProps
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% DESCRIP
%
% INPUTS:   SphTrixxxx - Catalog of Triangles
%           Stars - List of stars
%
% OUTPUT:   SphTri2xxxx - Catalog of Triangle with Area and Ip
%
% SUBROUTINES REQUIRED: SphTriArea.m
%                       SphTriPolarMoment.m
%
% ========================================================================

%Calculate properties of triangles

load SphTriM60L4;
load Stars;

for i=1:nTri
    if i/100 == floor(i/100)
        [ i nTri ]
    end

    v1 = Star( Tri(i).Stars(1) ).Vector;
    v2 = Star( Tri(i).Stars(2) ).Vector;
    v3 = Star( Tri(i).Stars(3) ).Vector;

    Tri(i).Area = SphTriArea( v1, v2, v3 );
    Tri(i).Ip   = SphTriPolarMoment( v1, v2, v3, 3, 0, 0 );
end

save SphTri2M60L4 Tri nTri FOVmax
```

E.4. ArcMidPt.m

```
% ========================================================================
%
% ArcMidPt.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Returns unit vector to midpoint of arc between two vectors
%
% INPUTS:   v1 - vector 1
%           v2 - vector 2
%
% OUTPUT:   m - vector pointing to midpoint of arc between v1 and v2
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function m = ArcMidPt( v1, v2 );

r = v2 - v1;

m = v1 + r ./ 2;

m = m / norm(m);
```

E.5. BigBTreeAdd.m

```
% ========================================================================
%
% BigBTreeAdd.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Adds elements to binary tree.
% "Big" routines best for large amounts of data, since
% memory is preallocated, but requires global variables and so only
% one can be made at a time.
%
% INPUTS:   Val - value to be stored in binary tree
%           n   - location in binary tree array (typically at end + 1)
%
% OUTPUT:   Ptr - pointer array of data sorted by val
%
% SUBROUTINES REQUIRED: BigBTreeAdd.m
%                       BigBTreeAscend.m
%
% ========================================================================

function BigBTreeAdd( Val, n )

global TreeVal TreeLeft TreeRight

% New value placed at current end of array

% n = size( TreeVal, 2 );
% n = n + 1;

TreeVal(n)   = Val;
TreeLeft(n)  = 0;       % Indicate end of left branch
TreeRight(n) = 0;      % Indicate end of right branch

if n == 1                  % If it's the first item in the tree, its the root
    done = 1;
else
```

```
    done = 0;
end

% Move through tree until proper position for value is found

i = 1;      % Start at root of tree (first item in tree array)

while done == 0         % repeat until Tri(j) positioned in tree
    if Val < TreeVal(i)        % if value < current leaf in tree, move left
        if TreeLeft(i) == 0;
            TreeLeft(i) = n;
            done = 1;
        else
            i = TreeLeft(i);
        end

    elseif Val >= TreeVal(i)    % if value >= current leaf in tree, move right
        if TreeRight(i) == 0;
            TreeRight(i) = n;
            done = 1;
        else
            i = TreeRight(i);
        end
    end
end
```

E.6. BigBTreeAscend.m

```
% ========================================================================
%
% BigBTreeAscend.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Recursive routine to move through binary tree in order. Stores
% order in pointer array
% "Big" routines best for large amounts of data, since
% memory is preallocated, but requires global variables and so only
% one can be made at a time.
%
% INPUTS:   x - current binary tree location (enter with x = root)
%           i - pointer array location (start i = 0)
%
% OUTPUT:   i - new pointer array location
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function i = BigBTreeAscend( x, i )

global TreeLeft TreeRight Ptr

if x ~= 0
    i = BigBTreeAscend( TreeLeft(x), i );

    i = i + 1;
    Ptr(i) = x;

    if i/1000 == floor(i/1000)
        [i]
    end

    i = BigBTreeAscend( TreeRight(x), i );
end
```

E.7. BigBTreeSort.m

```
% ========================================================================
%
% BigBTreeSort.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Sorts Property.Data by Property.Val and returns sorted pointer array
% using binary trees. "Big" routines best for large amounts of data, since
% memory is preallocated, but requires global variables and so only
% one can be made at a time.
%
% INPUTS:   Prop - structure to be sorted
%             .val - value by which they are to be sorted
%             .data - data to be sorted
%
% OUTPUT:   Ptr - pointer array of data sorted by val
%
% SUBROUTINES REQUIRED: BigBTreeAdd.m
%                       BigBTreeAscend.m
%
% ========================================================================

function Ptr = BigBTreeSort( Prop );

global TreeVal TreeLeft TreeRight Ptr

% Create binary tree of Prop

'Creating Tree'

n = size( Prop, 2 );
Ptr = zeros( 1, n );

%Tree = [];
for i=1:n
    if i/1000 == floor(i/1000)
        [i n]
```

```
    end

    BigBTreeAdd( Prop(i), i );
end

'Ascending Tree'

% Create an ordered pointer arry of Prop

root = 1;
i = BigBTreeAscend( root, 0 )  % Output tree in order
```

E.8. BTreeAdd.m

```
% ========================================================================
%
% BTreeAdd.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Adds element to binary tree in order of val.
%
% INPUTS:   Val - value to determine sort
%           Data - data to be stored with it
%           Tree - Binary Tree to search
%
% OUTPUT:   Tree - updated binary tree
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function Tree = BTreeAdd( Val, Data, Tree )

% New value placed at current end of array

n = size( Tree, 2 );
n = n + 1;

Tree(n).Val   = Val;
Tree(n).Data  = Data;
Tree(n).left  = 0;       % Indicate end of left branch
Tree(n).right = 0;      % Indicate end of right branch

if n == 1                  % If it's the first item in the tree, its the root
    done = 1;
else
    done = 0;
end

% Move through tree until proper position for value is found
```

```
i = 1;      % Start at root of tree (first item in tree array)

while done == 0         % repeat until Tri(j) positioned in tree
    if Val < Tree(i).Val        % if value < current leaf in tree, move left
        if Tree(i).left == 0;
            Tree(i).left = n;
            done = 1;
        else
            i = Tree(i).left;
        end

    elseif Val >= Tree(i).Val    % if value >= current leaf in tree, move right
        if Tree(i).right == 0;
            Tree(i).right = n;
            done = 1;
        else
            i = Tree(i).right;
        end
    end
end
```

E.9. BTreeFind.m

```
% ========================================================================
%
% BTreeFind.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Locates value in binary tree. If more than one exist, an array is output
% of all locations.
%
% INPUTS:   Val - value to be found
%           Tree - Binary Tree to search
%
% OUTPUT:   Ptr - array of locations where value was found
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function Ptr = BTreeFind( Val, Tree )

% Find location of Value in binary tree
% Tree Structure:   Tree.Val   - Item to be stored
%                   Tree.left  - pointer to left branch
%                   Tree.right - pointer to right branch

% Move throught tree until value is found or search is exhausted

i = 1;      % Start at root of tree (first item in tree array)

Ptr = [];
done = 0;
while done == 0

    if Val == Tree(i).Val
        Ptr = [ Ptr i ];
    end
```

```
    if Val < Tree(i).Val        % if value < current leaf in tree, move left
        if Tree(i).left == 0;
            done = 1;
        else
            i = Tree(i).left;
        end
    elseif Val >= Tree(i).Val    % if value >= current leaf in tree, move right
        if Tree(i).right == 0;
            done = 1;
        else
            i = Tree(i).right;
        end
    end
end
```

E.10. CalcFOV.m

```
% ========================================================================
%
% CalcFOV.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Determines the field of view required to see spherical triangle
%
% INPUTS:   p1 - vector of triangle vertex no. 1
%           p2 - vector of triangle vertex no. 2
%           p3 - vector of triangle vertex no. 3
%
% OUTPUT:   alpha - diameter of required field of view (rads)
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function alpha = CalcFOV( p1, p2, p3 )

% Point is also the vector from the origin to the point.

% Vector r from each pt to another pt. Length of each r is a side of a triangle

r12 = p2 - p1;
r23 = p3 - p2;
r31 = p1 - p3;

% Calc length of vector from pt to pt

a = sqrt( r12(1)^2 + r12(2)^2 + r12(3)^2 );
b = sqrt( r23(1)^2 + r23(2)^2 + r23(3)^2 );
c = sqrt( r31(1)^2 + r31(2)^2 + r31(3)^2 );

% Determine interior angles of triangle

A = acos( (b^2 + c^2 - a^2)/(2*b*c) );
```

```
% A*180/pi
B = acos( (a^2 + c^2 - b^2)/(2*a*c) );
% B*180/pi
C = acos( (a^2 + b^2 - c^2)/(2*b*a) );
% C*180/pi

% If all the angles within the triangle are less than 90 degrees, the field of
% view is deterimned by drawing a circle through the three pts

% If any angle is greater than 90 degrees, the field of view is determined
% by the greatest distance from any pt to another.

r = 0;

if max( [ A B C ] ) < (pi/2)

    % CalcFOV determines the field of view occupied by three pts on a sphere
    % p1, p2 and p3 are arrays: [ x y z ]

    % Distance to center from pt 1 to pt 2 must be equal (eq 1)

    A1(1) = - 2*p1(1) + 2*p2(1);
    A1(2) = - 2*p1(2) + 2*p2(2);
    A1(3) = - 2*p1(3) + 2*p2(3);
    B(1) = - (p1(1)^2 + p1(2)^2 + p1(3)^2) + (p2(1)^2 + p2(2)^2 + p2(3)^2);

    % Distance to center from pt 2 to pt 3 must be equal (eq 2)

    A2(1) = - 2*p2(1) + 2*p3(1);
    A2(2) = - 2*p2(2) + 2*p3(2);
    A2(3) = - 2*p2(3) + 2*p3(3);
    B(2) = - (p2(1)^2 + p2(2)^2 + p2(3)^2) + (p3(1)^2 + p3(2)^2 + p3(3)^2);

    % All points must be on the same plane (eq 3)

    P = [ -p1'; p2'-p1'; p3'-p1' ];

    A3(1) = ( P(2,2) * P(3,3) ) - ( P(2,3) * P(3,2) );
    A3(2) = ( P(2,3) * P(3,1) ) - ( P(2,1) * P(3,3) );
    A3(3) = ( P(2,1) * P(3,2) ) - ( P(2,2) * P(3,1) );
    B(3) = - det( P );
```

```
    A = [ A1; A2; A3 ];

    x = inv(A)*B';

    r = sqrt( (p1(1)-x(1))^2 + (p1(2)-x(2))^2 + (p1(3)-x(3))^2 );

else

    r = max( [a b c] )/2; % Make negative to indicate "skinny"

end

% FOV is twice the angle formed by triangle with opp=r and hyp=1

alpha = [ 2*asin(r/1) ];
% alpha*180/pi
```

E.11. CatalogAngles.m

```
% ========================================================================
%
% CatalogAngles.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Creates catalog of angles suitable for Angle Recognition Algorithm
%
% INPUTS -  Star data (Stars.mat)
%           Output of CreateQuadTree.m (QTxxxx.mat)
%
% OUTPUT -  Angle Catalog (Ang****.mat)
%
% SUBROUTINES REQUIRED: FindNeighborNodes.m
%                       PlotNode.m
%                       Find1NodeAngs.m
%                       Fing2NodeAngs.m
%                       WaitForKeyPress.m
%
% ========================================================================

clear all;

global Star Node FOVmax gmode

% Set graphics mode: gmode = 0   No graphics. Fastest results
%                            1   Stop after each node
%                            2   Stop after each angle

gmode = 1;

load QTM60L4
load Stars

FOVmax = SearchLimit;
FOVmax * 180/pi
```

```
nAng = 0;
Ang = [];

% Set up nodes to record when combos with two nodes are made

for i=1:nNodes
    Node(i).Doubles = [];
end

tic

% Go through all the nodes

for i=1:nNodes
    if Node(i).Level == MaxLevel

% Find all nodes surrounding center node

        NeighborNodes = FindNeighborNodes( i );     % Get array of surrounding nodes
        nNeighborNodes = size( NeighborNodes, 2 );

        if gmode ~= 0
            hold off;
            PlotNode( i, 'g' );
            hold on;
            grid on;

            for j=1:nNeighborNodes
                PlotNode( NeighborNodes(j),'b' );
            end
        end

% Find all suitable triangles within target node

        Ang = [Ang Find1NodeAngs( i, FOVmax )];

% Find all suitable triangles between target and any other surrounding node

        if nNeighborNodes >= 1

            for m = 1:nNeighborNodes
```

```
                j = NeighborNodes(m);

                % First check that these two nodes haven't been checked before

                k = find( Node(i).Doubles == j );

                if size( k,2 ) == 0
                    Ang = [Ang Find2NodeAngs( i, j, FOVmax )];

                    % Record nodes so they're not counted again

                    Node(i).Doubles = [ Node(i).Doubles j ];
                    Node(j).Doubles = [ Node(j).Doubles i ];
                end
            end
        end

        if gmode ~= 0
            WaitForKeyPress;
        end

        [i nNodes size( Ang,2 )]
        toc

    else
        [i nNodes size( Ang,2 )]
        toc

    end

end

nAng = size( Ang, 2 )

save AngM60L4 Ang nAng FOVmax
```

E.12. CatalogSphTris.m

```
% ========================================================================
%
% CatalogSphTris.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Creates a catalog of spherical triangles suitable for spherical triangle
% method.
%
% INPUTS:   Quad-Tree of Stars (QTxxxx.mat)
%           Stars (Stars.mat)
%
% OUTPUT:   Spherical Triangle Catalog (SphTrixxxx.mat)
%
% SUBROUTINES REQUIRED: FindNeighborNodes.m
%                       PlotNode.m
%                       Find1NodeSphTris.m
%                       Find2NodeSphTris.m
%                       Find3NodeSphTris.m
%                       WaitForKeyPress.m
%
% ========================================================================

clear all

global Star Node gmode  % FOVmax fast gmode

% Set graphics mode: gmode = 0   No graphics. Fastest results
%                            1   Summary
%                            2   Stop after each pair

gmode = 2;

load QTM60L4
load Stars

FOVmax = SearchLimit;
```

```
FOVmax * 180/pi

nTri = 0;
Tri = [];

% Set up nodes to record when combos with two and three nodes are made

for i=1:nNodes
    Node(i).Doubles = [];
    Node(i).Triples = [];
end

% Initialize plot if graphics are desired

if gmode ~= 0
    hold off;
    plot3( 0,0,0 ,'oy','MarkerSize',2);
    hold on;
%     axis( [-1 1 -1 1 -1 1] );
    grid on;
end

tic

% Go through all the nodes

for i=1:nNodes
    if Node(i).Level == MaxLevel

% Find all nodes surrounding center node

        NeighborNodes = FindNeighborNodes( i );     % Get array of surrounding nodes
        nNeighborNodes = size( NeighborNodes, 2 );

        switch( gmode )
            case 1
                PlotNode( i, 'r' );
            case 2
                PlotNode( i, 'g' );

                for j=1:nNeighborNodes
```

```
                PlotNode( NeighborNodes(j),'b' );
            end
        end

% Find all suitable triangles within target node

        newTri = Find1NodeSphTris( i, FOVmax );
        Tri = [Tri newTri];

% Find all suitable triangles between target and any other surrounding node

        if nNeighborNodes >= 1

            for m = 1:nNeighborNodes
                j = NeighborNodes(m);

                % First check that these two nodes haven't been checked before

                k = find( Node(i).Doubles == j );

                if size( k,2 ) == 0
                    newTri = Find2NodeSphTris( i, j, FOVmax );
                    Tri = [ Tri newTri ];

                    % Record nodes so they're not counted again

                    Node(i).Doubles = [ Node(i).Doubles j ];
                    Node(j).Doubles = [ Node(j).Doubles i ];
                end
            end
        end

    % Find all suitable triangles between target and any two search nodes

        if nNeighborNodes >=3

            for m = 1:nNeighborNodes - 1
                j = NeighborNodes(m);

                for n = m+1:nNeighborNodes
                    k = NeighborNodes(n);
```

```
                % First check that these three nodes haven't been checked before

                    done = 0;
                    for x = 1:size( Node(i).Triples, 1 );
                        if Node(i).Triples(x,:) == [ j k ]
                            done = 1;
                            break;
                        elseif Node(i).Triples(x,:) == [ k j ]
                            done = 1;
                            break;
                        end
                    end

            % If not, count all triangles with a point in each node

                    if done == 0

                        newTri = Find3NodeSphTris( i, j, k, FOVmax );
                        Tri = [Tri newTri];

                        % Record nodes so they're not counted again

                        Node(i).Triples = [ Node(i).Triples; j k ];
                        Node(j).Triples = [ Node(j).Triples; i k ];
                        Node(k).Triples = [ Node(k).Triples; i j ];
                    end

                end
            end
        end

      if gmode == 1
          WaitForKeyPress;
      end
  end

    nTri = size( Tri, 2 );
    [i nNodes nTri]
    toc
end
```

```
save SphTriM60L4 Tri nTri FOVmax
```

E.13. CompileResults.m

```
% ========================================================================
%
% CompileResults.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Creates histograms, pie charts, etc of results.
%
% INPUTS:   Resultsxxxx - Output of RandAttTest.m
%
% OUTPUT:   none
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

clear all;

load Results
nResults = size( Results, 2 );

% Initialize variables for histograms

slim = 20       % Max number of stars to
fail(1:slim+1) = 0;
bad(1:slim+1) = 0;
correct(1:slim+1) = 0;
cant(1:slim+1) = 0;

tlim = 20       % Max number of seconds
timeFail(1:tlim+1) = 0;
timeBad(1:tlim+1) = 0;
timeCorrect(1:tlim+1) = 0;
timeCant(1:tlim+1) = 0;

plim = 20       % Max number of pivots
pivotFail(1:tlim+1) = 0;
```

```
pivotBad(1:tlim+1) = 0;
pivotCorrect(1:tlim+1) = 0;
pivotCant(1:tlim+1) = 0;

% Go through all the results

for i=1:nResults
    nStarsInFOV = size( Results(i).StarsInFOV, 2 );

    if nStarsInFOV >= 0

        % Histogram for stars in FOV vs. Occurances

        if nStarsInFOV > slim
            nStarsInFOV = slim;
        end

        fail(nStarsInFOV+1)    = fail(nStarsInFOV+1)    + Results(i).Fail;
        bad(nStarsInFOV+1)     = bad(nStarsInFOV+1)     + Results(i).Bad;
        correct(nStarsInFOV+1) = correct(nStarsInFOV+1) + Results(i).Correct;
        cant(nStarsInFOV+1)    = cant(nStarsInFOV+1)    + Results(i).Cant;

        t0 = round( Results(i).time + 1 );
        if t0 > tlim+1
            t0 = tlim+1;
        end

        timeFail(t0)    = timeFail(t0)    + Results(i).Fail;
        timeBad(t0)     = timeBad(t0)     + Results(i).Bad;
        timeCorrect(t0) = timeCorrect(t0) + Results(i).Correct;
        timeCant(t0)    = timeCant(t0)    + Results(i).Cant;

        p0 = Results(i).nPivots + 1;
        if p0 > plim + 1
            p0 = plim + 1;
        end

        pivotFail(p0)    = pivotFail(p0)    + Results(i).Fail;
        pivotBad(p0)     = pivotBad(p0)     + Results(i).Bad;
        pivotCorrect(p0) = pivotCorrect(p0) + Results(i).Correct;
        pivotCant(p0)    = pivotCant(p0)    + Results(i).Cant;
```

```
    end

end

figure(1);
bar( 0:slim, [correct; cant; fail; bad]','stacked' );
xlabel('Number of Stars in Field of View');
ylabel('Occurances')
axis tight;

% Pie chart of overall results

failsum = sum( fail(1:slim+1) );
badsum = sum( bad(1:slim+1) );
correctsum = sum( correct(1:slim+1) );
cantsum = sum( cant(1:slim+1) );

figure(2);
pie( [ correctsum, cantsum, failsum, badsum ], [ 1 0 0 0 ] );

% Histogram for CPU time required vs. Occurances

figure(3);
bar( 0:tlim, [timeCorrect; timeCant; timeFail; timeBad]','stacked' );
xlabel('Time Required (sec)');
ylabel('Occurances')
axis tight;
timemean = mean([Results.time])
timestd  = std([Results.time])
timemed  = median([Results.time])

% Histogram for pivots vs occurances

figure(4);
bar( 0:plim, [pivotCorrect; pivotCant; pivotFail; pivotBad]','stacked' );
xlabel('Pivots Required');
ylabel('Occurances')
axis tight;
pivotmean = mean([Results.nPivots])
pivotstd  = std([Results.nPivots])
```

```
pivotmed  = median([Results.nPivots])
```

E.14. CoordXform.m

```
% ========================================================================
%
% CoordXform.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Transforms point in cartesian from one frame to another
%
% INPUTS:   vi - vector in intertial frame
%           b  - body frame (unit vectors in inertial frame)
%
% OUTPUT:   vb - vector in body frame
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function vb = CoordXform( vi, bframe );

% Make axis of frames easier to see

i1 = [ 1 0 0 ]';
i2 = [ 0 1 0 ]';
i3 = [ 0 0 1 ]';

b1 = bframe.b1;
b2 = bframe.b2;
b3 = bframe.b3;

% Create Rotation Matrix

l1 = dot( i1, b1 );
l2 = dot( i1, b2 );
l3 = dot( i1, b3 );

m1 = dot( i2, b1 );
m2 = dot( i2, b2 );
```

```
m3 = dot( i2, b3 );

n1 = dot( i3, b1 );
n2 = dot( i3, b2 );
n3 = dot( i3, b3 );

R = [ l1 l2 l3; m1 m2 m3; n1 n2 n3 ];

% Determine new v in b coordinates

vb = R'*vi;
```

E.15. CreateLinKvector.m

```
% ========================================================================
%
% CreateLinKvector.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Creates K-Vector array based on line connecting first and last points
% of sorted data
%
% INPUTS:   Prop - Array of data for which K-Vector is created
%
% OUTPUT:   Kvec - structured array of K-vector
%             .Ptr - pointer array to sorted data
%             .A   - slope of K-Vector
%             .B   - y-intercept
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function Kvec = CreateLinKvector( Prop )

n = size( Prop, 2 );    % Determine size of array (its 1-d)

Kvec.Ptr  = zeros( 1, n );  % Preallocate memory

% Calculate line which passes through first and last point
% Area = a*index^2 + b

Kvec.A = ( Prop( n ) - Prop( 1 ) )/(n-1);
Kvec.B = Prop( 1 ) - Kvec.A*1;

% Plot as desired

figure(1);
i = 1:n;
y = Kvec.A * i + Kvec.B;
```

```
plot(1:n, [Prop(:)], 1:n, y(:));

% Create K-Vector

j = 1;
for i = 1:n-1

    if (i/1000) == floor(i/1000)
        [i n]
    end

    y = Kvec.A*i + Kvec.B;

    % Want K-Vector at i to refer to index of Area closest to but
    % not over Area resulting from parabola equation

    while Prop(j) <= y
        j = j + 1;
        if j > n
            j = n;
            break;
        end
    end

    j = j - 1;      % while loop ends at one index too high
                    % comment above line to make D'Mortari K-Vector
    Kvec.Ptr(i) = j;
end

Kvec.Ptr(n) = n;      % must set Kvec(n) manually
```

E.16. CreateParabKvector.m

```
% ========================================================================
%
% CreateParabKvector.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Creates K-Vector array based on parabola connecting first and last
% points of sorted data
%
% INPUTS:   Prop - Array of data for which K-Vector is created
%
% OUTPUT:   Kvec - structured array of K-vector
%             .Ptr - pointer array to sorted data
%             .A   - x-coefficient
%             .B   - y-intercept
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function Kvec = CreateParabKvector( Prop )

n = size( Prop, 2 );    % Determine size of array (its 1-d)

Kvec.Ptr  = zeros( 1, n );

% Calculate parabola which passes through first and last point
% Area = a*index^2 + b

Kvec.A = ( Prop( n ) - Prop( 1 ) )/(n^2-1);
Kvec.B = Prop( 1 ) - Kvec.A*1^2;

% Plot as desired

figure(2);
i = 1:n;
y = Kvec.A * i.^2 + Kvec.B;
```

```
plot(1:n, [Prop(:)], 1:n, y(:));

% Create K-vector

j = 1;
for i = 1:n-1

    if (i/1000) == floor(i/1000)
        [i n]
    end

    y = Kvec.A*i^2 + Kvec.B;

    % Want K-Vector at i to refer to index of Area closest to but
    % not over Area resulting from parabola equation

    while Prop(j) <= y
        j = j + 1;
        if j > n
            j = n;
            break;
        end
    end

    j = j - 1;      % while loop ends at one index too high
                    % comment above line to make D'Mortari K-Vector
    Kvec.Ptr(i) = j;
end

Kvec.Ptr(n) = n;      % must set Kvec(n) manually
```

E.17. CreateStarQuadTree.m

```
% ========================================================================
%
% CreateStarQuadTree.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Creates a quad-tree of stars so that angle and spherical triangle
% catalogs can be created efficiently. Maximum search distance
% also calculated.
%
% INPUTS:   Star List (Stars.mat)
%
% OUTPUT:   Quad-Tree (QTxxx.mat)
%
% SUBROUTINES REQUIRED: PlotNode.m
%                       GetNodeNum.m
%                       WaitForKeyPress.m
%
% ========================================================================

clear all;

global MaxLevel gmode

MagLimit = 6;
MaxLevel = 4;

gmode = 2;  %1 = graphics yes, 0 = no graphics, 2= wait

%Calculate max number of nodes (Always less verticies)

maxnodes = 0;
for i=1:MaxLevel
    maxnodes = maxnodes + 4^i;
end
maxnodes = maxnodes
```

```
load Stars

tic

% Initialize plot if graphics are desired

if gmode ~= 0
  hold off;
  plot3( 0,0,0 ,'om','MarkerSize',2);
  hold on;
  axis( [-1 1 -1 1 -1 1] );
  grid on;
end

InitVertices;
InitNodes;

%  Put stars into quad-tree

for i=1:nStars
    [i nStars]

    if gmode == 2
        hold off;
        PlotNode(1,'r');
        hold on;
        PlotNode(2,'r');
        PlotNode(3,'r');
        PlotNode(4,'r');
        plot3( 0,0,0 ,'oy','MarkerSize',2);
        axis( [-1 1 -1 1 -1 1] );
        grid on;
    end

    if Star(i).Mag < MagLimit

        if gmode ~= 0
            plot3( Star(i).Vector(1), Star(i).Vector(2), ...
                   Star(i).Vector(3),'og','MarkerSize',2);
        end
```

```
        % Find node in which star should reside, then add star to it

        nnum = GetNodeNum( Star(i).Vector );
        Node(nnum).Stars = [ Node(nnum).Stars i ];

    else
        break
    end

% Good for demo -- wait for keypress between stars

    if gmode == 2
        WaitForKeyPress;
    end
end

toc

% Just a check... should add up to the correct number of stars

nStars = 0;
for i=1:nNodes
    nStars = nStars + size( Node(i).Stars, 2 );
end
nStars

% Find distance that can be searched with this level quad-tree

SearchLimit = 2*pi;
for i=1:nNodes
    if Node(i).SDist < SearchLimit
        SearchLimit = Node(i).SDist;
    end
end
SearchLimit*180/pi

save QTM60L4 Node nNodes Vertex nVertex MagLimit MaxLevel SearchLimit
```

E.18. Find1NodeAngs.m

```
% ========================================================================
%
% Find1NodeAngs.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Finds all angles <= FOVmax within target node
%
% INPUTS:   i - targetNode
%           FOVmax - maximum arc (rads)
%
% OUTPUT:   Ang - structure array of angles <= FOVmax in target node
%             .Stars - star nos. that make up the angle
%             .dotpr - dot product of vectors to stars
%                      (or cosine of angle)
%
% SUBROUTINES REQUIRED: PlotArc.m
%                       WaitForKeyPress.m
%
% ========================================================================

function Ang = Find1NodeAngs( i, FOVmax )

global Node Star gmode

nStars = size( Node(i).Stars, 2 );

nAng = 0;
Ang = [];

if nStars >=2

    for j = 1:nStars - 1
        s1 = Node(i).Stars(j);
        s1V = Star(s1).Vector;

        for k = j+1:nStars
```

```
            s2 = Node(i).Stars(k);
            s2V = Star(s2).Vector;

            if gmode ~= 0
                plot3( s1V(1), s1V(2), s1V(3),'og','MarkerSize',2);
                plot3( s2V(1), s2V(2), s2V(3),'or','MarkerSize',2);
            end

            dotpr = dot( s1V, s2V );

            if acos( dotpr ) <= FOVmax
                nAng = nAng+1;
                Ang(nAng).Stars = [ s1 s2 ];
                Ang(nAng).dotpr = dot( s1V, s2V );

                if gmode ~= 0
                    PlotArc( s1V, s2V, 'b' );
                    if gmode == 2
                        WaitForKeyPress;
                    end
                end
            end

        end

    end

end
```

E.19. Find1NodeSphTris.m

```
% ========================================================================
%
% Find1NodeSphTris.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Finds all spherical triangles that occupy a FOV less than FOVmax within
% target node.
%
% INPUTS:   i - target node no.
%           FOVmax - FOV limit for spherical triangle
%
% OUTPUT:   Tri - structure array of angles <= FOVmax in target node
%             .Stars - star nos. that make up the angle
%             .dotpr - dot product of vectors to stars
%                      (or cosine of angle)
%
% SUBROUTINES REQUIRED: PlotArc.m
%                       WaitForKeyPress.m
%
% ========================================================================

function Tri = Find1NodeSphTris( i, FOVmax )

global Node Star gmode

nStars = size( Node(i).Stars, 2 );

nTri = 0;
Tri = [];

if gmode == 2
    for j=1:nStars
        s1  = Node(i).Stars(j);
        s1V = Star(s1).Vector;

        plot3( s1V(1), s1V(2), s1V(3),'or','MarkerSize',2);
```

```
    end
end

if nStars >=3

    for j = 1:nStars - 2
        s1  = Node(i).Stars(j);
        s1V = Star(s1).Vector;

        for k = j+1:nStars - 1
            s2  = Node(i).Stars(k);
            s2V = Star(s2).Vector;

            if acos( dot(s1V,s2V) ) < FOVmax

                for l = k+1:nStars
                    s3  = Node(i).Stars(l);
                    s3V = Star(s3).Vector;

                    if acos( dot(s1V,s3V) ) < FOVmax
                        if acos( dot( s2V, s3V) ) < FOVmax

                            FOV = CalcFOV( s1V, s2V, s3V );

                            if FOV <= FOVmax
                                nTri = nTri+1;
                                Tri(nTri).Stars = sort( [ s1 s2 s3 ] );
                                Tri(nTri).FOV = FOV;

                                if gmode == 2
                                    PlotSphericalTri( s1V, s2V, s3V, 'm' );
                                    WaitForKeyPress;
                                end
                            end

                        end
                    end

                end

            end
```

```
        end

    end

end
```

E.20. Find2NodeAngs.m

```
% ========================================================================
%
% Find2NodeAngs.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Makes a list of all angles <=FOVmax which have one star in one node,
% and one in another.
%
% INPUTS:   i - first node no.
%           j - second node no.
%           FOVmax - limit angle (rads)
%
% OUTPUT:   Ang - structure array of angles <= FOVmax in target node
%             .Stars - star nos. that make up the angle
%             .dotpr - dot product of vectors to stars
%                      (or cosine of angle)
%
% SUBROUTINES REQUIRED: PlotArc.m
%                       WaitForKeyPress.m
%
% ========================================================================

function Ang = Find2NodeAngs( i, j, FOVmax )

global Node Star gmode

nAng = 0;
Ang = [];

for x = 1:size( Node(i).Stars, 2 )
    s1 = Node(i).Stars(x);
    s1V = Star(s1).Vector;

    for y = 1:size( Node(j).Stars, 2 )
        s2 = Node(j).Stars(y);
        s2V = Star(s2).Vector;
```

```
        if gmode ~= 0
            plot3( s1V(1), s1V(2), s1V(3),'og','MarkerSize',2);
            plot3( s2V(1), s2V(2), s2V(3),'or','MarkerSize',2);
        end

        dotpr = dot( s1V, s2V );

        if acos( dotpr ) <= FOVmax
            nAng = nAng+1;
            Ang(nAng).Stars = [ s1 s2 ];
            Ang(nAng).dotpr = dotpr;

            if gmode ~= 0
                PlotArc( s1V, s2V, 'b' );

                if gmode == 2
                    WaitForKeyPress;
                end
            end
        end

    end
end
```

E.21. Find2NodeSphTris.m

```
% ========================================================================
%
% Find2NodeSphTris.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Finds all spherical triangles that occupy a FOV less than FOVmax.
% Spherical triangles have one vertex in node i, two in node j, or
% vice-versa.
%
% INPUTS:   i - first node no.
%           j - second node no.
%           FOVmax - FOV limit for spherical triangle
%
% OUTPUT:   Tri - structure array of angles <= FOVmax in target node
%             .Stars - star nos. that make up the angle
%             .dotpr - dot product of vectors to stars
%                      (or cosine of angle)
%
% SUBROUTINES REQUIRED: PlotArc.m
%                       WaitForKeyPress.m
%
% ========================================================================

function Tri = Find2NodeSphTris( i, j, FOVmax )

global Node Star gmode

nStarsi = size( Node(i).Stars, 2 );
nStarsj = size( Node(j).Stars, 2 );

nTri = 0;
Tri = [];

if gmode == 2
    for k=1:nStarsi
        s1  = Node(i).Stars(k);
```

```
        s1V = Star(s1).Vector;
        plot3( s1V(1), s1V(2), s1V(3),'or','MarkerSize',2);
    end
    for k=1:nStarsj
        s1  = Node(j).Stars(k);
        s1V = Star(s1).Vector;
        plot3( s1V(1), s1V(2), s1V(3),'or','MarkerSize',2);
    end
end

if nStarsi >= 2

    % Two stars from target node, one star from surrounding node

    for x = 1:nStarsi - 1
        s1  = Node(i).Stars(x);
        s1V = Star(s1).Vector;

        for y = x+1:nStarsi
            s2 = Node(i).Stars(y);
            s2V = Star(s2).Vector;

            if acos( dot( s1V, s2V ) ) < FOVmax

                for z = 1:nStarsj
                    s3 = Node(j).Stars(z);
                    s3V = Star(s3).Vector;

                    if acos( dot( s1V, s3V )) < FOVmax
                        if acos( dot( s2V, s3V )) < FOVmax

                            FOV = CalcFOV( s1V, s2V, s3V );

                            if FOV <= FOVmax
                                nTri = nTri + 1;
                                Tri(nTri).Stars = sort( [ s1 s2 s3 ] );
                                Tri(nTri).FOV = FOV;

                                if gmode ~= 0
                                    PlotSphericalTri( s1V, s2V, s3V, 'm' );
                                end
```

```
                            end

                        end
                    end

                end

            end
        end
    end

end

% Repeat with 1 star from target node, two stars from surrounding node

nStars = size( Node(j).Stars, 2 );

if nStars >= 2

    for x = 1:size( Node(i).Stars, 2 )
    s1 = Node(i).Stars(x);
    s1V = Star(s1).Vector;

        for y = 1:size( Node(j).Stars, 2 ) - 1
            s2 = Node(j).Stars(y);
            s2V = Star(s2).Vector;

            if acos( dot( s1V, s2V ) ) < FOVmax

                for z = y+1:size( Node(j).Stars, 2 )
                    s3 = Node(j).Stars(z);
                    s3V = Star(s3).Vector;

                    if acos( dot( s1V, s3V ) ) < FOVmax
                        if acos( dot( s2V, s3V ) ) < FOVmax

                            FOV = CalcFOV( s1V, s2V, s3V );

                            if FOV <= FOVmax
                                nTri = nTri + 1;
                                Tri(nTri).Stars = sort( [ s1 s2 s3 ] );
```

```
                                Tri(nTri).FOV = FOV;

                                if gmode ~= 0
                                    PlotSphericalTri( s1V, s2V, s3V, 'm' );
                                end
                            end

                        end
                    end

                end

            end
        end
    end

end
```

E.22. Find3NodeSphTris.m

```
% ========================================================================
%
% Find3NodeSphTris.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Finds all spherical triangles that occupy a FOV less than FOVmax.
% Spherical triangles have one vertex in each of the three nodes.
%
% INPUTS:   i - first node no.
%           j - second node no.
%           k - third node no.
%           FOVmax - FOV limit for spherical triangle
%
% OUTPUT:   Tri - structure array of angles <= FOVmax in target node
%             .Stars - star nos. that make up the angle
%             .dotpr - dot product of vectors to stars
%                      (or cosine of angle)
%
% SUBROUTINES REQUIRED: PlotArc.m
%                       WaitForKeyPress.m
%
% ========================================================================

function Tri = Find3NodeSphTris( i, j, k, FOVmax )

global Node Star gmode

nTri = 0;
Tri = [];

nStarsi = size( Node(i).Stars, 2 );
nStarsj = size( Node(j).Stars, 2 );
nStarsk = size( Node(k).Stars, 2 );

if gmode == 2
    for n=1:nStarsi
```

```
        s1  = Node(i).Stars(n);
        s1V = Star(s1).Vector;
        plot3( s1V(1), s1V(2), s1V(3),'or','MarkerSize',2);
    end
    for n=1:nStarsj
        s1  = Node(j).Stars(n);
        s1V = Star(s1).Vector;
        plot3( s1V(1), s1V(2), s1V(3),'or','MarkerSize',2);
    end
    for n=1:nStarsk
        s1  = Node(k).Stars(n);
        s1V = Star(s1).Vector;
        plot3( s1V(1), s1V(2), s1V(3),'or','MarkerSize',2);
    end
end

for x = 1:size( Node(i).Stars, 2 )
    s1  = Node(i).Stars(x);
    s1V = Star(s1).Vector;

    for y = 1:size( Node(j).Stars, 2 )
        s2  = Node(j).Stars(y);
        s2V = Star(s2).Vector;

        if acos( dot( s1V, s2V ) ) < FOVmax

            for z = 1:size( Node(k).Stars, 2 )
                s3  = Node(k).Stars(z);
                s3V = Star(s3).Vector;

                if acos( dot( s1V, s3V ) ) < FOVmax
                    if acos( dot( s2V, s3V ) ) < FOVmax

                        FOV = CalcFOV( s1V, s2V, s3V );

                        if FOV <= FOVmax
                            nTri = nTri+1;
                            Tri(nTri).Stars = sort( [ s1 s2 s3 ] );
                            Tri(nTri).FOV = FOV;

                            if gmode ~= 0
```

```
                                PlotSphericalTri( s1V, s2V, s3V, 'm' );
                                WaitForKeyPress;
                            end
                        end

                    end

                end
            end

        end

    end

end
```

E.23. FindNeighborNodes.m

```
% ========================================================================
%
% FindNeighborNodes.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Creates a list of target-level quad-tree nodes which surround target node.
%
% INPUTS:   tn - target node number
%
% OUTPUT:   NeighborNodes - array of nodes surrounding target node.
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function NeighborNodes=FindNeighborNodes( tn )

% Search node does not include target node (tn)
% Doesn't check level because when QT is created, only nodes at maximum
% depth are attached to vertexes (Vertex.Nodes)

global Node Vertex MaxLevel

% Construct a list of nodes surrounding inner node

NeighborNodes = [];

for i=1:3                                   % Three verticies per node

    vx = Node(tn).Vertex(i);

    for j=1:size( Vertex(vx).Nodes, 2 )

        snnum = Vertex(vx).Nodes(j);        % Get search node number

        if snnum ~= tn                      % Don't include target node
```

```
            k = find( NeighborNodes == snnum );
            if size( k,2 ) == 0
                NeighborNodes = [ NeighborNodes snnum ];
            end

        end
    end
end
```

E.24. FindStarsInFOV.m

```
% ========================================================================
%
% FindStarsInFOV.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Given a vector and field of view, this routine will output an array of
% all the stars in the field of view.
%
% INPUTS:   trvector - star tracker camera vector
%           FOV - field of view of camera
%           QTxxxx - quad-tree of stars (need fresh version loaded ea time)
%
% OUTPUT:   StarList - array of stars
%
% SUBROUTINES REQUIRED: GetNodeNum.m
%                       PlotNode.m
%                       FindNeighborNodes.m
%
% ========================================================================

function StarList = FindStarsInFOV( trvector, FOV )

global Star gmode Vertex nVertex

% Get node number in which vector points (nodes created as necessary)

load QTM60L4;   % Use unaltered version every time

nnum = GetNodeNum ( trvector );

if bitand( gmode, 1 ) == 1
    PlotNode( nnum, 'c' );
end

% Add stars in node to list if it is within FOV of tracker
```

```
nStars = 0;
StarList.Star = [];

for k = 1:size( Node(nnum).Stars, 2 )
    starnum = Node(nnum).Stars(k);

    theta = acos( dot( Star(starnum).Vector, trvector )/1 );

    if theta <= FOV/2
        nStars = nStars + 1;
        StarList(nStars).Star = starnum;
        if bitand( gmode, 1 ) == 1
            plot3( Star(starnum).Vector(1), Star(starnum).Vector(2), ...
                Star(starnum).Vector(3),'og','MarkerSize',2);
        end
    elseif bitand( gmode, 1) == 1
        plot3( Star(starnum).Vector(1), Star(starnum).Vector(2), ...
            Star(starnum).Vector(3),'or','MarkerSize',2);
    end
end

% Go through neighbor nodes and and stars to list if they are with FOV of tracker

NeighborNodes = FindNeighborNodes( nnum );     % Get array of surrounding nodes
nNeighborNodes = size( NeighborNodes, 2 );

for j = 1:nNeighborNodes

    if bitand( gmode, 1 ) == 1
        PlotNode( NeighborNodes(j), 'm' );
    end

    for k = 1:size( Node( NeighborNodes(j) ).Stars, 2 )
        starnum = Node( NeighborNodes(j) ).Stars(k);

        theta = acos( dot( Star(starnum).Vector, trvector )/1 );
        %             theta*180/pi

        if theta <= FOV/2
            nStars = nStars + 1;
            StarList(nStars).Star = starnum;
```

```
            if bitand( gmode, 1 ) == 1
                plot3( Star(starnum).Vector(1), Star(starnum).Vector(2), ...
                    Star(starnum).Vector(3),'og','MarkerSize',2);
            end
        elseif bitand( gmode, 1 ) == 1
            plot3( Star(starnum).Vector(1), Star(starnum).Vector(2), ...
                Star(starnum).Vector(3),'or','MarkerSize',2);
        end
    end
end
```

E.25. FindStDev.m

```
% ========================================================================
%
% FindStDev.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Plots standard deviation of a 1,000 random triangles measured 100 times.
%
% INPUTS:   SphTri2xxxx - catalog of spherical triangles with area and Ic
%
% OUTPUT:   none
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

clear all;

load Stars;
load SphTri2M60L4

sigm = (87.2665e-6)/3;  % Sigma bound for error (normal distribution)

for i=1:1000

    j = round( nTri*rand(1) );

    [i j]

    sv1.t = Star( Tri(j).Stars(1) ).Vector;
    sv2.t = Star( Tri(j).Stars(2) ).Vector;
    sv3.t = Star( Tri(j).Stars(3) ).Vector;

    Area(i) = SphericalTriArea( sv1.t, sv2.t, sv3.t );
    Ic(i)   = SphTriSecondMoment( sv1.t, sv2.t, sv3.t, 3, 0, 0 );

    for k = 1:100
```

```
        sv1.m = AddNoise( sv1.t, sigm );
        sv2.m = AddNoise( sv2.t, sigm );
        sv3.m = AddNoise( sv3.t, sigm );

        AreaM(k) = SphericalTriArea( sv1.m, sv2.m, sv3.m );
        IcM(k)   = SphTriSecondMoment( sv1.m, sv2.m, sv3.m, 4, 0, 0 );
    end
    StdevA(i) = std( AreaM );
    StdevI(i) = std( IcM );
end

% Find best fitting line

figure(1);
plot( Area, StdevA, 'or' );
title('Standard Deviation of Area Measurement vs. Triangle Area ');
xlabel('Triangle True Area');
ylabel('Measurement Standard Deviation');

figure(2);
plot( Ic, StdevI, 'or' );
title('Standard Deviation of Ic Measurement vs. Triangle Area ');
xlabel('Triangle True Second Moment');
ylabel('Measurement Standard Deviation');
```

E.26. FindWithLinKvec.m

```
% ========================================================================
%
% FindWithLinKvec.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Returns angle ptr index using K-Vector
%
% INPUTS:   Val - angle
%           Kvec - K-Vector array, and equation of line.
%
% OUTPUT:   i - index
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function i = FindWithLinKvec( Val, Kvec );

if Val < Kvec.B
    Val = Kvec.B;
end

x = floor( (Val - Kvec.B)/Kvec.A );

if x == 0
    x = 1;
elseif x > size( Kvec.Ptr, 2 )
    x = size( Kvec.Ptr, 2 );
end

i = Kvec.Ptr(x);

% Scan upward through triangles until match is found

% while Prop(i) < Val
%     i = i + 1;
```

```
% end
```

E.27. FindWithParabKvec.m

```
% ========================================================================
%
% FindWithParabKvec.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Returns spherical triangle ptr index using parabolic K-Vector
%
% INPUTS:   Val - area of spherical triangle
%           Kvec - K-Vector array, and equation of parabola.
%
% OUTPUT:   i - index
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function i = FindWithParabKvec( Val, Kvec );

if Val < (Kvec.B + Kvec.A)
    Val = Kvec.B + Kvec.A;
end

x = floor( sqrt( (Val - Kvec.B)/Kvec.A ) );

if x == 0
    x = 1;
elseif x > size( Kvec.Ptr, 2 )
    x = size( Kvec.Ptr, 2 );
end

i = Kvec.Ptr(x);

% Scan upward through triangles until match is found

% while Prop(i) < Val
%     i = i + 1;
```

```
% end
```

E.28. GetBodyFrame.m

```
% ========================================================================
%
% GetBodyFrame.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Returns random body frame such that b1 is colinear to v
%
% INPUTS:   v - vector for frame to be aligned
%
% OUTPUT:   bframe - 3x3 matrix, first col is b1, 2nd is b2 and 3rd is b3
%                    each is a vector in inertial frame
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function bframe = GetBodyFrame( v );

b1 = v;       % b1 aligned with v

br = b1;
while norm( cross( b1,br ) ) == 0  % Repeat until not nearly colinear
    br = GetRandomVector;    % Random vector
end

% b2 = b1 x br = random vector at right angle to b1

b2 = cross( b1, br );
b2 = b2 / norm( b2 );

% b3 = b1 x b2 = vector rt angle to b1 and b2

b3 = cross( b1, b2 );
b3 = b3 / norm( b3 );

bframe.b1 = b1;
```

```
bframe.b2 = b2;
bframe.b3 = b3;
```

E.29. GetNodeNum.m

```
% ========================================================================
%
% GetNodeNum.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Given a unit vector, this routine will determine what node number is lies
% within. If the node does not exist it will create nodes as necessary.
%
% INPUTS:   Vector - vector for which node will be found
%
% OUTPUT:   nnum - node number which vector should be placed
%
% SUBROUTINES REQUIRED: AddChildren.m
%                       WhichNode.m
%                       PlotNode.m
%
% ========================================================================

function nnum = GetNodeNum( Vector )

global Node Star MaxLevel gmode

% Drill through quad-tree to target level

for level=1:MaxLevel

    % Find children of current node (start with first fourn nodes in tree)

    switch level
        case 1                      % Start with first four nodes in Node list
            nodeptr = [1 2 3 4];
        otherwise                   % Otherwise use children of current node
            if size( Node(nnum).Children, 2 ) == 0      %If no children, add them
                AddChildren( nnum, level );
            end
            nodeptr = Node(nnum).Children;
```

```
    end

    % Determine which of the four nodes vector should be placed

    nnum = WhichNode( Vector, nodeptr );

    if gmode ~= 0
        if level == MaxLevel
            PlotNode( nnum, 'r' );
        end
    end

end
```

E.30. GetRandomVector.m

```
% ========================================================================
%
% GetRandomVector.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Creates random unit vector in space
%
% INPUTS: none
%
% OUTPUT: v - random vector in cartesian coords
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function v = GetRandomVector

% Random attitude in spherical coords

alpha = rand(1)*2*pi;       % Az:  Range from 0 to 360 degrees
beta = rand(1)*pi - pi/2;   % Alt: Range from -90 to 90 degrees

% Convert to cartesian coords

v = [ cos( beta )*cos( alpha ); cos( beta )*sin( alpha ); sin( beta ) ];
```

E.31. GetVertex.m

```
% ========================================================================
%
% GetVertex.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Find if the mid-point vertex desired already exists. If so, it returns
% the vertex number. If the vertex doesn't exist, it is created and added
% to the list of verticies and the number of that vertex is returned.
%
% INPUTS:   Vx1 - Vertex no. 1
%           Vx2 - Vertex no. 2
%
% OUTPUT:   x = vertex no at midpoint of Vertex no 1 and 2
%
% SUBROUTINES REQUIRED: ArcMidPt.m
%
% ========================================================================

function x=GetVertex( Vx1, Vx2 );

global Vertex nVertex gmode

% Check to see if the asked-for vertex already exists.

i = find( Vertex(Vx1).Neighbor == Vx2 );

if size( i,2 ) ~= 0

    x = Vertex(Vx1).Child( i );

    if gmode ~= 0
        plot3( Vertex(x).Vector(1), Vertex(x).Vector(2), ...
            Vertex(x).Vector(3),'ob','MarkerSize',7);
    end

else
```

```
    nVertex = nVertex + 1;

    Vertex(nVertex).Vector = ArcMidPt( Vertex(Vx1).Vector, ...
        Vertex(Vx2).Vector );
    Vertex(nVertex).Neighbor = [];
    Vertex(nVertex).Nodes = [];

    Vertex(Vx1).Neighbor = [ Vertex(Vx1).Neighbor Vx2 ];
    Vertex(Vx1).Child = [ Vertex(Vx1).Child nVertex ];

    Vertex(Vx2).Neighbor = [ Vertex(Vx2).Neighbor Vx1 ];
    Vertex(Vx2).Child = [ Vertex(Vx2).Child nVertex ];

    x = nVertex;

    if gmode ~= 0
        plot3( Vertex(x).Vector(1), Vertex(x).Vector(2), ...
            Vertex(x).Vector(3),'ob','MarkerSize',7);
    end
end
```

E.32. InitNodes.m

```
% ========================================================================
%
% InitNodes.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Initializes first level nodes in quad-tree. Results is a spherical
% tetrahedron.
%
% INPUTS:   none
%
% OUTPUT:   none
%
% SUBROUTINES REQUIRED: SearchDist.m
%                       PlotNode.m
%
% ========================================================================

% Notice that search distance is subtracted from pi because the starting
% triangles have angles greater than 90 degrees.

global Node nNodes gmode

Vx1V = Vertex( 1 ).Vector;
Vx2V = Vertex( 2 ).Vector;
Vx3V = Vertex( 3 ).Vector;
Vx4V = Vertex( 4 ).Vector;

% Node 1

Node(1).Parent = 0;
Node(1).Vertex = [ 1 2 3 ];
Node(1).Children = [];
Node(1).Stars = [];
Node(1).Level = 1;

if gmode ~= 0
```

```
    PlotNode( 1,'r' );
end
Node(1).SDist = pi - SearchDist( Vx1V, Vx2V, Vx3V );

% Node 2

Node(2).Parent = 0;
Node(2).Vertex = [ 1 3 4 ];
Node(2).Children = [];
Node(2).Stars = [];
Node(2).Level = 1;

if gmode ~= 0
    PlotNode( 2, 'r' );
end
Node(2).SDist = pi - SearchDist( Vx1V, Vx3V, Vx4V );

% Node 3

Node(3).Parent = 0;
Node(3).Vertex = [ 1 4 2 ];
Node(3).Children = [];
Node(3).Stars = [];
Node(3).Level = 1;

if gmode ~= 0
    PlotNode( 3, 'r' );
end
Node(3).SDist = pi - SearchDist( Vx1V, Vx4V, Vx2V );

% Node 4

Node(4).Parent = 0;
Node(4).Vertex = [ 4 3 2 ];
Node(4).Children = [];
Node(4).Stars = [];
Node(4).Level = 1;

if gmode ~= 0
    PlotNode( 4, 'r');
end
```

```
Node(4).SDist = pi - SearchDist( Vx4V, Vx3V, Vx2V );

nNodes = 4;
```

E.33. InitVertices.m

```
% ========================================================================
%
% InitVertices.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Initializes first level vertices in quad-tree. Locations of vertices
% results in a sherical tetrahedron
%
% INPUTS:   none
%
% OUTPUT:   none
%
% SUBROUTINES REQUIRED: SearchDist.m
%                       PlotNode.m
%
% ========================================================================

global Vertex nVertex gmode

nVertex = 4;

% Tetrahedron, circumradius=1, centroid at (0,0,0)

R = 1;
a = 4 * R / sqrt(6);
x = sqrt(3)/3 * a;
r = sqrt(6)/12 * a;
d = sqrt(3)/6 * a;
% h = sqrt(6)/3 * a

% First four vertexes form a (spherical) tetrahedron

Vertex(1).Vector = [ 0 0 R ]';
Vertex(1).Neighbor = [];
Vertex(1).Child = [];
Vertex(1).Nodes = [];
```

```
Vertex(2).Vector = [ x 0 -r ]';
Vertex(2).Neighbor = [];
Vertex(2).Child = [];
Vertex(2).Nodes = [];

Vertex(3).Vector = [ -d a/2 -r ]';
Vertex(3).Neighbor = [];
Vertex(3).Child = [];
Vertex(3).Nodes = [];

Vertex(4).Vector = [ -d -a/2 -r ]';
Vertex(4).Neighbor = [];
Vertex(4).Child = [];
Vertex(4).Nodes = [];

% Plot if desired

if gmode ~= 0
    for i=1:4
        plot3( Vertex(i).Vector(1), Vertex(i).Vector(2), ...
            Vertex(i).Vector(3),'ob','MarkerSize',7);
    end
end
```

E.34. MatchStarsWAngs.m

```
% ========================================================================
%
% MatchStarsWAngs.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Takes a list of stars in FOV, and tries to match up stars in catalog
% using angle method.
%
% INPUTS:   StarsInFOV - Array of stars in field of view & measured vectors
%           climit - possible solution limit
%           plimit - pivot limit
%           sigm - standard deviation of measurement error
%           sigx - sigma bound for angle recognition (recommend 3)
%
% OUTPUT:   Results - structure with result
%
% SUBROUTINES REQUIRED: PlotInFOV.m
%                       BTreeAdd.m
%                       BTreeFind.m
%
% ========================================================================

function Results = MatchStarsWAngs( StarsInFOV, climit, plimit, sigm, sigx );

global gmode Ang nAng AngPtr Kvec FOV FOVmax

nStarsInFOV = size( StarsInFOV, 2 );

% IF LESS THAN TWO STARS, ANGLES CANNOT BE MADE

if nStarsInFOV < 2
    nCombs = 0;
    Combs = [];
    Results.nPivots = 0;
    Results.nFinalists = [];
    Results.Match = [];
```

```
    '-- NOT ENOUGH STARS --'
else
    nCombs = nchoosek( nStarsInFOV, 2 )

    % CREATE LIST OF ANGLES, INCL. AREA, AND MAX & MIN TRIANGLE INDICIES

    clear A;
    k = 0;
    start = 1;
    maxAng = 0;

    for s1=1:nStarsInFOV-1
        sv1 = StarsInFOV(s1).mv;
        for s2=s1+1:nStarsInFOV
            sv2 = StarsInFOV(s2).mv;

            mDotPr = dot( sv1, sv2 );
            mAng   = acos( mDotPr );

            if mAng < FOVmax

                Amin = mAng - sigm*sigx*sqrt(2);
                Amax = mAng + sigm*sigx*sqrt(2);

                imin = FindWithLinKvec( cos(Amax), Kvec );
                imax = FindWithLinKvec( cos(Amin), Kvec );

                range = imax - imin;

                if range <= climit      % Only keep with tris within combo limit
                    k = k + 1;
                    A(k).Stars = [ s1 s2 ];
                    A(k).mAng = mAng;
                    A(k).mDotPr = mDotPr;
                    A(k).imin = imin;
                    A(k).imax = imax;
                    A(k).llist = 0;           % Used for linked list

                    if A(k).mAng > maxAng
                        start = k;
                        maxAng = A(k).mAng;
```

```
                end

            end

        end

    end
end
nCombs = k

% CREATE LINKED LIST OF TRIANGLES, SUCH THAT NO MORE THAN ONE STAR CHANGES AT A TIME
% GIVING PRIORITY TO TRIANGLES WITH SMALLEST RANGE OF POSSIBLE SOLUTIONS

A(start).llist = 999;       % makes ineligible (also EOL)
prev = start;
next = start;
nPivots = 0;
for j=1:nCombs
    maxAng = 0;
    done = false;

    for k=1:nCombs
        if A(k).llist == 0
            x = 0;
            for m=1:2
                switch A(k).Stars(m)
                    case A(prev).Stars(1)
                        x = x + 1;
                    case A(prev).Stars(2)
                        x = x + 1;
                end
            end
            if x == 1
                if A(k).mAng > maxAng
                    next = k;
                    maxAng = A(k).mAng;
                end
                done = true;
            end
        end
    end
```

```
    A(prev).llist = next;
    A(next).llist = 999;
    prev = next;

    if done == true
        nPivots = nPivots + 1;
        if nPivots == plimit
            break;
        end
    else
        break;
    end
end

nPivots = nPivots

% Start with triangle with fewest possible solutions (first in list)

if nCombs > 0
    nFinalists = A(start).imax - A(start).imin

    for j=1:nFinalists
        anum = AngPtr( A(start).imin+j-1 );
        Finalists(j).Ang = anum;
        Finalists(j).Stars = Ang( anum ).Stars;
    end

    Results.nFinalists = nFinalists;

    if bitand( gmode, 4 ) == 4
        sv1 = StarsInFOV( A(start).Stars(1) ).mv;
        sv2 = StarsInFOV( A(start).Stars(2) ).mv;
        PlotInFOV( sv1, sv2, FOV, 0, '-g' );
        PlotInFOV( sv2, sv1, FOV, 0, '-g' );
    end
else
    nFinalists = 0;
    Results.nFinalists = 0;
end
```

```
    % PIVOT AS REQUIRED TO NARROW POSSIBLE SOLUTIONS

    fail = 0;
    k = A(start).llist;

    for j=1:nPivots

        % If combinations reduce to 0, search has failed
        % If combinations reduce to 1, search is complete

%         WaitForKeyPress;    % Use to stop at every pivot

        switch nFinalists
            case 0
                %                     '-- FAILED SEARCH --'
                %                     fail = 1;
                nPivots = j - 1;
                break;
            case 1
                nPivots = j - 1;
                break;
        end

        % Plot in FOV as desired (bit 3 of gmode set)

        if bitand( gmode, 4) == 4
            PlotInFOV( sv1, sv2, FOV, 0, '-b' );
            PlotInFOV( sv2, sv1, FOV, 0, '-b' );
            sv1 = StarsInFOV( A(k).Stars(1) ).mv;
            sv2 = StarsInFOV( A(k).Stars(2) ).mv;
            PlotInFOV( sv1, sv2, FOV, 0, '-g' );
            PlotInFOV( sv2, sv1, FOV, 0, '-g' );
        end

        [ j (A(k).imax - A(k).imin) ]
        Results.nFinalists = [ Results.nFinalists (A(k).imax - A(k).imin) ];

        % Compare current finalists to possible stars that match
        % current triangle. Keep those triangles that share two stars.

        % Put stars in to binary tree
```

```
saTree = [];
sbTree = [];

for b = A(k).imin:A(k).imax
    anum = AngPtr(b);

    saTree = BTreeAdd( Ang(anum).Stars(1), anum, saTree );
    sbTree = BTreeAdd( Ang(anum).Stars(2), anum, sbTree );

end

n1 = 0;
F1 = [];
for a = 1:nFinalists    % Go through current finalists
    s1 = Finalists(a).Stars(1,1);
    s2 = Finalists(a).Stars(1,2);

    match = [];

    p = BTreeFind( s1, saTree );
    for b = 1:size(p,2)
        c = saTree( p(b) ).Data;
        match = [c match];
    end

    p = BTreeFind( s2, saTree );
    for b = 1:size(p,2)
        c = saTree( p(b) ).Data;
        match = [c match];
    end

    p = BTreeFind( s1, sbTree );
    for b = 1:size(p,2)
        c = saTree( p(b) ).Data;
        match = [c match];
    end

    p = BTreeFind( s2, sbTree );
    for b = 1:size(p,2)
        c = saTree( p(b) ).Data;
```

```
            match = [c match];
        end

            for b=1:size(match,2)
                n1 = n1 + 1;
                F1(n1).Ang   = [ match(b); Finalists(a).Ang ];
                F1(n1).Stars = [ Ang( match(b) ).Stars; Finalists(a).Stars ];
            end

            if n1 > nFinalists
                nPivots = j - 1;
                break;
            end

        if n1 > nFinalists
            break;
        end

    end

    if n1 > nFinalists
        break;
    end

    Finalists = F1;     % New list (F1) becomes current finalists
    nFinalists = n1

    k = A(k).llist;     % Advance to next combination in linked list
end

% COMPILE RESULTS

Results.nPivots = nPivots;

switch nFinalists
    case 0
        '-- NO SOLUTION --'
        Results.Match = [];
        nResults = 0;
    case 1
        Results.Match = Finalists(1).Stars(1,1:2);
```

```
            n1 = 2;
            for j=2:nPivots+1
                for k = 1:2
                    match = false;
                    for m=1:n1
                        if Finalists(1).Stars(j,k) == Results.Match(m)
                            match = true;
                            break;
                        end
                    end
                    if match == false
                        Results.Match = [ Results.Match Finalists(1).Stars(j,k) ];
                        n1 = n1 + 1;
                    end
                end
            end
            nResults = n1;
        otherwise
            '-- INCONCLUSIVE RESULTS --'
            Results.Match = [];
            nResults = 0;
    end
end
```

E.35. MatchStarsWSphTris.m

```
% ========================================================================
%
% MatchStarsWSphTris.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Takes a list of stars in FOV, and tries to match up stars in catalog
% using spherical triangle method.
%
% INPUTS:   StarsInFOV - Array of stars in field of view & measured vectors
%           climit - possible solution limit
%           plimit - pivot limit
%           sigm - standard deviation of measurement error
%           sigx - sigma bound for angle recognition (recommend 3)
%
% OUTPUT:   Results - structure with result
%
% SUBROUTINES REQUIRED: PlotInFOV.m
%                       BTreeAdd.m
%                       BTreeFind.m
%                       CalcFOV.m
%
% ========================================================================

function Results = MatchStarsWSphTris( StarsInFOV, climit, plimit, sigm, sigx );

global gmode Tri nTri TriPtr Kvec FOV FOVmax

nStarsInFOV = size( StarsInFOV, 2 );

% IF LESS THAN THREE STARS, TRIANGLES CANNOT BE MADE

if nStarsInFOV < 3
    nCombs = 0;
    Combs = [];
    Results.nPivots = 0;
    Results.nFinalists = [];
```

```
    Results.Match = [];
    '-- NOT ENOUGH STARS --'
else
    nCombs = nchoosek( nStarsInFOV, 3 )

    % CREATE LIST OF TRIANGLES, INCL. AREA, AND MAX & MIN TRIANGLE INDICIES

    clear T;
    k = 0;
    start = 1;
    maxProp = 0;

    % Create array of Triangles in FOV, including range of possible
    % solutions from Triangle Catalog using K-Vector

    for s1=1:nStarsInFOV-2
        sv1 = StarsInFOV(s1).mv;
        for s2=s1+1:nStarsInFOV-1
            sv2 = StarsInFOV(s2).mv;

            if acos( dot( sv1, sv2 ) ) <= FOVmax

                for s3=s2+1:nStarsInFOV
                    sv3 = StarsInFOV(s3).mv;

                    mFOV = CalcFOV( sv1, sv2, sv3 );  % Determine if skinny
                    if mFOV <= FOVmax

                        % Measure area of triangle, determine bounds on
                        % error

                        Prop = SphTriArea( sv1, sv2, sv3 );
                        var  = StarAreaCov( sv1, sv2, sv3, sigm );

                        % Calculate minimum and maximum areas

                        PropMin = Prop - sigx * sqrt(var);
                        PropMax = Prop + sigx * sqrt(var);

                        % Determine ranges of possible solutions using K-Vector
```

```
                    pmin = FindWithParabKvec( PropMin, Kvec );
                    pmax = FindWithParabKvec( PropMax, Kvec );
                    range = pmax - pmin;

                    if range <= climit      % Only keep tri list within combo limit
                        k = k + 1;
                        T(k).Stars = [ s1 s2 s3 ];
                        T(k).Prop  = Prop;
                        T(k).Prop2 = -1;    % Don't fill till needed
                        T(k).pmin  = pmin;
                        T(k).pmax  = pmax;
                        T(k).llist = 0;     % Used for linked list

                        % Find Triangle with greatest area to start list

                        if T(k).Prop > maxProp
                            start = k;
                            maxProp = T(k).Prop;
                        end
                    end
                end
            end
        end
    end
end
nCombs = k

% CREATE LINKED LIST OF TRIANGLES IN FOV, SUCH THAT NO MORE THAN
% ONE STAR CHANGES AT A TIME GIVING PRIORITY TO TRIANGLES WITH
% LARGEST POSSIBLE AREASS

T(start).llist = 999;       % makes ineligible (also EOL)
prev = start;
next = start;
nPivots = 0;
for j=1:nCombs
    maxProp = 0;
    done = false;

    for k=1:nCombs
        if T(k).llist == 0
```

```
            x = 0;
            for m=1:3
                switch T(k).Stars(m)
                    case T(prev).Stars(1)
                        x = x + 1;
                    case T(prev).Stars(2)
                        x = x + 1;
                    case T(prev).Stars(3)
                        x = x + 1;
                end
            end
            if x == 2       % At least two common pts
                if T(k).Prop > maxProp
                    next = k;
                    maxProp = T(k).Prop;
                end
                done = true;
            end
        end
    end

    T(prev).llist = next;
    T(next).llist = 999;    % EOL
    prev = next;

    if done == true
        nPivots = nPivots + 1;
        if nPivots == plimit
            break;
        end
    else
        break;
    end
end

nPivots = nPivots

% Start with triangle with highest property value

if nCombs > 0
    nFinalists = T(start).pmax - T(start).pmin
```

```
k = 0;

% Get Ic of first triangle, find range of allowable Ic

s1  = T(start).Stars(1);
s2  = T(start).Stars(2);
s3  = T(start).Stars(3);
sv1 = StarsInFOV(s1).mv;
sv2 = StarsInFOV(s2).mv;
sv3 = StarsInFOV(s3).mv;
T(start).Prop2 = SphTriPolarMoment( sv1, sv2, sv3, 3, 0, 0 );
var = 3*((7.14e-4)*T(start).Prop2 + 3e-9);
Prop2min = T(start).Prop2 - var;
Prop2max = T(start).Prop2 + var;

for j=1:nFinalists
    tnum = TriPtr( T(start).pmin+j-1 );

    % Include only if Ic is within allowable range

    if Tri(tnum).Ip >= Prop2min
        if Tri(tnum).Ip <= Prop2max
            k = k + 1;
            Finalists(k).Tri   = tnum;
            Finalists(k).Stars = Tri(tnum).Stars;
        end
    end
end
nFinalists = k
Results.nFinalists = k;

if bitand( gmode, 4 ) == 4
    sv1 = StarsInFOV( T(start).Stars(1) ).mv;
    sv2 = StarsInFOV( T(start).Stars(2) ).mv;
    sv3 = StarsInFOV( T(start).Stars(3) ).mv;
    PlotInFOV( sv1, sv2, FOV, 0, '-g' );
    PlotInFOV( sv2, sv3, FOV, 0, '-g' );
    PlotInFOV( sv3, sv1, FOV, 0, '-g' );
    PlotInFOV( sv1, sv2, FOV, 0, '-g' );
end
```

```
    else
        nFinalists = 0;
        Results.nFinalists = 0;
    end

    % ============================================
    % PIVOT AS REQUIRED TO NARROW POSSIBLE SOLUTIONS
    % ============================================

    fail = 0;
    k = T(start).llist;

    for j=1:nPivots

        % If number of finalists reduces to 0, search has failed
        % If number of finalists reduces to 1, search is complete

%         WaitForKeyPress;  % Use to stop at every pivot

        switch nFinalists
            case 0
                nPivots = j - 1
                break;
            case 1
                nPivots = j - 1
                break;
        end

        % Plot in FOV as desired (bit 3 of gmode set)

        if bitand( gmode, 4) == 4
            PlotInFOV( sv1, sv2, FOV, 0, '-b' );
            PlotInFOV( sv2, sv3, FOV, 0, '-b' );
            PlotInFOV( sv3, sv1, FOV, 0, '-b' );
            PlotInFOV( sv1, sv2, FOV, 0, '-b' );
            sv1 = StarsInFOV( T(k).Stars(1) ).mv;
            sv2 = StarsInFOV( T(k).Stars(2) ).mv;
            sv3 = StarsInFOV( T(k).Stars(3) ).mv;
            PlotInFOV( sv1, sv2, FOV, 0, '-g' );
            PlotInFOV( sv2, sv3, FOV, 0, '-g' );
            PlotInFOV( sv3, sv1, FOV, 0, '-g' );
```

```
    PlotInFOV( sv1, sv2, FOV, 0, '-g' );
end

[ j (T(k).pmax - T(k).pmin) ]
Results.nFinalists = [ Results.nFinalists (T(k).pmax - T(k).pmin) ];

% Create three binary trees of possible triangles,
% sorted by first star, second start and third star

saTree = [];
sbTree = [];
scTree = [];

s1  = T(k).Stars(1);
s2  = T(k).Stars(2);
s3  = T(k).Stars(3);
sv1 = StarsInFOV(s1).mv;
sv2 = StarsInFOV(s2).mv;
sv3 = StarsInFOV(s3).mv;
T(k).Prop2 = SphTriPolarMoment( sv1, sv2, sv3, 3, 0, 0 );
var = 3*((7.14e-4)*T(start).Prop2 + 3e-9);
Prop2min = T(k).Prop2 - var;
Prop2max = T(k).Prop2 + var;

for b = T(k).pmin:T(k).pmax
    tnum = TriPtr(b);

    % Add only those with correct range of Ic

    if Tri(tnum).Ip >= Prop2min
        if Tri(tnum).Ip <= Prop2max
            saTree = BTreeAdd( Tri( tnum ).Stars(1), tnum, saTree );
            sbTree = BTreeAdd( Tri( tnum ).Stars(2), tnum, sbTree );
            scTree = BTreeAdd( Tri( tnum ).Stars(3), tnum, scTree );
        end
    end
end

F1 = [];    % Reset list of finalists
n1 = 0;
```

```
for a = 1:nFinalists
    s1 = Finalists(a).Stars(1,1);
    s2 = Finalists(a).Stars(1,2);
    s3 = Finalists(a).Stars(1,3);

    match = [];

    % Find triangles that have s1 in 1st position.
    % Of those, if s2 is in 2nd or 3rd position it's a match.
    % Or, if s3 is in 2nd or 3rd position it's a match.

    p = BTreeFind( s1, saTree );
    for b = 1:size(p,2)
        c = saTree( p(b) ).Data;
        switch s2
            case Tri( c ).Stars(2)
                match = [ c match ];
            case Tri( c ).Stars(3)
                match = [ c match ];
            otherwise
                switch s3
                    case Tri( c ).Stars(2)
                        match = [ c match ];
                    case Tri( c ).Stars(3)
                        match = [ c match ];
                end
        end
    end

    % Find triangles that have s1 in 2nd position.
    % Of those, if s2 is in 1st or 3rd position it's a match.
    % Or, if s3 is in 1st or 3rd position it's a match.

    p = BTreeFind( s1, sbTree );
    for b = 1:size(p,2)
        c = sbTree( p(b) ).Data;
        switch s2
            case Tri( c ).Stars(1)
                match = [ c match ];
            case Tri( c ).Stars(3)
                match = [ c match ];
```

```
        otherwise
            switch s3
                case Tri( c ).Stars(1)
                    match = [ c match ];
                case Tri( c ).Stars(3)
                    match = [ c match ];
            end
    end
end

% Find triangles that have s1 in 3rd position.
% Of those, if s2 is in 1st or 2nd position it's a match.
% Or, if s3 is in 1st or 2nd position it's a match.

p = BTreeFind( s1, scTree );
for b = 1:size(p,2)
    c = scTree( p(b) ).Data;
    switch s2
        case Tri( c ).Stars(1)
            match = [ c match ];
        case Tri( c ).Stars(2)
            match = [ c match ];
        otherwise
            switch s3
                case Tri( c ).Stars(1)
                    match = [ c match ];
                case Tri( c ).Stars(2)
                    match = [ c match ];
            end
    end
end

% Find triangles that have s2 in 1st position.
% Of those, if s3 is in 2nd or 3rd position it's a match.

p = BTreeFind( s2, saTree );
for b = 1:size(p,2)
    c = saTree( p(b) ).Data;
    switch s3
        case Tri( c ).Stars(2)
            match = [ c match ];
```

```
        case Tri( c ).Stars(3)
            match = [ c match ];
    end
end

% Find triangles that have s2 in 2nd position.
% Of those, if s3 is in 1st or 3rd position it's a match.

p = BTreeFind( s2, sbTree );
for b = 1:size(p,2)
    c = sbTree( p(b) ).Data;
    switch s3
        case Tri( c ).Stars(1)
            match = [ c match ];
        case Tri( c ).Stars(3)
            match = [ c match ];
    end
end

% Find triangles that have s2 in 3rd position.
% Of those, if s2 is in 1st or 2nd position it's a match.

p = BTreeFind( s2, scTree );
for b = 1:size(p,2)
    c = scTree( p(b) ).Data;
    switch s3
        case Tri( c ).Stars(1)
            match = [ c match ];
        case Tri( c ).Stars(2)
            match = [ c match ];
    end
end

for b=1:size(match,2)
    n1 = n1 + 1;
    F1(n1).Tri   = [ match(b); Finalists(a).Tri ];
    F1(n1).Stars = [ Tri( match(b) ).Stars; Finalists(a).Stars ];
end

if n1 > nFinalists
    nPivots = j - 1;
```

```
            break;
        end

    end

    % If no of finalists increases from previous round, abandon matching

    if n1 > nFinalists
        break;
    end

    % Newly created list (F1) becomes current finalists

    Finalists = F1;
    nFinalists = n1

    % Advance to next combination in linked list

    k = T(k).llist;
end

% COMPILE RESULTS

Results.nPivots = nPivots;

switch nFinalists
    case 0      % Search failed to match triangle
        '-- NO SOLUTION --'
        Results.Match = [];
        nResults = 0;
    case 1      % Search successful, create array of stars in FOV
        Results.Match = Finalists(1).Stars(1,1:3);
        n1 = 3;
        for j=2:nPivots+1
            for k = 1:3
                match = false;
                for m=1:n1
                    if Finalists(1).Stars(j,k) == Results.Match(m)
                        match = true;
                        break;
                    end
```

```
                    end
                    if match == false
                        Results.Match = [ Results.Match Finalists(1).Stars(j,k) ];
                        n1 = n1 + 1;
                    end
                end
            end
            nResults = n1;
        otherwise   % Unable to reduce possible solutions to one
            '-- INCONCLUSIVE RESULTS --'
            Results.Match = [];
            nResults = 0;
    end
end
```

E.36. PlotArc.m

```
% ========================================================================
%
% PlotArc.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Plots node in current figure, using arcs
%
% INPUTS:   p1  unit vector to starting point
%           p2  unit vector to ending point
%           c   color
%
% OUTPUT:   none
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function PlotArc( p1, p2, c );

% Find normal to plane created by vectors P1 and P2

P = cross( p1, p2 );
beta = asin( P(3)/ sqrt(P(1)^2+P(2)^2+P(3)^2) );
alpha = atan2( P(2),P(1) );

% First rotate about z-axis, then y-axis to get plane by vectors coplaner
% to x-y axis.

Rz1 = [ cos(alpha) -sin(alpha) 0; sin(alpha) cos(alpha) 0; 0 0 1];
Ry = [ cos(beta-pi/2) 0 -sin(beta-pi/2); 0 1 0; sin(beta-pi/2) 0 cos(beta-pi/2)];

p1a = Ry^-1 * Rz1^-1 * p1;
p2a = Ry^-1 * Rz1^-1 * p2;

% Rotate about z-axis again to get pt 1 to x-axis
```

```
theta1 = atan2( p1a(2), p1a(1) );

Rz2 = [ cos(theta1) -sin(theta1) 0; sin(theta1) cos(theta1) 0; 0 0 1];

p1b = Rz2^-1 * p1a;
p2b = Rz2^-1 * p2a;

% Create arc from pt 1 to pt 2 on xy plane

theta2 = atan2( p2b(2), p2b(1) );

% if theta2>0               % Use for fixed arc per step
%     step = 1 * pi/180;
% else
%     step = -1 * pi/180;
% end

step = theta2 / 10;         % Use for fixed steps per arc

npt = 0;
for i=0:step:theta2
    npt = npt + 1;
    ptb(1,npt) = 1*cos(i);
    ptb(2,npt) = 1*sin(i);
    ptb(3,npt) = 0;
end

% Rotate arc back to vectors' original orientation

for i=1:npt
    pt(:,i) = Rz1 * Ry * Rz2 * ptb(:,i);
end

% Plot

plot3( pt(1,:), pt(2,:), pt(3,:), c );
```

E.37. PlotInFOV.m

```
% ========================================================================
%
% PlotInFOV.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Shows what star tracker sees in 2-D. Draws line from P1 to P2 in
% star tracker frame. To draw star, set P2 to zero, and
%
% INPUTS:   P1 - vector to point 1
%           P2 - vector to point 2
%           FOV - Star Tracker field of view (rad)
%           clr - clear FOV if 1
%           c - color and style of line
%
% OUTPUT:   none
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function PlotInFOV( P1, P2, FOV, clr, c )

% Plane will pass through pt P and, and be perpedicular to vector P
% Equation of plane: Ax + By + Cz = D

% Determine equation of plane that will be perpendicular to vector P
% and pass through pt P

P = [1 0 0]';  % Center of FOV in star tracker frame

if clr == 1

    figure(2);
    hold off;
    plot( 0, 0,'+b');
    axis( [ -FOV/2*180/pi FOV/2*180/pi -FOV/2*180/pi FOV/2*180/pi ] );
```

```
    hold on;

    i = 1;
    for j=0:2*pi/30:2*pi
        x(i) = FOV/2 * 180/pi * cos( j );
        y(i) = FOV/2 * 180/pi * sin( j );
        i = i + 1;
    end
    plot( x, y, '-b' );

else

    A = P(1);
    B = P(2);
    C = P(3);
    D = A^2 + B^2 + C^2;

    % Find where vector for star (P1) intersects plane

    t = D / ( P(1) * P1(1) + P(2) * P1(2) + P(3) * P1(3) );
    P1a = [ P1(1) * t P1(2) * t P1(3) * t ]';

    % Rotate plane about z-axis, then y-axis so its parallel to x-y plane

    mag = sqrt( P(1)^2 + P(2)^2 + P(3)^2 );
    beta = pi/2 - asin( P(3) / mag );
    alpha = atan2( P(2), P(1) );

    Ry = [ cos(-beta) 0 -sin(-beta); 0 1 0; sin(-beta) 0 cos(-beta)];
    Rz = [ cos(alpha) -sin(alpha) 0; sin(alpha) cos(alpha) 0; 0 0 1];

    % P = Ry^-1 * Rz^-1 * P         % Check: should be [0 0 1]'

    P1xy = Ry^-1 * Rz^-1 * P1a;      % Results in [x y 1]'

    % Second point

    % Find where vector for star (P2) intersects plane

    t = D / ( P(1) * P2(1) + P(2) * P2(2) + P(3) * P2(3) );
    P2a = [ P2(1) * t P2(2) * t P2(3) * t ]';
```

```
    % Rotate plane about z-axis, then y-axis so its parallel to x-y plane

    P2xy = Ry^-1 * Rz^-1 * P2a;      % Results in [x y 1]'

    % Plot

    x = [ P1xy(1) P2xy(1) ] .* 180/pi;
    y = [ P1xy(2) P2xy(2) ] .* 180/pi;

    figure(2)
    plot( x, y, c );

end
```

E.38. PlotNode.m

```
% ========================================================================
%
% PlotNode.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Plots node in current figure, using arcs
%
% INPUTS:   n - Node Number
%           c - color
%
% OUTPUT:   none
%
% SUBROUTINES REQUIRED: PlotArc.m
%
% ========================================================================

function PlotNode( n, c );

global Vertex Node

v1 = Vertex( Node(n).Vertex(1) ).Vector;
v2 = Vertex( Node(n).Vertex(2) ).Vector;
v3 = Vertex( Node(n).Vertex(3) ).Vector;

PlotArc( v1, v2, c);
hold on;
PlotArc( v2, v3, c);
PlotArc( v3, v1, c);
```

E.39. PlotSphericalCap.m

```
% ========================================================================
%
% PlotSphericalCap.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Plots a circle centered at point v1 that has a cap angle of gamma.
%
% INPUTS:   v - vector to center of cap
%           gamma - cap angle
%           c - color of cap
%
% OUTPUT:   none
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function PlotSphericalCap( v, gamma, c )

x = v(1);
y = v(2);
z = v(3);

beta =  pi/2 - acos( z/norm(v) );
alpha = atan2( y, x );

% Plot a sphere on x-y plane

R = sin(gamma/2);
X = cos(gamma/2);

nsegs = 20;

% Create circle on y-z plane, out on x-axis

n = 0;
```

```
for theta=0:2*pi/nsegs:2*pi
    n = n + 1;
    Pt(1,n) = X;
    Pt(2,n) = R*cos(theta);
    Pt(3,n) = R*sin(theta);
end

% Rotate to Vector

Ry = [ cos(-beta) 0 -sin(-beta); 0 1 0; sin(-beta) 0 cos(-beta)];
Rz = [ cos(-alpha) -sin(-alpha) 0; sin(-alpha) cos(-alpha) 0; 0 0 1];

for i=1:n
    Pt(:,i) = Rz^-1 * Ry^-1 * Pt(:,i);
end

plot3( Pt(1,:), Pt(2,:), Pt(3,:), c );
```

E.40. PlotSphericalTri.m

```
% ========================================================================
%
% PlotSphericalTri.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Plots spherical triangle given unit vectors poining to vertices
%
% INPUTS:   v1 - vector to first vertex
%           v2 - vector to second vertex
%           v3 - vector to third vertex
%           c - color
%
% OUTPUT:   none
%
% SUBROUTINES REQUIRED: PlotSphericalArc.m
%
% ========================================================================

function PlotSphericalTri( v1, v2, v3, c )

PlotArc( v1, v2, c);
hold on;
PlotArc( v2, v3, c);
PlotArc( v3, v1, c);
```

E.41. RandAttTest.m

```
% ========================================================================
%
% RandAttTest.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Creates random attitude, determines stars in FOV, performs coordinate
% transform to put star positions into star tracker frame, adds error
% then tests either angle or spherical triangle method. Results are
% tabulated then saved.
%
% INPUTS:   Stars - star list
%           SphTri2xxxx - Catalog of Spherical Triangles w/area and Ip
%           SphTriPtr - Pointer array and K-Vec for Sph Tri Catalog
%           Angxxxx - Catalog of Angles
%           AngPtr - Pointer array and K-Vec for Angle Catalog
%
% OUTPUT:   Results - structure containing results of all tests
%
% SUBROUTINES REQUIRED: MatchStarsWAngs.m
%                       MatchStarsWSphTris.m
%                       PlotInFOV.m
%                       GetRandomVector.m
%                       GetBodyFrame.m
%                       PlotSphericalCap.m
%                       FindStarsInFOV.m
%                       CoordXform.m
%                       AddNoise.m
%                       WaitForKeyPress.m
%
% ========================================================================

clear all;

global nVertex nNodes Star nStars gmode
global Tri nTri TriPtr Ang nAng AngPtr Kvec
global trvector FOV FOVmax
```

```
Method = 2;             % Set to 1 for Angle Method, 2 for SphTri Method

ntests = 1000           % Number of tests

gmode = 7;              % Bit 1 = 3-D View of FOV
                        % Bit 2 = Star Tracker FOV
                        % Bit 3 = Wait between tests

FOV = 8 * pi/180;       % Star Tracker field of view
sigm = (87.2665e-6)/3;  % Measurement Standard Deviation
sigx = 3;               % Sigma bound for search
climit = 10000;         % Limit to number of combinations to pursue

load Stars;

switch Method
    case 1
        'ANGLE METHOD'
        load AngM60L4
        load AngPtrM60L4
        plimit = 9      % Pivot Limit
    case 2
        'SPHERICAL TRIANGLE METHOD'
        load SphTri2M60L4
        load SphTriPtrM60L4
        plimit = 3      % Pivot Limit
end

%
% LOOP THROUGH RANDOM ATTITUDES
%

for i = 1:ntests;
    i

    % Set-up 3-d plot if graphics are desired

    if bitand( gmode,1 ) == 1
        figure(1);
        hold off;
```

```
    plot3( 0,0,0 ,'oy','MarkerSize',2);
    hold on;
    grid on;
end

% GET RANDOM TRACKER DIRECTION & ATTITUDE

trvector = GetRandomVector;
bframe   = GetBodyFrame( trvector );

Results(i).trvector = trvector;

if bitand( gmode,1 ) == 1
    PlotSphericalCap( trvector, FOV, 'b' );
    plot3( [0 bframe.b1(1)], [0 bframe.b1(2)], [0 bframe.b1(3)],'b');
    plot3( [0 bframe.b2(1)*0.5], [0 bframe.b2(2)*0.5], [0 bframe.b2(3)*0.5],'b');
    plot3( [0 bframe.b3(1)*0.5], [0 bframe.b3(2)*0.5], [0 bframe.b3(3)*0.5],'b');
end

% FIND STARS WITHIN FIELD OF VIEW

StarsInFOV = FindStarsInFOV( trvector, FOV );
nStarsInFOV = size( [ StarsInFOV.Star ], 2 );
[nStarsInFOV]

% ADD MEASUREMENT ERROR TO STAR VECTORS

for j=1:nStarsInFOV
    vi = Star( StarsInFOV(j).Star ).Vector;  % True vector
    vb = CoordXform( vi, bframe ) ;          % Convert to tracker frame
    vm = AddNoise( vb, sigm );               % Add noise (measured vector)

    StarsInFOV(j).tv = vb;
    StarsInFOV(j).mv = vm;
end

% Plot stars if desired

if bitand( gmode, 2 ) == 2
    PlotInFOV( [0 0 0]', [0 0 0]', FOV, 1, 'm' );
    for j = 1:nStarsInFOV
```

```
        PlotInFOV( StarsInFOV(j).tv, StarsInFOV(j).tv, FOV, 0, 'ob' );
        PlotInFOV( StarsInFOV(j).mv, StarsInFOV(j).mv, FOV, 0, '+r' );
    end
end

% MATCH UP STARS IN FOV USING SPHERICAL TRIANGLES

t0 = cputime;
switch Method
    case 1
        R1 = MatchStarsWAngs( StarsInFOV, climit, plimit, sigm, sigx );
    case 2
        R1 = MatchStarsWSphTris( StarsInFOV, climit, plimit, sigm, sigx );
end
t1 = cputime - t0

Results(i).nPivots = R1.nPivots;
Results(i).nFinalists = R1.nFinalists;
Results(i).Match = R1.Match;
Results(i).StarsInFOV  = [ StarsInFOV.Star ];
Results(i).time = t1;
nResults = size( Results(i).Match, 2 );

% IF UNABLE TO MATCH, RECORD FAILURE

Results(i).Fail = false;
Results(i).Cant = false;
if nStarsInFOV > Method
    if nResults == 0
        Results(i).Fail = true;
    end
else
    Results(i).Cant= true;
end

% CHECK THAT RESULTS ARE CORRECT. IF NOT, RECORD ERROR.

Results(i).Bad = false;
Results(i).Correct = false;
if nResults > 0
    for j=1:nResults
```

```
            found = 0;
            for k=1:nStarsInFOV
                if Results(i).Match(j) == StarsInFOV(k).Star
                    found = 1;
                    break;
                end
            end
            if found == 0
                break;
            end
        end
        if found == 0
            '-- INCORRECT RESULTS --'
            Results(i).Bad = true;
        else
            '** CORRECT RESULTS **'
            Results(i).Correct = true;
        end
    end

    Results(i)

    if bitand( gmode, 4 ) == 4
        WaitForKeyPress;
    end

end

save Results Results
```

E.42. SearchDist.m

```
% ========================================================================
%
% SearchDist.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Determines minimum height (arc) of a spherical triangle
% Works if height is less or equal to 90 degrees.
% If height is greater subtract results from pi
%
% INPUTS:   v1 - vector no. 1 to vertex of spherical triangle
%           v2 - vector no. 2 to vertex of spherical triangle
%           v3 - vector no. 3 to vertex of spherical triangle
%
% OUTPUT:   d - minimum height of triangle (rads)
%
% SUBROUTINES REQUIRED: PlotArc.m
%
% ========================================================================

function d = SearchDist( v1, v2, v3 )

global gmode

a = acos( dot(v1,v2) );
b = acos( dot(v2,v3) );
c = acos( dot(v3,v1) );

% Determine longest side (shortest search distance)

if a > b
    if a > c
        % long side is a
        r1 = v1;
        r2 = v2;
        r3 = v3;
    else
```

```
        % long side is c
        r1 = v3;
        r2 = v1;
        r3 = v2;
    end
else
    if b > c
        % long side is b
        r1 = v2;
        r2 = v3;
        r3 = v1;
    else
        % long side is c
        r1 = v3;
        r2 = v1;
        r3 = v2;
    end
end

% Find normal to plane formed by vectors to long side

n1 = cross(r1,r2);
n1 = n1 / norm(n1);

% Find norm to plane perp to plane formed by vectors
% to long side and passes through r3.

n2 = cross(n1,r3);

if norm(n2) ~= 0

    % If cross product is not zero plane exists

    n2 = n2 / norm(n2);

    % Find vector formed where plane formed by vectors to
    % long side intersects perp plane

    r4 = cross( n2, n1 );
    r4 = r4 / norm( r4 );
```

```
else

    % If cross product is zero, r3 is rt angle to both
    % r1 and r2. r4 can be any pt along arc between r1
    % and r2. Mid pt chosen so arc r3 to r4 is easy to see.

    r4 = ArcMidPt( r1, r2 );

end

% plot3( [0 r4(1)], [0 r4(2)], [0 r4(3)], 'b' );

if gmode == 2
    PlotArc( r3, r4, 'b' );
end

% Distance is equal to arc between r3 and r4

d = acos( dot( r4, r3 ) );
```

E.43. SortAngles.m

```
% ========================================================================
%
% SortAngles.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Creates Pointer Array and K-Vector for Unosrted Catalog of Angles
% for Angle Method Use
%
% INPUTS:   Angxxxx - Unsorted Catalog Of Angles
%
% OUTPUT:   AngPtrxxxx - Pointer Array of Angles, sorted by angle
%
% SUBROUTINES REQUIRED: BigBTreeSort.m
%                       CreateLinKvector.m
%
% ========================================================================

clear all;

load AngM60L4A

global TreeVal TreeLeft TreeRight

nAng = 10000
Ang = Ang(1:nAng);

% Preallocate memory for everything

'Tree Allocation'
TreeVal   = zeros( 1, nAng );
TreeLeft  = zeros( 1, nAng );
TreeRight = zeros( 1, nAng );

'TriPtr Allocation'
AngPtr    = zeros( 1, nAng );
```

```
'Kvec Allocation'
DotPrs    = zeros( 1, nAng );
Kvec.Ptr  = zeros( 1, nAng );

% Index Stars by Spherical Area

'Sort'

AngPtr = BigBTreeSort( [ Ang.dotpr ] );

% Create linear Kvector of Angles

'K-Vector'

for i=1:nAng
    if i/1000 == floor(i/1000)
        [i nAng]
    end

    DotPrs(i) = Ang( AngPtr(i) ).dotpr;
end

Kvec = CreateLinKvector( DotPrs )

save AngPtrM60L4 AngPtr Kvec
```

E.44. SortSphTris.m

```
% ========================================================================
%
% SortSphTris.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Creates Pointer Array and K-Vector for Unosrted Catalog of Angles
% for Angle Method Use
%
% INPUTS:   SphTrixxxx - Unsorted Catalog Of Spherical Triangles
%
% OUTPUT:   SphTriPtrxxxx - Pointer Array of Sph. Tris, sorted by area
%
% SUBROUTINES REQUIRED: BigBTreeSort.m
%                       CreateParabKvector.m
%
% ========================================================================

clear all;

load SphTri2M60L4

global TreeVal TreeLeft TreeRight

nTri = 10000
Tri = Tri(1:nTri);

% Preallocate memory for everything

'Tree Allocation'
TreeVal   = zeros( 1, nTri );
TreeLeft  = zeros( 1, nTri );
TreeRight = zeros( 1, nTri );

'TriPtr Allocation'
TriPtr    = zeros( 1, nTri );
```

```
'Kvec Allocation'
SphArea   = zeros( 1, nTri );
Kvec.Ptr  = zeros( 1, nTri );

% Index Stars by Spherical Area

'Sort'

TriPtr = BigBTreeSort( [ Tri.Area ] );

% Create Parabolic Kvector of Stars

'K-Vector'

for i=1:nTri
    if i/1000 == floor(i/1000)
        [i nTri]
    end

    SphArea(i) = Tri( TriPtr(i) ).Area;
end

Kvec = CreateParabKvector( SphArea )

save SphTriPtrM60L4 TriPtr Kvec
```

E.45. SphTriArea.m

```
% ========================================================================
%
% SphTriArea.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Calculates the area of a spherical triangle with unit radius
%
% INPUTS:   v1 - unit vector to vertex 1
%           v2 - unit vector to vertex 2
%           v3 - unit vector to vertex 3
%
% OUTPUT:   A - area
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function A = SphTriArea( v1, v2, v3 );

% Assumes points are on a sphere, with a radius of 1

a = acos( dot( v1, v2 ) / 1 );
b = acos( dot( v2, v3 ) / 1 );
c = acos( dot( v3, v1 ) / 1 );
s = (a+b+c)/2;

A = 4*atan( sqrt( tan(s/2)*tan((s-a)/2)*tan((s-b)/2)*tan((s-c)/2) ) );
```

E.46. SphTriCentroid.m

```
% ========================================================================
%
% SphTriCentroid.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Calculates the centroid of a spherical triangle with unit radius
%
% INPUTS:   v1 - unit vector to vertex 1
%           v2 - unit vector to vertex 2
%           v3 - unit vector to vertex 3
%
% OUTPUT:   c - unit vector to centroid
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function c = SphTriCentroid( v1, v2, v3 )

c = v1 + v2 + v3;
c = c ./ 3;
c = c / norm(c);
```

E.47. SphTriPolarMoment.m

```
% ========================================================================
%
% SphTriPolarMoment.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Calculates the polar moment of a spherical triangle unit
%
% INPUTS:   v1 - unit vector to vertex 1
%           v2 - unit vector to vertex 2
%           v3 - unit vector to vertex 3
%           MaxLevel - level of recursion (recommend 3 minimum)
%           gc - global centroid (ignored on entry)
%           level - current level of recursion (set to 0 when calling)
%
% OUTPUT:   Ip - Polar Moment of Spherical Triangle
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function Ip = SphTriPolarMoment( v1, v2, v3, MaxLevel, gc, level )

gmode = 0;

% Find global centroid at first pass

if level == 0
    gc = SphTriCentroid( v1, v2, v3 );
    if gmode ~= 0
        hold off;
        figure(10);
    end
end

if gmode ~= 0
    PlotSphericalTri( v1, v2, v3, 'r' );
```

```
end

if level < MaxLevel

    % Find mid point of edges of triangle

    m12 = ArcMidPt( v1, v2 );
    m23 = ArcMidPt( v2, v3 );
    m31 = ArcMidPt( v3, v1 );

    % Further divide triangle

    Ip = SphTriPolarMoment( v1, m12, m31, MaxLevel, gc, level+1 );
    Ip = Ip + SphTriPolarMoment( v2, m12, m23, MaxLevel, gc, level+1 );
    Ip = Ip + SphTriPolarMoment( v3, m23, m31, MaxLevel, gc, level+1 );
    Ip = Ip + SphTriPolarMoment( m12, m23, m31, MaxLevel, gc, level+1 );

else

    % Calculate polar moment about global centroid

    lc = SphTriCentroid( v1, v2, v3 );  % local centroid
    if gmode ~= 0
        PlotArc( lc, gc, 'g' );
    end

    dA = SphTriArea( v1, v2, v3 );      % area of element
    theta = acos( dot( lc, gc ) );            % arc distance to centroid
    Ip = theta^2 * dA;

end
```

E.48. StarAreaCov.m

```
% ========================================================================
%
% StarAreaCov.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Determines variance of spherical triangle angle measurement.
%
% INPUTS:   b1 - vector to vertex 1
%           b2 - vector to vertex 2
%           b3 - vector to vertex 3
%           sigm - standard deviation of measurement
%
% OUTPUT:   p_area - variance of area measurement
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function p_area = StarAreaCov( b1, b2, b3, sigm )

a=acos(b1'*b2);
b=acos(b2'*b3);
c=acos(b1'*b3);
s=0.5*(a+b+c);
area=4*atan(sqrt(tan(s/2)*tan((s-a)/2)*tan((s-b)/2)*tan((s-c)/2)));

sm=s;
am=a;
bm=b;
cm=c;

z2=tan(sm/2)*tan((sm-am)/2)*tan((sm-bm)/2)*tan((sm-cm)/2);
z=sqrt(z2);

u1=1/cos((am+bm+cm)/4)^2*tan((bm+cm-am)/4)*tan((am+cm-bm)/4)*tan((am+bm-cm)/4);
u2=tan((am+bm+cm)/4)/cos((bm+cm-am)/4)^2*tan((am+cm-bm)/4)*tan((am+bm-cm)/4);
```

```
u3=tan((am+bm+cm)/4)*tan((bm+cm-am)/4)/cos((am+cm-bm)/4)^2*tan((am+bm-cm)/4);
u4=tan((am+bm+cm)/4)*tan((bm+cm-am)/4)*tan((am+cm-bm)/4)/cos((am+bm-cm)/4)^2;

dzda=-1/8*z2^(-0.5)*(u1-u2+u3+u4);
dzdb=-1/8*z2^(-0.5)*(u1+u2-u3+u4);
dzdc=-1/8*z2^(-0.5)*(u1+u2+u3-u4);

% Note z2 can be neglected here

h=4/(1+z2)*[dzda dzdb dzdc];

% cov(noise_area)

p_area=2*sigm^2*h*h';
```

E.49. StarDistrib.m

```
% ========================================================================
%
% StarDistrib.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Determine the distributions of stars in over 1000 random attitudes
% given a FOV and magnitude limit. Brute force check of all stars to see if
% they are in the field of view. Good check of results of RandAttTest.m,
% since the dsitribution of stars should look alike.
%
% INPUTS: Stars - list of stars
%
% OUTPUT:   none
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

clear;

vFOV = 8 * pi / 180         % Star Tracker Field of view (rads)
magLim = 6                  % Magnitude limit of star tracker
hmax = 20;                  % Lump all counts greater than hmax stars together.

load Stars                  % Load star catalog (ordered by magnitude)

ntests = 1000

tic

% Do ntests number of random attitudes and count stars in FOV

for i = 1:ntests
    i
    h(i) = 0;
```

```
    % Random attitude in spherical coords

    alpha = rand(1)*2*pi;
    beta = rand(1)*pi- pi/2;

    % Convert to cartesian

    x = [ cos( beta )*sin( alpha ); sin( beta ); cos( beta )*cos( alpha ) ];

    % Check all stars on lists to see if its in FOV

    for j = 1:nStars
        if Star(j).Mag >= magLim    % break j Loop at first overly dim star
            break
        else
            gamma = acos( dot( Star(j).Vector, x ) );
            if gamma <= vFOV / 2
                h(i) = h(i) + 1;    % Increment number of stars for that test
            end
        end
    end

    if h(i) > hmax+1                % All occurances above hmax are lumped together
        h(i) = hmax+1;
    end
end

% Create a histogram of results

nh=hist( h, 0:hmax+1 );
nhmax = max(nh)

figure(1)
hist( h, 0:hmax+1 )
xlabel('Stars in FOV')
ylabel('Occurances')
title('Number of Stars in FOV Based on Random Attitude Tests')
axis tight;

% Graph probability of seeing a certain number of stars within FOV
```

```
s(hmax+3)=0;
for i=hmax+2:-1:1
    s(i) = s(i+1)+nh(i);
    p(i) = s(i)/ntests*100;
end

figure(2)

bar(0:hmax+1,p)
xlabel('Minimum Number of Stars in FOV')
ylabel('Chance of occuring (%)')
title('Number of Stars in FOV Based on Random Attitude Tests')
axis tight;

toc
```

E.50. WaitForKeyPress.m

```
% ========================================================================
%
% TITLE.M
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Waits for a key press. Ignores mouse clicks so that views can be
% manipulated. Requires at least one open plot to work.
%
% INPUTS:   none
%
% OUTPUT:   none
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

'Press A Key...'

w=0;

while w ~= 1;
    w = waitforbuttonpress;
end
```

E.51. WhichNode.m

```
% ========================================================================
%
% WhichNode.m
%
% THESIS: FAST STAR PATTERN RECOGNITION USING SPHERICAL TRIANGLES
% Craig L Cole
% 8 January 2003
%
% Determines which of four nodes vector lies within
%
% INPUTS:   Vt - target vector to be located
%           nodeptr - array of four node nos. to examine
%
% OUTPUT:   n - node no. in which target vector lies within
%
% SUBROUTINES REQUIRED: none
%
% ========================================================================

function n=WhichNode( Vt, nodeptr );

global Vertex Node

n = 0;

for i = 1:4

    % Get Vertex Numbers that make up the node being pointed to

    v1 = Node( nodeptr(i) ).Vertex(1);
    v2 = Node( nodeptr(i) ).Vertex(2);
    v3 = Node( nodeptr(i) ).Vertex(3);

    % Calculate cross product between vertex vectors. The results are
    % the values used for the equation of the plane formed by them

    n12 = cross( Vertex(v1).Vector, Vertex(v2).Vector );
    n23 = cross( Vertex(v2).Vector, Vertex(v3).Vector );
    n31 = cross( Vertex(v3).Vector, Vertex(v1).Vector );
```

```
    % Insert values from normals into coefficients for equations of planes

    A = [ n12(1) n23(1) n31(1) ];
    B = [ n12(2) n23(2) n31(2) ];
    C = [ n12(3) n23(3) n31(3) ];
    D = 0;                                % plane passes through (0,0,0)

    % x,y,z of Star

    x = Vt(1);
    y = Vt(2);
    z = Vt(3);

    % If star is within planes formed by vertex vectors, the solution of
    % all three plane equations will be positive.

    if A(1)*x + B(1)*y + C(1)*z + D >= 0
        if A(2)*x + B(2)*y + C(2)*z + D >= 0
            if A(3)*x + B(3)*y + C(3)*z + D >= 0
                n = nodeptr(i);
                break
            end
        end
    end

end
```