PERFORMANCE ANALYSIS OF A FIXED POINT STAR TRACKER ALGORITHM FOR USE ONBOARD A PICOSATELLITE

A Thesis

presented to

the Faculty of

California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Aerospace Engineering

by

Kenneth Daniel Diaz

August 2006

APPROVAL PAGE

TITLE: Performance Analysis of a Fixed Point Star Tracker Algorithm for Use Onboard a Picosatellite

AUTHOR: Kenneth Daniel Diaz

DATE SUBMITTED: 8/9/2006

Dr. Jordi Puig-Suari	
Advisor	Signature
Dr. Eric Mehiel	
Committee Member	Signature
Dr. David Marshall	
Committee Member	Signature

ABSTRACT

PERFORMANCE ANALYSIS OF A FIXED POINT STAR TRACKER ALGORITHM FOR USE ONBOARD A PICOSATELLITE

Kenneth Daniel Diaz

This thesis explores the feasibility of performing star tracker pattern matching algorithms for attitude determination in fixed point for potential uses onboard a picosatellite. Many algorithms have been developed and published covering the pattern matching process of a star tracker. This thesis examines three different pattern matching algorithms. Tools were developed to emulate camera hardware and stellar centroiding for input to each algorithm. Based on the functions used within the algorithm and its overall complexity, a single algorithm was selected to be implemented in fixed point in MATLAB. Several modifications to the algorithm were necessary to allow for operation in fixed point. For 1000 simulations, when enough stars were present in floating point: 97.6% of the time the correct match was found; 0.1% of the time an incorrect match was found; and 2.3% of the time no match was found. For 100 simulations when enough stars were present in fixed point: 89% of the time the correct match was found; 0% of the time an incorrect match was found; and 11% of the time no match was found. The results of the two methods were comparable and the loss in accuracy upon converting to fixed point was expected. Instances when the star tracker failed to find a correct match were examined and the probable causes were determined. Solutions to these problems were also suggested for future implementation on the operational system.

ACKNOWLEDGEMENTS

First and foremost I would like to thank my parents and my brother for all of the love and support they have given me throughout my life, especially my years here at Cal Poly. I would not be the person I am today without you and the rest of our family. Dr. P, thank you for all the time you spent working with me on this thesis. This work would not be nearly as good as it is had you not motivated me by showing so much personal interest. You are a good friend and I look forward to staying in contact with you over the years. Thanks to the PolySat and CubeSat teams. From the guidance from our senior team members, I was able to learn and prosper in this project. The ridiculous times in the lab will be lasting memories. Erin, without a doubt you dragged me kicking and screaming through our final year at Cal Poly. Had it not been for your wisdom, friendship and willingness to put up with me, I would not have done as well as I have. Thank you for everything you have done for me. Finally, without my good friends on the deck, at high camp, and base camp I never would have been able to release the built up frustration from school and the project. I could never thank you guys enough for those times.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

1.1 Small Satellite Background

Aerospace companies are starting to come to the realization that with the ongoing development of smaller and better electronics, spacecraft no longer have to be the massive vehicles that they once were. This concept has led to the "smaller-faster-cheaper" idea in the aerospace industry that has lead to the success of vehicles such as Mars Pathfinder. Universities worldwide are beginning to catch on to this idea as well.

While industry is making note of these concepts, there still seems to be hesitation among the large companies to really push for the small satellites. Universities, however, are capable of doing so because there is little to lose when education is the primary goal. Students across the globe have begun developing small satellites over the past decade. These satellites typically are constrained by mass, volume and power available. They have a range of capabilities – from simply sending a radio signal letting the student operators know it's still alive, to taking pictures with cell phone cameras and sending the images back to Earth. The payloads that can be flown onboard these picosatellites, however, are extremely limited due to the fact that these vehicles have few (if any) means of attitude control. Attitude control will open up the doors to endless opportunities for picosats to fly more interesting, and more technologically advanced payloads. However, in order to attain attitude control the vehicle must first be able to accurately perform attitude determination.

1.2 Attitude Determination Background

Most sensors currently available for use onboard small, university satellites are commercial off-the-shelf (COTS) parts. Typically, these parts were not designed for use on space technology. Sensors that many of the picosatellites have onboard are magnetometers, MEMS rate gyros, and sun sensors. All three of these are usually low quality sensors leading to poor accuracy. The magnetometers are usually embedded within the satellite, which means that the readings that they will produce are poor due to the satellite circuitry enveloping them. This is why on the bigger satellites magnetometers are placed out on booms so they are away from the disturbances caused by the rest of the vehicle. The MEMS gyros typically will not provide an accurate reading at slow rates which most of the picosatellites will be tumbling at. No high quality sun sensors have been developed yet for a system as small as a university satellite. Some schools are attempting to use solar cells as rough sun sensors. For attitude knowledge to converge over time, each of these sensors must be passed through some sort of filter. None have the capability of accurately providing an instantaneous estimate of spacecraft attitude.

1.3 Star Trackers

1.3.1 Commercial Star Trackers

Star trackers are the only sensors available that standard spacecraft can use to provide a real-time attitude estimate. This is done by taking an image of the stars, and then comparing the image to a look-up table stored in memory. This system allows a spacecraft to know exactly how it's pointing at any given time. A comparison of

commercial star trackers can be found in Figure 1.1. In each chart the accuracy of three star trackers and a typical sun sensor are plotted relative to price, mass, and power consumption. All three star trackers are in the hundreds of thousands of dollars, much more than most university programs can afford for a single sensor. Each star tracker is at least 2 kilograms as well, much more mass than small satellites such as CubeSats could allocate to a single sensor. The power consumption would take all of the available power from most small satellites because most picosatellites operate at about 3 – 5 volts. These systems simply aren't capable of being placed inside small satellites.

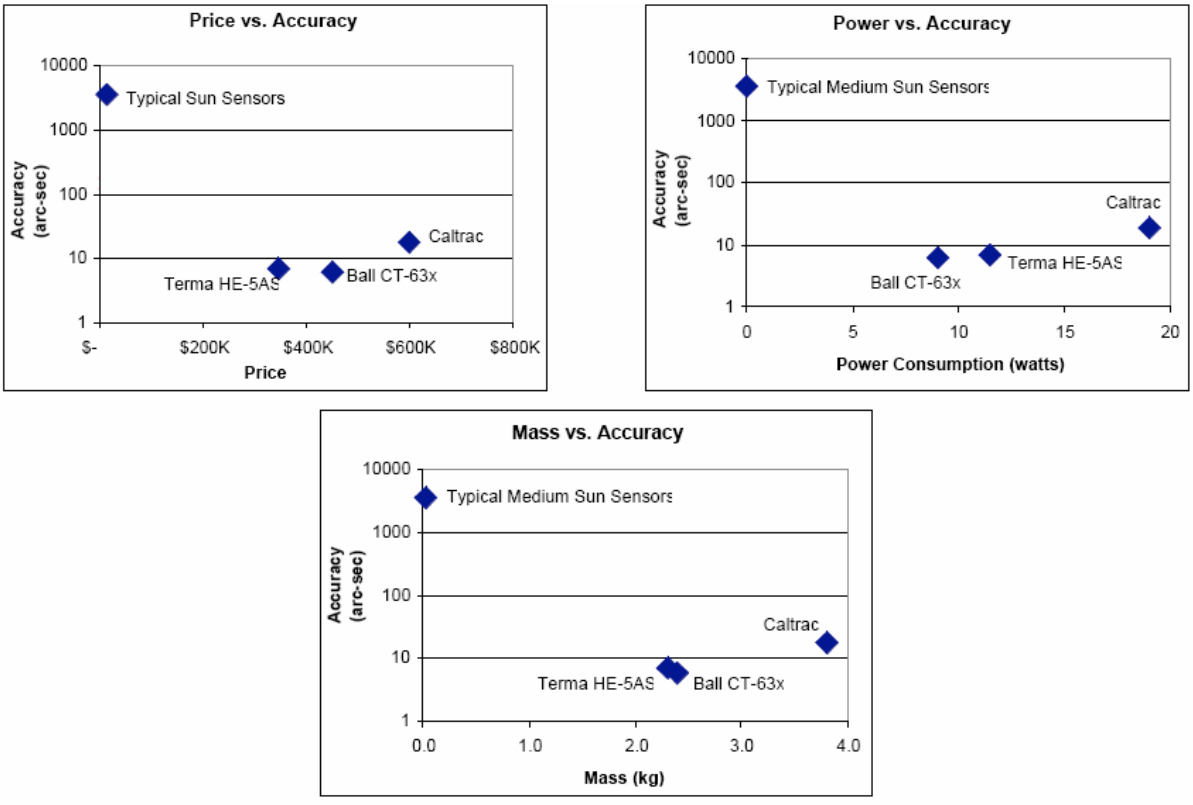

Figure 1.1: Comparison of Commercial Star Trackers

1.3.2 Can Students Develop Star Trackers for Small Univ. Satellites?

Students have shown their ability to take ideas – such as small satellite development – and turn them into reality. This would suggest that students may be capable of developing new ways of using current COTS parts in ways that industry has not yet tried. If students were able to develop a star tracker from cheap, COTS components which was comparable in performance to the high-priced, massive, power-consuming star trackers used by industry, the small satellite industry would be revolutionized.

1.3.2.1 CubeSat Cameras

Thus far several of the CubeSat developers have put small, cell phone cameras in space. The most notable satellite is University of Tokyo's XI-IV CubeSat which was launched in 2003. It carried a 256 x 240 imager onboard. The satellite and one of the images downloaded from it can be seen below in Figure 1.2.

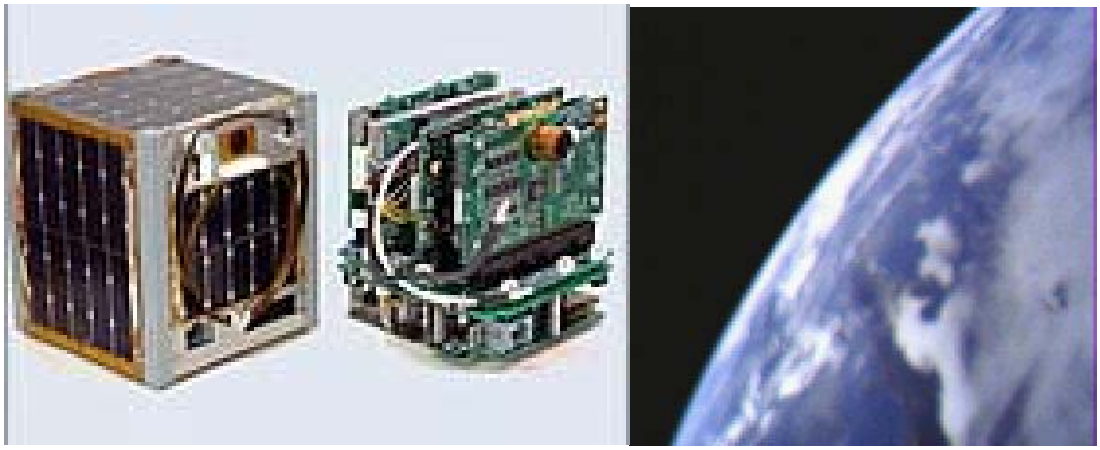

Figure 1.2: a) Univ. of Tokyo's XI-IV; b) Image from XI-IV

With simple cameras already developed and proven on small satellites, the next logical step would be to look into processing capabilities.

1.3.2.2 Processing for a Star Tracker

In order for a star tracker to perform correctly, it must have a processor which is capable of performing matrix algebra, as well as trigonometric functions. There must also be a significant amount of memory devoted to the star catalog and look-up table. Currently most small university satellites do not use processors which can perform the complex trig calculations. This is because the small processors primarily run in fixed point. Some of the processors do have a way of emulating floating point, allowing for the use of math libraries containing trig functions. This however can take a lot of time, resulting in more power consumption. The larger satellites use large, power-hungry star trackers which can operate in floating point. This allows for the high accuracy which those star trackers provide. Universities and small satellite developers must find a way to make the star tracker algorithms operate entirely in fixed point, using primarily COTS components.

1.4 Thesis Overview

1.4.1 Thesis Motivation

The work presented will evaluate the feasibility of performing star tracker algorithms onboard a simple processor which may be used on a small satellite. The small satellite community's ability to fly more complex and scientifically valuable payloads is very limited by satellites' inabilities to perform real time attitude determination. Simple star trackers may be the key that can open the door to all of the payloads that the small satellite community currently does not have access to. By evaluating whether or not simple star tracker algorithms can be effectively run on the low-power, simple processors

available to the community, a huge step forward is made in the evolution of small satellites.

1.4.2 Thesis Framework

All algorithms in this thesis are written using MATLAB. This was done because MATLAB has become a universal tool for simulation in the aerospace industry. The assumption is made that images have been taken and stars have been centroided. This is done via a camera emulator that was developed. The emulator will be discussed in further detail in Section 4.3. The functions used in the development of the fixed point code have been simplified as much as possible in order to make them capable of operating on a fixed point processor.

CHAPTER 2: COORDINATE FRAME DEFINITIONS

2.1 Earth-Centered Inertial (ECI)

The Earth-Centered Inertial (ECI) frame is centered on the Earth and fixed in inertial space. The x-axis is aligned with the radial vector from the Sun to the Earth on the vernal equinox, also known as the line of Aries. The z-axis is aligned with the Earth's orbital angular momentum vector. The y-axis is formed from the right-hand rule between the x- and z- axes. Figure 2.1 shows a drawing of the ECI frame.

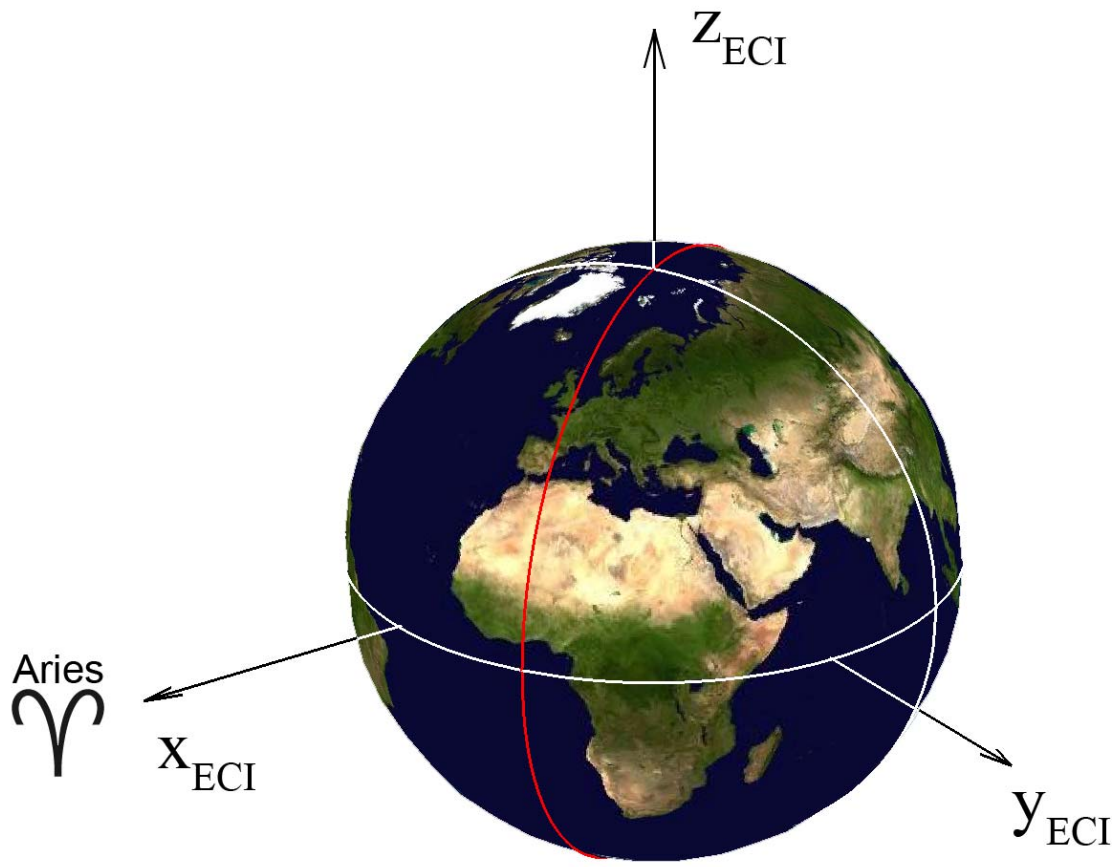

Figure 2.1: Earth-Centered Inertial Frame

2.2 Body Frame

The Body-Fixed (Body) frame is aligned geometrically with the spacecraft. The axes are an orthogonal set of unit vectors typically aligned with three of the principle moments of inertia of the spacecraft. Figure 2.2 shows a drawing of the Body frame.

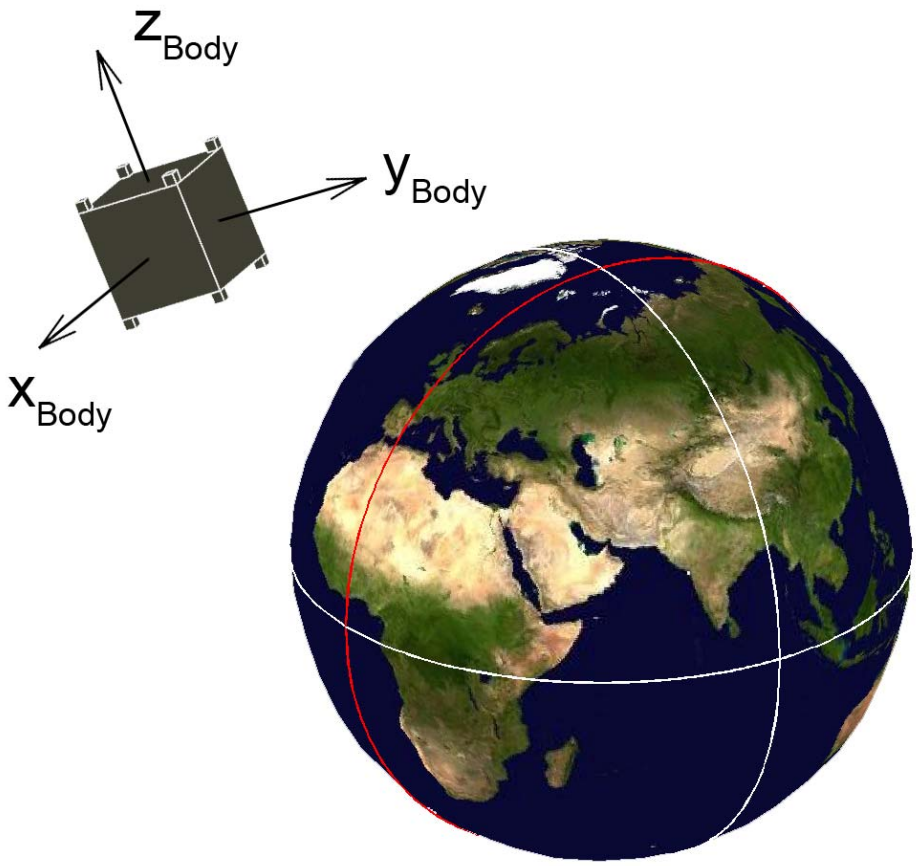

Figure 2.2: Body-Fixed Frame

2.3 Camera Frame

The camera frame is aligned with the camera onboard the spacecraft. For the sake of simplicity it is assumed that the camera frame and body frame are aligned for this thesis. This assumption can be augmented with a single rotation matrix between those two

frames once the final vehicle is integrated and the orientations of each frame are known. Figure 2.3 below shows a diagram of the camera frame.

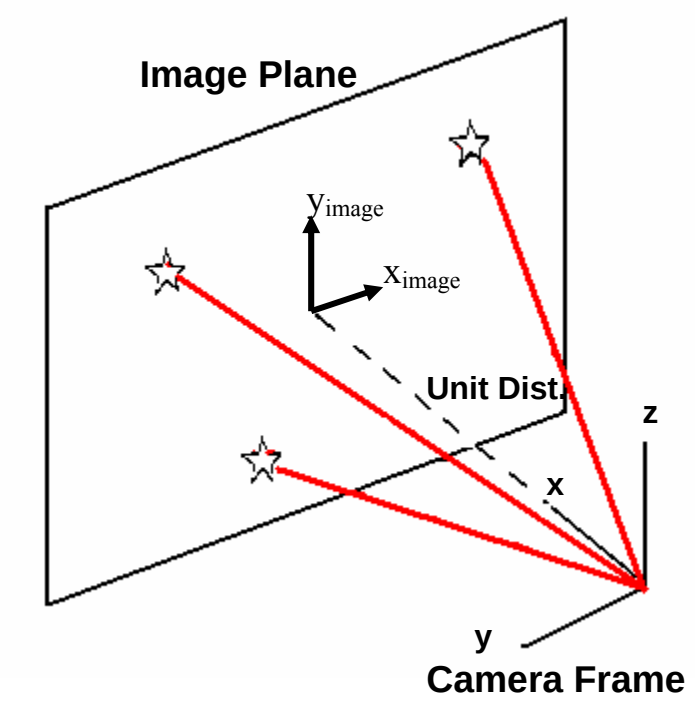

Figure 2.3: Camera Frame

In the camera frame, the x-axis is normal to the image plane, at a unit distance. The y-axis of the camera frame is parallel to the negative x-axis on the image, and the z-axis in the camera frame is parallel to the y-axis on the image.

CHAPTER 3: PATTERN MATCHING METHODS

The following are generic descriptions of three different algorithms used for pattern recognition of star images. Each algorithm has already been developed and is publicly available.

3.1 Area & Polar Moment Method

This method relies on the area and polar moment of the planar triangles found in the image to estimate spacecraft attitude. The area and polar moment of the triangles on the image can be compared to the same values stored in the look-up table. How the triangle is built can be seen below in Figure 3.1.

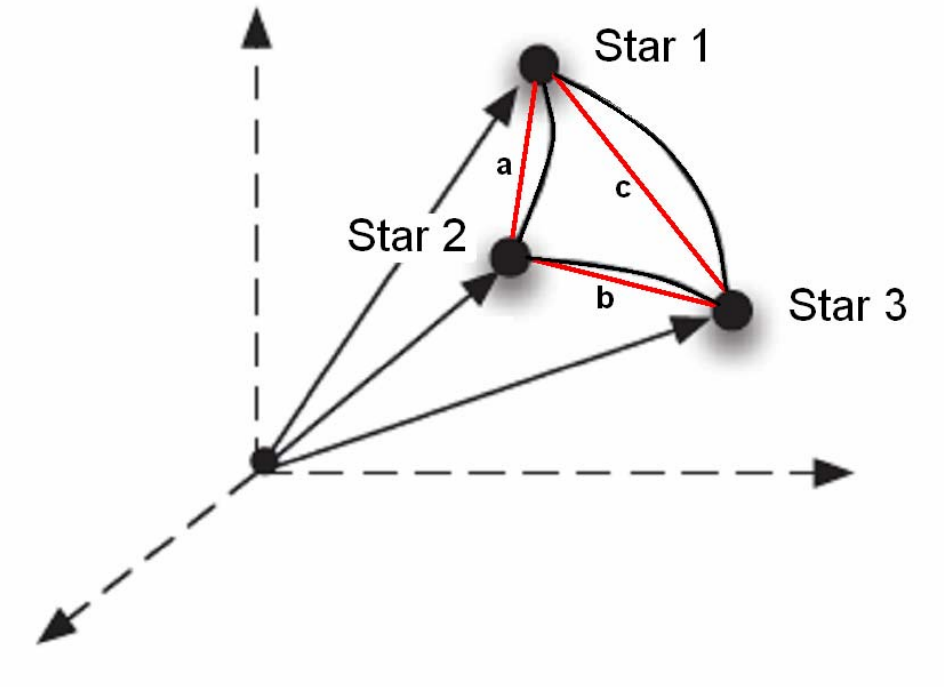

Figure 3.1: Area & Polar Moment of Planar Triangles

The following set of equations is used to calculate area and moments.

$$Area = \sqrt{s(s-a)(s-b)(s-c)}$$
$$Moment = Area\left(a^2 + b^2 + c^2\right)/36$$
$$s = \frac{1}{2}(a+b+c)$$
$$a = \left\|\vec{V}1 - \vec{V}2\right\|$$
$$b = \left\|\vec{V}2 - \vec{V}3\right\|$$
$$c = \left\|\vec{V}1 - \vec{V}3\right\|$$

The vectors, $\vec{V}1$, $\vec{V}2$, and $\vec{V}3$, represent the vector from the origin to the respective star in the coordinate frame being used at the moment of operation (ECI, or camera). The variables 'a', 'b', and 'c' can be seen above in Figure 3.1 as the sides of the planar triangle.

These operations are performed in ECI while creating the look-up table. This will only happen on the ground on a regular computer. When implemented onboard the satellite, this algorithm will be carried out in the camera frame.

3.2 Planar Angle Method

This method, as before, relies on constructing a planar triangle. However for this method the angles that make up the triangle are used in the searching process, rather than the areas and moments. Figure 3.2 shows how the stars are used to construct the triangle, and the three planar angles associated with that triangle.

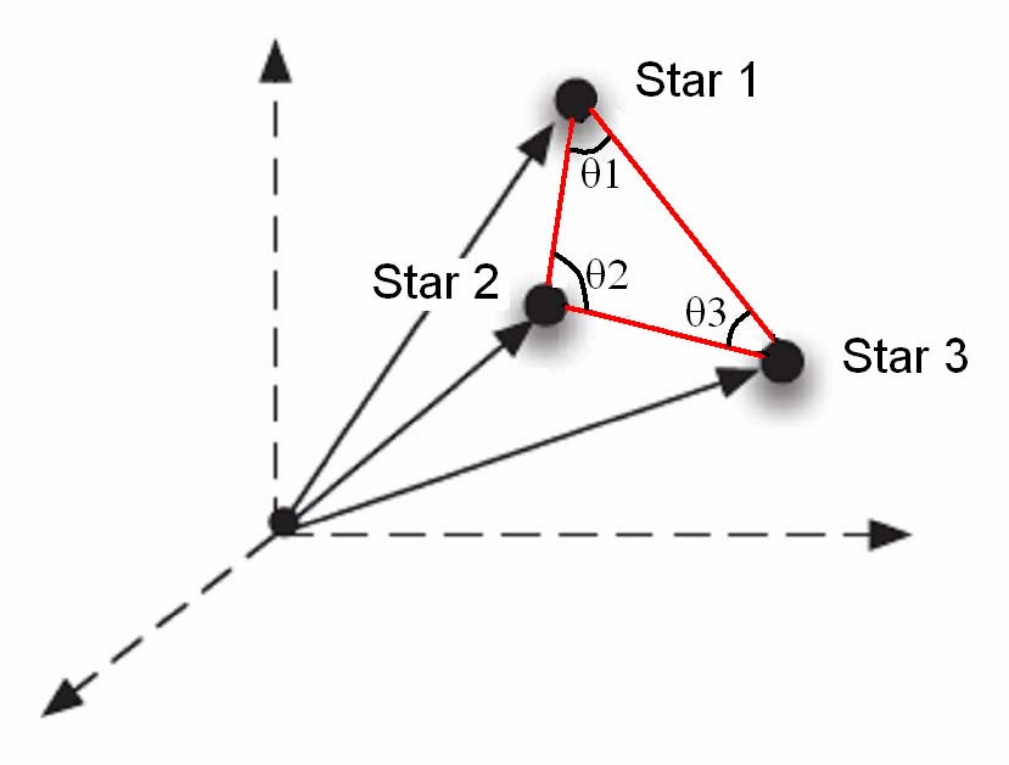

Figure 3.2: Planar Angle Method

The equations required to derive the three planar angles can be found below.

$$\theta_1 = \arccos\left(\frac{a^2 - c^2 - b^2}{-2cb}\right)$$

$$\theta_2 = \arccos\left(\frac{b^2 - a^2 - c^2}{-2ac}\right)$$

$$\theta_3 = \arccos\left(\frac{c^2 - a^2 - b^2}{-2ab}\right)$$

The equations implement the Law of Cosines in order to calculate the planar angles. For this method ‘a’, ‘b’, and ‘c’ are all calculated the same way as in Section 3.1.

Once again this method is applied for two cases: once on ground for look-up table development, and then onboard the satellite for pattern matching purposes. As before the

ground portion operates in the ECI frame and the onboard portion operates in the camera/body frame.

3.3 Vector Angle Method

The final method that was evaluated for pattern matching was using the angle between the star vectors as the values for defining star triads. How the stellar vectors are defined and the angle between those vectors can be seen below in Figure 3.3.

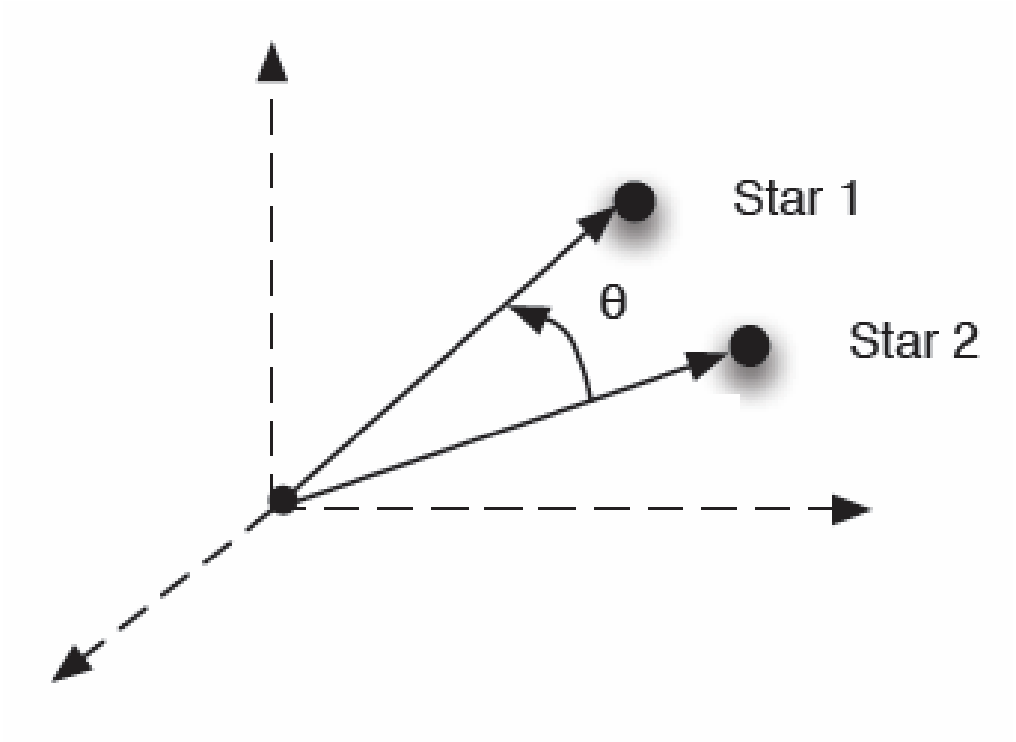

Figure 3.3: Vector Angle Method

In order to calculate the angles between the vectors, the basic equation for the dot-product of two vectors is used, as seen below.

$$\theta = \frac{\arccos\left(\vec{V}_1 \cdot \vec{V}_2\right)}{\left\|\vec{V}_1\right\| \cdot \left\|\vec{V}_2\right\|}$$

This method was used for the development of the look-up table. In order to simplify calculations, two different methods were evaluated for determining this angle based on stellar locations on the image. One of those methods involved building the stellar vectors and taking the dot product, and the other relied pixel distances and angular resolution of the camera. Both methods will be discussed in further detail in Chapter 6:.

CHAPTER 4: SIMULATION SETUP

4.1 Star Catalog

The star tracker must accurately match stars from an image to those from a database. In order to build this database with all of the necessary information (areas, angles, etc.), a star catalog first had to be developed. For this thesis the star catalog was required to include star locations in ECI coordinates as well as the Right Ascension and elevation of each star. The more expensive star trackers also include the magnitude of each star in the catalog. This allows for more accurate matching of stars. However for this application, the quality of the imager was unknown, and therefore it was not known whether there will be an effective method for distinguishing between stars of different magnitudes. Also, in order to maintain the philosophy of simplicity, it was decided to only base this star tracker on stellar coordinates alone rather than including magnitudes.

The magnitudes were, however, used to determine the number of stars which were stored in the catalog. Because the imager quality was unknown, it was assumed that only the brightest stars would be visible. Therefore only stars with magnitudes 4 and brighter were included in the star catalog. This limit was applied to the NASA I/239 star catalog, which comprises data for visible stars from the Hipparcos and Tycho star databases. The 2-D map of the star catalog used for the star tracker presented in this thesis can be seen in Figure 4.1, while the 3-dimensional model in ECI can be seen in Figure 4.2.

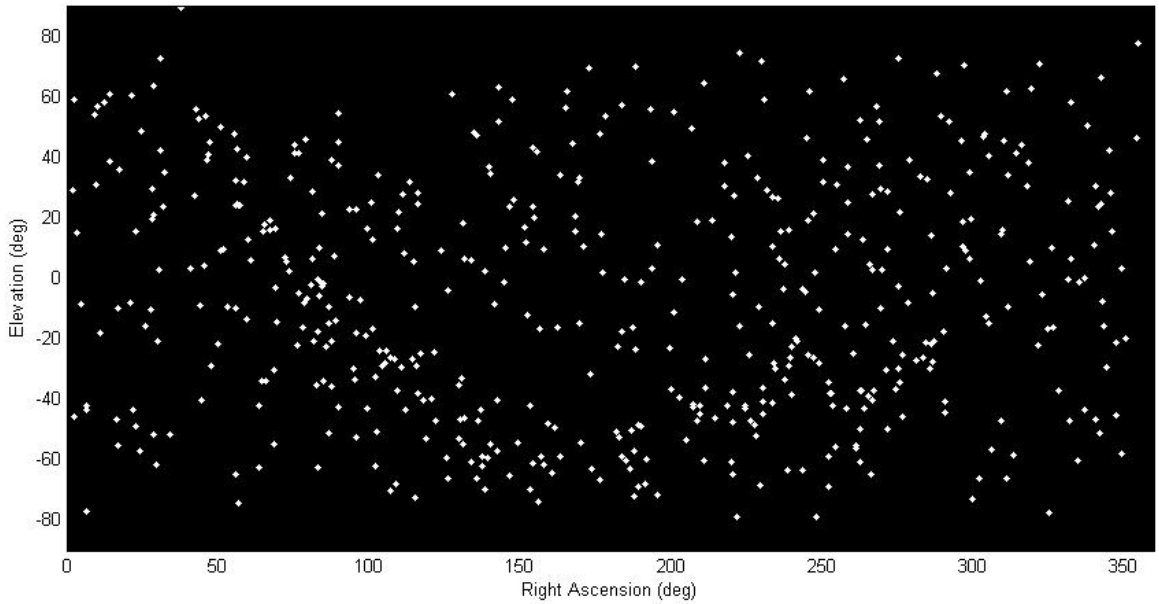

Figure 4.1: 2-Dimensional Star Catalog

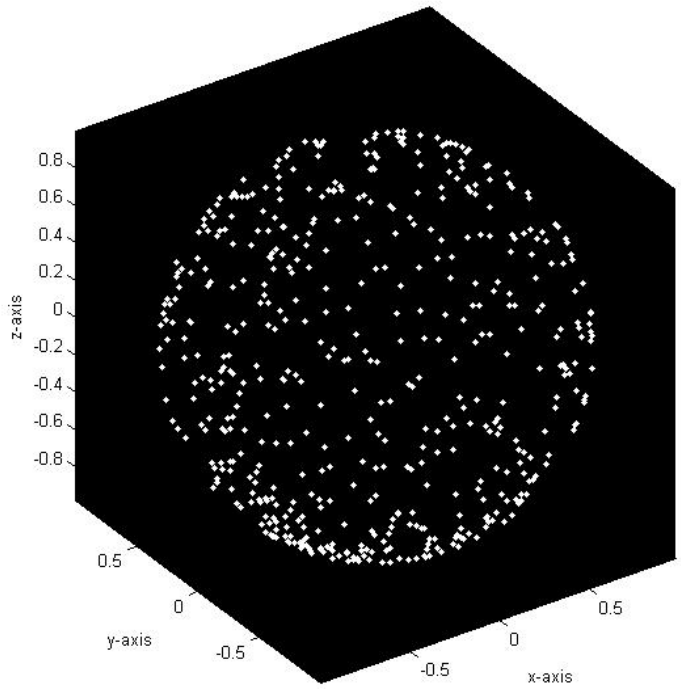

Figure 4.2: 3-Dimensional Star Catalog in ECI

4.2 Look-Up Table

Once the star catalog was selected, it was time to build a database containing all of the relevant data that was necessary in the pattern matching and quaternion determination process.

4.2.1 Finding All Possible Triads

The first step to build the catalog was to determine all of the possible triads in the star catalog, which exist within a given filed of view. The maximum field of view selected was 30 degrees. This decision was based on the fact that most of the commercial lenses for cheap cameras were anywhere between 10 to 50 degrees. 30 degrees was found to be enough to have a significant number of stars present on the image for any given orientation.

This process was accomplished by taking every star in the catalog, and performing the dot product followed by and arccosine operation with every other star in the catalog. Only the combinations which were within 30 degrees of each other were stored. When three stars were all found to be within 30 degrees of each other, the triad was stored in the final look-up table.

4.2.2 Look-Up Table Development

Once all the triads were known, the values relevant to each algorithm could then be calculated. The equations presented in Chapter 3: were used to calculate the areas, moments, and angles necessary for each pattern matching algorithm.

4.2.3 Look-Up Table Format

4.2.3.1 Area & Polar Moment Method

The Look-Up database for the Area & Polar Moment Method was sorted in ascending order based on area of the planar triangle. The table containing all of the triads was also sorted simultaneously ensuring that the triads and their corresponding areas and moments all maintained the same index number. Table 1 and the relationships below it describe how the database was stored.

Table 1: Area & Polar Moment Look-Up Table Format

Triad 1	Area1	Moment1
Triad 2	Area2	Moment2
...	**...**	**...**
Triad n	Area(n)	Moment(n)

$$Area1 < Area2 < ... < Area(n)$$

4.2.3.2 Planar Angle, and Vector Angle Method

For the other two methods for pattern recognition, a different method was used for organizing the database. Because both of these methods use three angles to distinguish between different triads, it becomes necessary to not only sort the rows of the table, but the columns as well. To make the table as easy as possible to read and search through, the angles for each triad were sorted in ascending order from left to right, and then the rows were sorted in ascending order as well. Table 2 and the equations below better describe how the two look-up tables for these two methods were formatted.

Table 2: Planar Angle, and Vector Angle Look-Up Table Format

Triad 1	Angle1-1	Angle1-2	Angle1-3
Triad 2	Angle2-1	Angle2-2	Angle2-3
...	...	...	...
Triad n	Angle(n)-1	Angle(n)-2	Angle(n)-3

$$Angle1-1 < Angle1-2 < Angle1-3$$
$$Angle1-1 < Angle2-1 < \ldots < Angle(n)-1$$

4.2.4 Look-Up Table Size

With the star catalog, the sets of triads, and the angles (or areas and moments) of each triad stored into a large look-up table, the total size came out to be approximately 8 megabytes (MB). This is large, but still feasible for the majority of the picosatellites. The more stars that are added, the larger the look-up table will be. It would most likely increase at an exponential rate.

4.3 Camera Emulator

Because there is no actual camera hardware involved, a camera emulator had to be developed. This emulator provided the coordinates on the image of the centroids of each star. The units of the image are in pixels. It was assumed that the stars were centroided to the accuracy of one pixel, and that the centroid was located in the center of the pixel. For simulation purposes, the index numbers of the stars on the image were also part of the emulator output. This, in essence, allows the user to validate whether or not the stars that the algorithm claims are on the image are actually there.

4.3.1 Image Simulation Process

In order to simulate an image, several steps had to be taken. A bore sight generator provided a random orientation for the camera. This came in the form of a single bore sight vector, which was defined by the selection of a random right ascension, elevation, and roll. The dot product operation was then performed on this vector with the vector of every star in the catalog. The arc-cosine operation was then performed as well. The equations can be seen below.

$$dot\_product = V_{Random} \cdot V_{Star_n}$$
$$\theta = \arccos(dot\_product)$$

The stars that had θ within 15 degrees of the random bore sight vector were stored as part of the image. The 15 degrees was chosen because it is half of the 30 degree maximum field of view.

Once all of the stars that were within the field of view were determined, they were projected onto a flat plane. This flat plane was the image. Figure 4.3 depicts the process of projecting the stars from the ECI unit sphere onto the image plane. The black dots represent the stars on the sphere, while the white dots represent the stars projected onto the image.

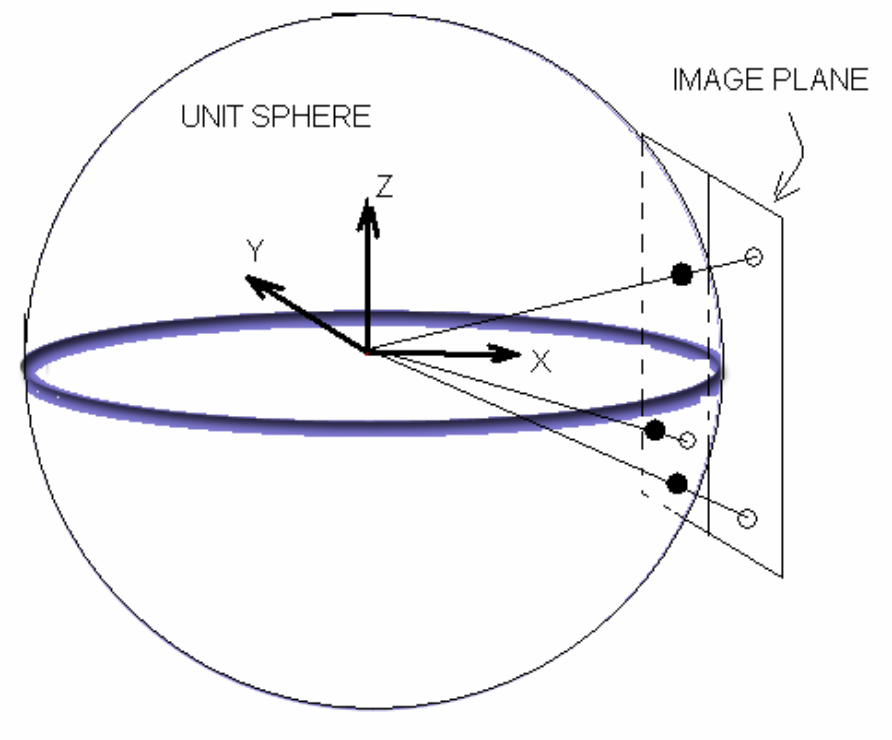

Figure 4.3: Image Simulation

The stars on the image were still being represented in the ECI coordinate frame. Therefore the entire image had to be rotated to the camera frame and then converted to units of pixels.

The rotation that took the stars from the random bore sight to the camera frame can be seen below. Every ‘c’ and ‘s’ represent cosine and sine respectively.

$$ {}^{Cam}T^{Bore} = \begin{bmatrix} c(\phi)c(\theta) & c(\theta)s(\phi) & s(\theta) \\ -s(\phi)c(\psi)-s(\psi)s(\theta)c(\phi) & c(\phi)c(\psi)-s(\psi)s(\theta)s(\phi) & s(\psi)c(\theta) \\ s(\phi)s(\psi)-c(\phi)c(\psi)s(\theta) & -s(\psi)c(\phi)-c(\psi)s(\theta)s(\phi) & c(\psi)c(\theta) \end{bmatrix} $$

For the above matrix, right ascension is represented by φ, elevation by θ, and roll about the bore sight by ψ.

In order to convert the units of the image to pixels, a conversion factor was calculated based on the field of view and image size. The relationship can be seen below.

$$ conversion = \frac{npixels/2}{\tan\left(\frac{\theta}{2}\right)} $$

This conversion factor was simply multiplied by the Y, and Z coordinates of the stars in the camera frame in order to get the centroids into pixel values.

4.3.2 Emulator Output

The emulator outputs several different values. First and foremost is the ‘image’ matrix. Table 3 describes the format of the matrix. In this table, ‘n’ is the total number of stars on the image, and ‘N’ is the index number of each star.

Table 3: Image Matrix Format

N_1	X-Location $_1$	Y-Location $_1$
...	...	...
N_n	X-Location $_n$	Y-Location $_n$

The other outputs of the emulator are the random right ascension, elevation, and roll about the bore sight, and the actual quaternions.

4.4 Search Algorithm

The search algorithm used for this thesis was a very rudimentary method. It implemented a “Brute Force” approach to searching. This was done due to the fact that simulation time was not one of the key performance parameters being analyzed.

The Lost-In-Space Pyramid Algorithm was evaluated for use in this system. However, in order to fully implement it, it would have become necessary to develop the ‘k-vector’ search technique. This technique provides a very fast method of performing searches through large amounts of data, which is ideal under these circumstances. On the other hand, maintaining the prior assumption that simulation run-time was not under scrutiny, it was decided to develop a more simple system that would do a more thorough search of the look-up table, even though it would take more time.

4.4.1 Modified Pyramid Algorithm

The pyramid algorithm used the basic technique of using the angles (or areas and moments in some algorithms) of individual triads on the image to find matches within the look-up table. The idea behind the pyramid was to use a fourth star as a verification of the match found on the first three stars.

The modification to the original algorithm was that a unique solution was not originally found for the first triad. Instead a tolerance was put on the search so that multiple solutions could be found. This also allowed for error in the calculation of the angles on the image. Once a table of solutions was found for the first three stars, a fourth star was selected. This fourth star created three more triads, resulting in a pyramid. An example of this can be seen below in Figure 4.4.

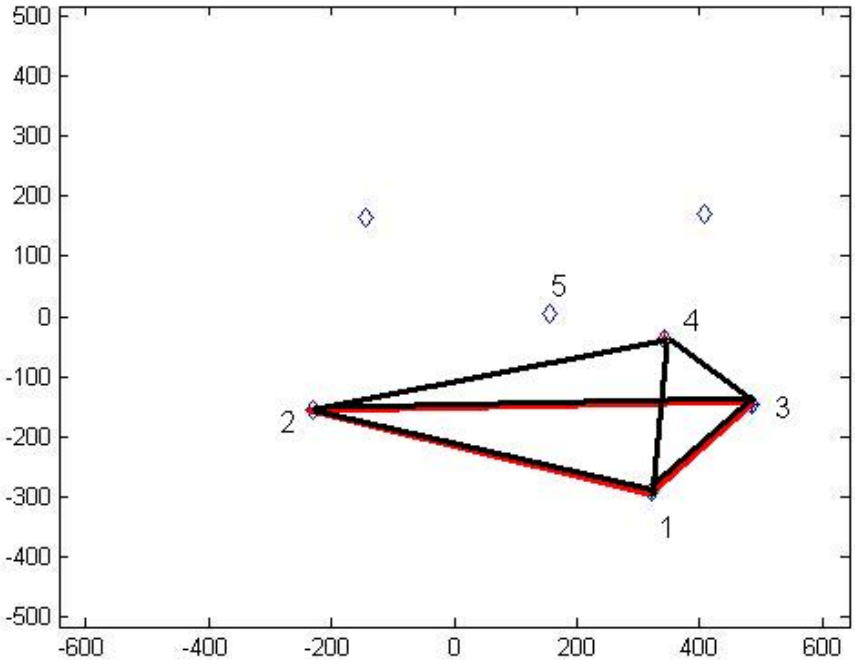

Figure 4.4: Pyramid Development

The red triangle represents the first triad, while the black triangles represent the second, third, and fourth triads.

Once the second, third and fourth triads were determined a search would be performed for each of them as well. This resulted in multiple solutions for each of the four triads. What was stored in each solution is the index numbers from the star catalog of each star. For a match to be found, a unique set of four stars must exist within the four sets of solutions. An example of this can be seen below. The four tables represent the four sets of solutions. The highlighted rows display a unique set of four stars among all of the solutions.

Triad 1:

113	104	86
122	134	111
172	154	175
142	151	166

Triad 2:

322	323	336
148	151	142
70	63	66
397	401	411
456	457	467

Triad 3:

339	330	371
142	148	166
49	57	59
334	325	367
167	181	109
300	287	322

Triad 4:

81	73	92
148	151	125
151	148	166

For this example, the original triad was matched was made up of the stars with index numbers 142, 151, and 166. The fourth star used for verification was 148.

This was the overall process used in searching for matches on the image. If no solution could be found, then the algorithm moved on to the next triad and repeated the process until a solution was found or until there were no more triads. It should also be noted that using this method required that an image needed at least 4 stars in order for the star tracker to perform its operations.

4.5 Quaternion Extraction

Once a match had been found, the quaternions could be determined. The first step was to build stellar vectors in the camera frame based on the image. In order to do this, the location on the image was used in terms of pixel values. It is known that in the camera frame, the image was a unit distance away from the origin in the x-direction. Therefore the only other dimensions that were needed were the y- and z-directions. These were supplemented by the pixel locations. On the image, the x-axis corresponded to the negative y-direction in the camera frame, and the y-axis on the image corresponded to the positive z-direction in the camera frame, as is shown in Figure 4.5.

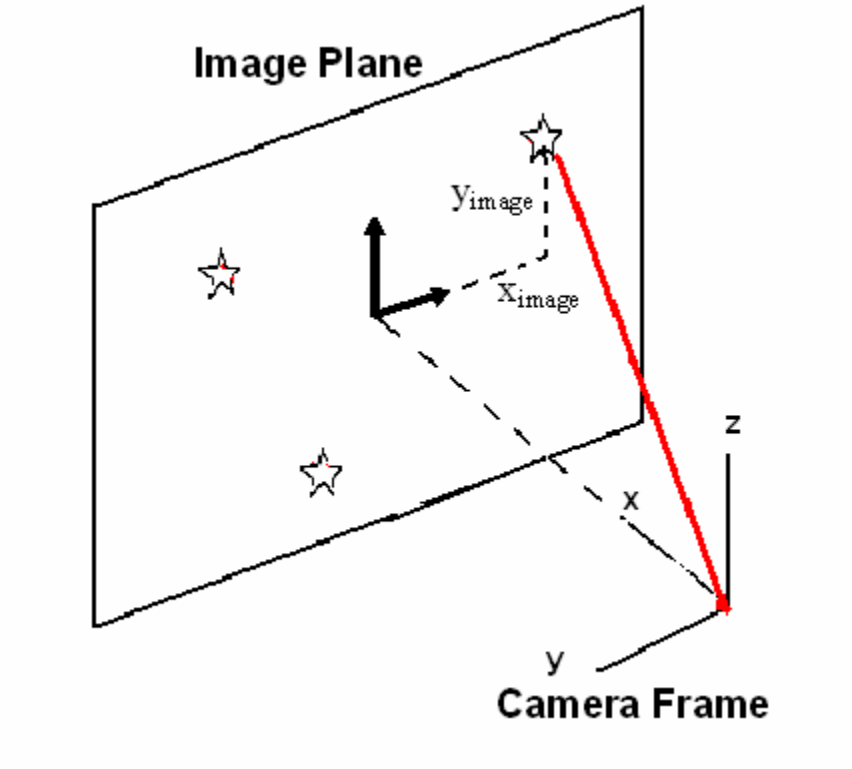

Figure 4.5: Star Coordinate Transformation

In order to get the image coordinates into the camera frame, the locations of the star on the image were divided by the same conversion factor that was used in Section 4.3.1. Then combining these values with the unit value in the x-direction provided a stellar vector in the camera frame. The equations used to calculate the vector can be seen below.

$$\vec{V}_{cam} = \begin{bmatrix} 1 \\ -x_{image} / conversion \\ y_{image} / conversion \end{bmatrix}$$

The vector then had to be normalized as follows.

$$\hat{V}_{cam} = \frac{\vec{V}_{cam}}{\left\| \vec{V}_{cam} \right\|}$$

This process was done for each of the three stars of the initial triad. The three vectors were then put into a single 3x3 matrix.

$$V_{CAM} = \left[\hat{V}_{1cam} \vdots \hat{V}_{2cam} \vdots \hat{V}_{3cam}\right]$$

The ECI coordinates of the three stars were also known from the look-up table. They were also put into a 3x3 matrix as follows.

$$V_{ECI} = \left[\hat{V}_{1ECI} \vdots \hat{V}_{2ECI} \vdots \hat{V}_{3ECI}\right]$$

The transformation matrix between the two coordinate frames could then be determined by inverting V_{ECI} and multiplying it by V_{CAM}.

$$T = V_{CAM} V_{ECI}^{-1}$$

The transformation matrix could be represented using the four parts of the quaternion as follows.

$$T = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$= \begin{bmatrix} q_1^2 - q_2^2 - q_3^2 + q_4^2 & 2(q_1 q_2 + q_3 q_4) & 2(q_1 q_3 - q_2 q_4) \\ 2(q_1 q_2 - q_3 q_4) & -q_1^2 + q_2^2 - q_3^2 + q_4^2 & 2(q_3 q_2 + q_1 q_4) \\ 2(q_1 q_3 + q_2 q_4) & 2(q_3 q_2 - q_1 q_4) & -q_1^2 - q_2^2 + q_3^2 + q_4^2 \end{bmatrix}$$

In order to extract the quaternions from the transformation matrix, the set of equations below had to be used.

$$q_4 = \pm 0.5\sqrt{1 + a_{11} + a_{22} + a_{33}}$$
$$q_1 = 0.25(a_{23} - a_{32})/q_4$$
$$q_2 = 0.25(a_{31} - a_{13})/q_4$$
$$q_3 = 0.25(a_{12} - a_{21})/q_4$$

The sign of q_4 could be either positive or negative as long as it was carried through in the other operations – both lead to the same set of quaternions.

The method used above to calculate the stellar vectors was used whenever it became necessary to calculate stellar vectors from the image plane, in every algorithm presented in this thesis. It was ideal because it avoided the use of trigonometric functions.

4.6 Star Tracker Simulation Flow

In order to better visualize the flow of the pattern matching and quaternion determination process, a flow chart was assembled and can be found in Figure 4.6. The blue oval shapes are the four options for algorithm completion. The pink diamonds represent decision, or branching points. The process started with image simulation via the camera emulator. The algorithm required four stars to be present on the image. If not enough stars were present then the algorithm immediately closed and outputted, “Not Enough Stars”. If four or more stars existed, then the algorithm went through all of the stars and constructed all of the possible triads on the image, as well as the relevant information based on the pattern matching method being used.

The first triad was selected and a search of the Look-up table was performed. All of the solutions that were within the selected tolerance were stored. In order for the algorithm to continue, at least one solution had to be found. If no solution was found then a check was performed to see if there were any more triads left to search. If there were no more triads,

the process ended and the algorithm outputted “No Matches”. If there were more triads then the algorithm moved on to the next one and started over.

However, if at least one solution existed for the original triad, then the fourth star was selected. This fourth star, along with the original three, created three new triads. The search was performed for the three new triads as was done the first. After the search was complete, what were left were four sets of solutions (one set for each triad). Among these solutions, there should have ideally been a unique set of four stars. A search was performed of the four sets of solutions, in order to find this unique set. If no unique solution was found, the algorithm moved on to the next triad on the image. If there were no more triads in the image, the algorithm gave the output “No Matches”.

If a unique match was found, the results were compared to the actual stars which were known from the camera emulator. If the match was incorrect, the algorithm stopped and displayed “Incorrect Match”. However in the event when the solution matched what the camera emulator provided, the quaternions were calculated, and the algorithm outputted “Correct Match”.

This process was eventually put into a loop in order to run Monte Carlo simulations, the results of which can be found in the following chapters.

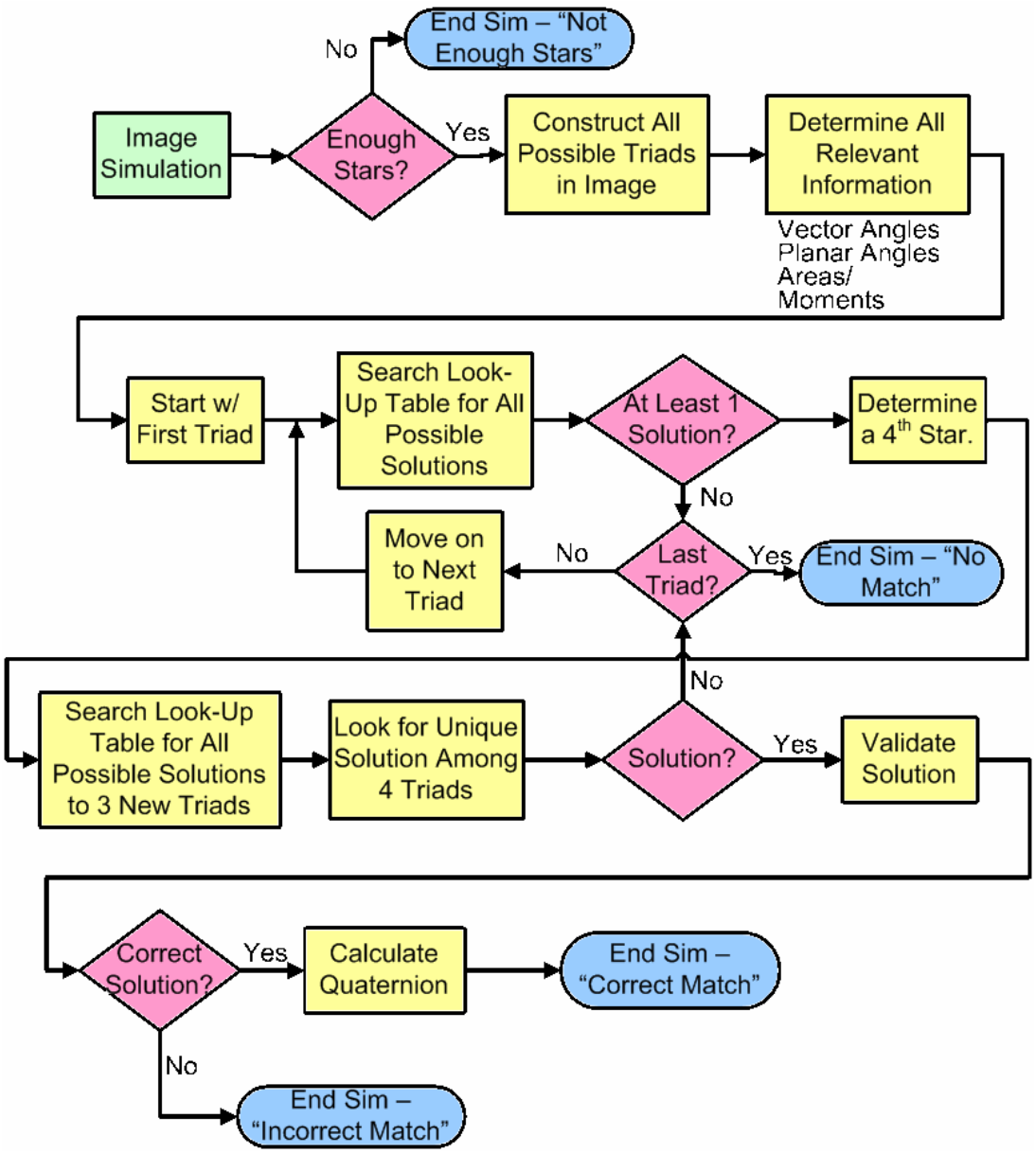

Figure 4.6: Simulation Flow

CHAPTER 5: FLOATING POINT SIMULATION RESULTS

5.1 Simulation Parameters

All three algorithms were run with similar simulation parameters for a generic comparison. The following parameters were used.

- Number of Simulations: 1000 random orientations
- Field of View (FOV): 30 degrees
- Search Tolerance:
  - Area Moment Method ± 1% of calculated value
  - Planar Angle Method ± 0.2 degrees
  - Vector Angle Method ± 0.2 degrees
- Image Array Size 1280 x 1024 pixels

The tolerances that were selected were the ones that at a glance seemed to provide the best accuracy for each method, in an effort to compare “apples to apples”.

5.2 Simulation Results

The following bar chart in Figure 5.1 compares the results of the three algorithms. The results are displayed in terms of the four options for exiting the algorithm: correct match, incorrect match, no match, or not enough stars.

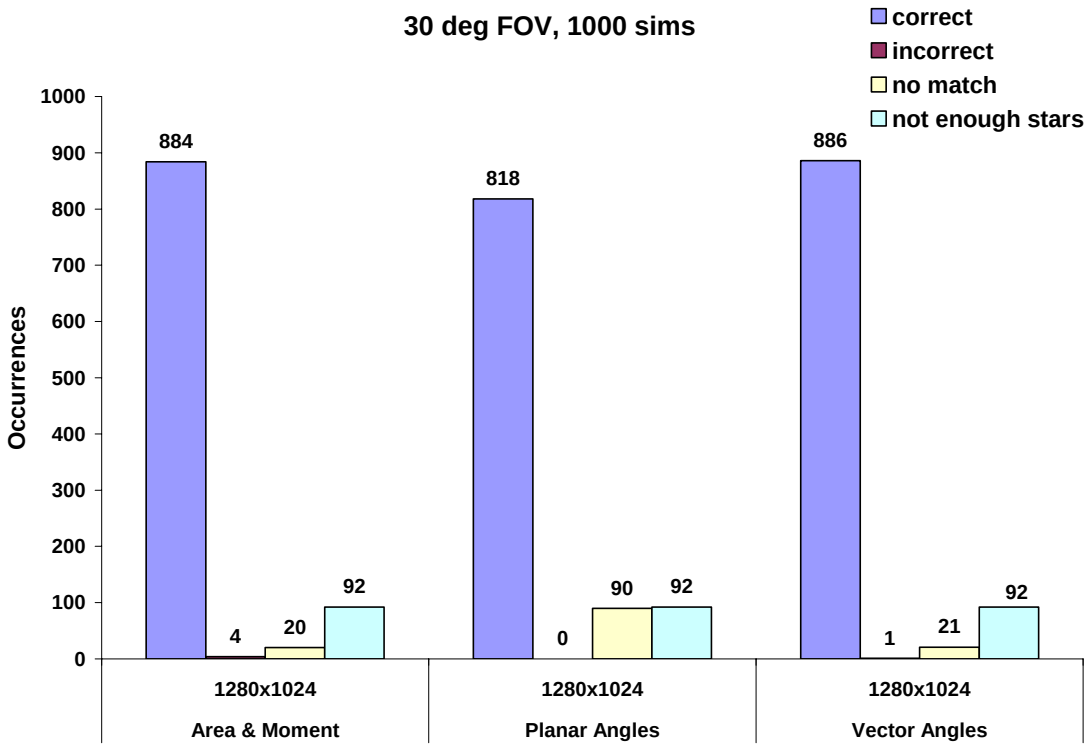

Figure 5.1: Performance Comparison of 3 Candidate Algorithms

What this reveals is that all three algorithms seem to work fairly well. However the Area & Moment method and Vector Angle method seem to find a correct match for approximately 67 of the 90 instances when the Planar Angle method could not.

In analyzing the results of the two methods which performed comparably, it can further be seen that the Vector Angle method only had a single instance when an incorrect match was found, while the Area & Moment method had 4 instances.

For these reasons the Vector Angle method was selected as the algorithm which would be converted into fixed point.

5.3 Failure Analysis

Prior to fixed point implementation of the Vector angle method, it was necessary to understand what happened during the few instances when the star tracker failed to find a correct match.

5.3.1 Incorrect Matches

Only a single instance resulted in an incorrect match. In order to determine the cause of that error, that single simulation was analyzed. Seemingly, it turned out that this was just a fluke incident occurring when two pyramids have striking resemblance. As you can see in Figure 5.2, the image on the left is the image provided to the algorithm, and the image on the right is what the star tracker claimed the image should look like. The stars highlighted in red are the ones which were matched. It can easily be seen that the two are very similar, which validates the thought that the incorrect match was just by chance.

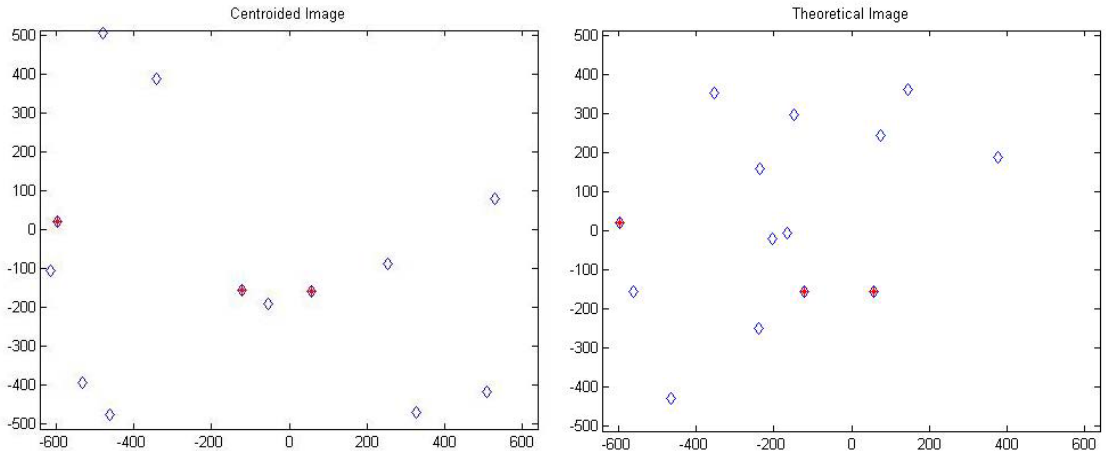

Figure 5.2: Incorrect Match Comparison

What allowed an incorrect match to actually be found would be the fact that the image is assumed to be centroided to the center of the pixel. If a more accurate centroiding technique were implemented on the real star tracker, the algorithm would more than likely find the correct match in this particular situation.

The instances when an incorrect match was found were expected to increase when the algorithm was implemented in fixed point, due to the fact that there was less accuracy on the fractional values.

5.3.2 No Matches

There were 21 instances when no match was found. In order to better understand why no matches were found for these cases, the orientations were plotted on the 2-D stellar map, as seen in Figure 5.3.

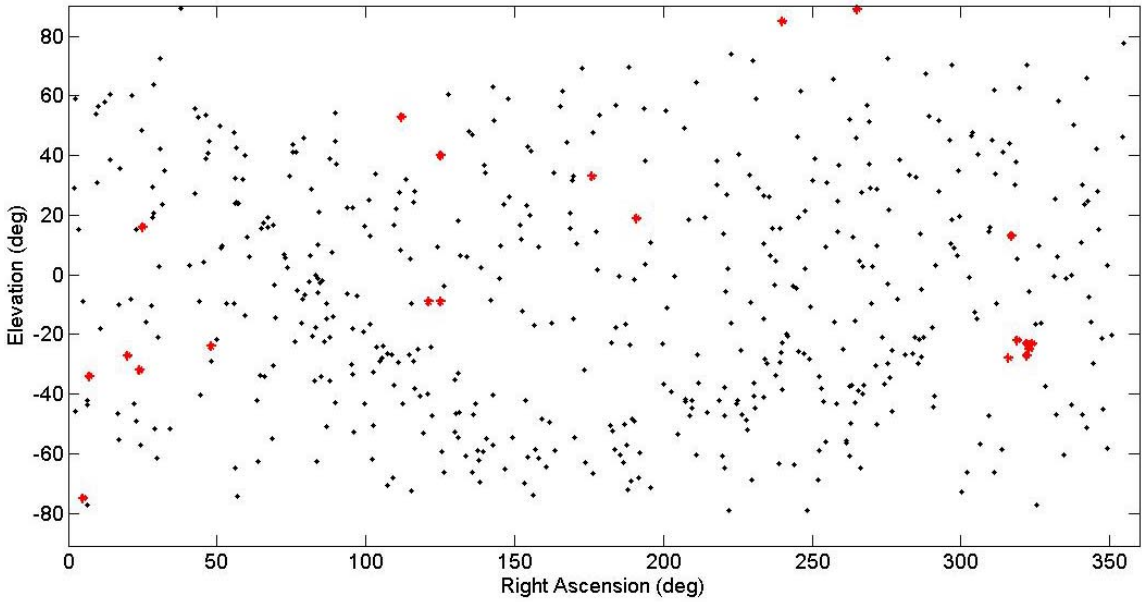

Figure 5.3: Stellar Map w/ Instances of No Match Found

The red stars represent the bore sight of the camera. From this map it could be seen that when no match was found, typically the orientation was pointing at a portion of the sky where there were very few stars. What this implied was that "No Match" was the result only when four or five stars were on the image and there was enough error on the angular calculations to prevent the algorithm from finding a correct match. Therefore it can be inferred that the more stars there are on the image, the more likely that a correct match will be found.

5.3.3 Not Enough Stars

The final way that the star tracker could fail to find a correct match was if not enough stars were present on the image. Conceptually, it was easy to understand why there wouldn't be enough stars on the image: because that particular orientation pointed at a relatively empty portion of the sky. But the question was raised, "Are there particular

regions in this map which are relatively empty?" Figure 5.4 once again displays the 2-D stellar map plotted with the orientations of the occurrences when not enough stars were on the image. As before, the red stars on the map represent the bore sights for the instances in question.

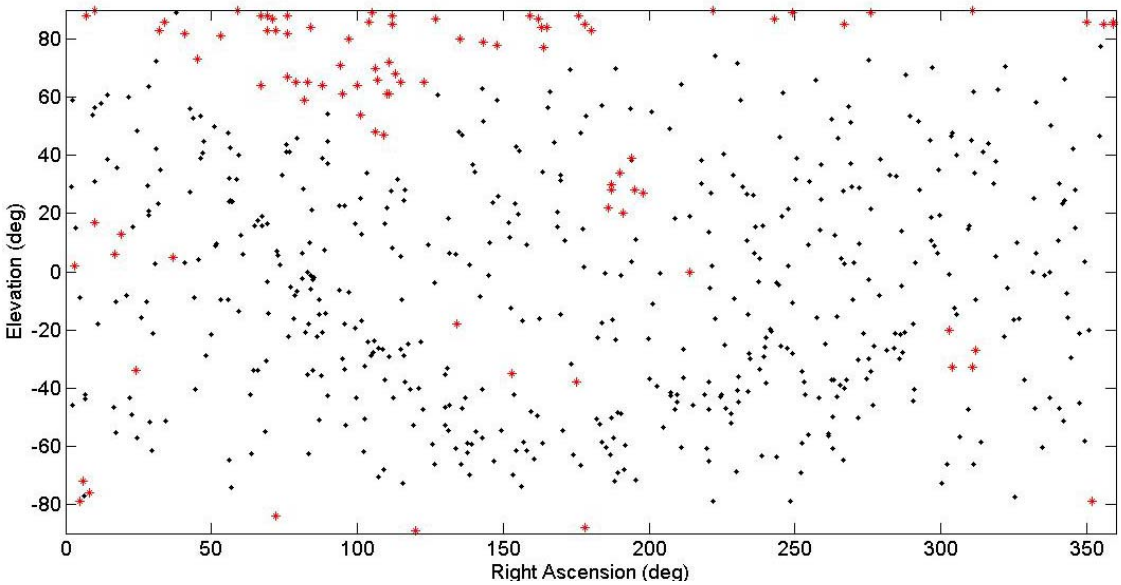

Figure 5.4: Stellar Map w/ Instances of Not Enough Stars

From this map it could be seen that 70 of the 92 times when not enough stars were found was when the camera was pointing in the polar regions (primarily the North Pole). This made sense because by limiting the number of stars in the catalog, the polar regions were left with very few stars.

5.4 Parameter Tuning

Several of the parameters were varied for the Vector Angle method in order to find a set of parameters that resulted in the best performance of the algorithm. The parameters which were varied were: field of view, search tolerance, and image array size. The

following bar charts in Figure 5.5 display the results of the parameter tuning process. The search tolerance was varied to be 0.4 degrees, 0.2 degrees, and 0.1 degree. The simulations were also run with image array sizes of 1280 x 1024, and 640 x 480.

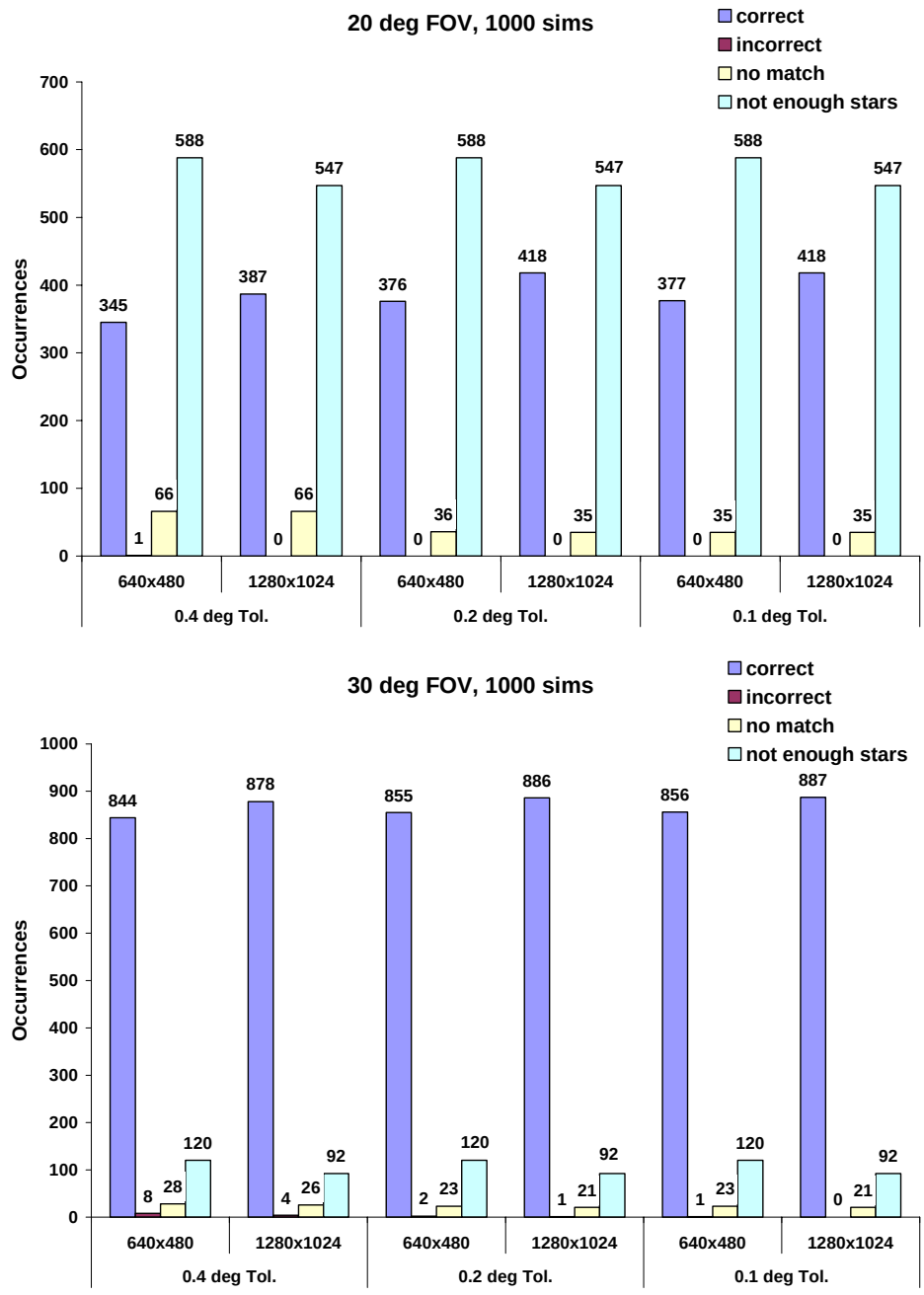

Figure 5.5: Parameter Tuning Results for 20 deg (top), and 30 deg (bottom) FOV

The top graph shows the results for a field of view of 20 degrees. It can be seen that by narrowing the field of view, the instances where not enough stars are present increases drastically from the previous simulation. Over 50% of the simulations didn't have enough stars in the image. This would not be ideal if an actual star tracker were being developed. Narrower fields of view are used by many of the bigger, higher powered star trackers. However those devices have extremely sensitive imagers, know what should be seen based upon attitude propagators, and also use the magnitudes of the stars within the search. The 20 degree field of view resulted in nearly no incorrect matches. But the fact that it so clearly limited the number of times a match could be found due to not enough stars killed the chance of 20 degrees being the optimum field of view for this simulation.

Upon analyzing the simulations with a 30 degree field of view, it could clearly be seen that the number of times when not enough stars were present was extremely less than with 20 degree field of view. However the number of times when an incorrect match was found was higher; most likely due to the fact that there were more stars present, which resulted in more opportunities for failure to occur. As the tolerance crept from 0.4 to 01 degrees, the performance of the algorithm steadily improved. Each time the tolerance decreased, the number of correct matches increased, and the number of incorrect and no matches decreased or stayed the same. The smaller imager size resulted in several more instances when not enough stars were available. This was caused by the aspect ratio of the two imagers not being equal, resulting in a few degrees of field of view that were available to the 1280 x 1024 imager, not being available to the 640 x 480 imager. Generally, the larger imager performed better than the smaller imager, which was

attributed to the fact that the larger imager had a high angular resolution than the smaller imager. The larger imager had approximately 0.023 degrees per pixel, while the smaller imager had only 0.044 degrees per pixel.

From this analysis it was decided that using the Vector Angle method with a tolerance of approximately 0.2 or 0.1 degrees, an imager size of 1280 x 1024 pixels, and a field of view of 30 degrees was the optimum set of parameters for the simulation. These were the parameters that would be carried over to the fixed point implementation.

CHAPTER 6: FIXED POINT IMPLEMENTATION

6.1 Changes to the Algorithm

6.1.1 No Trigonometric Functions

In order to implement the algorithm in fixed point, all trigonometric functions had to be removed from the code. The only trig function used was the arccosine function. This was used when determining the angle between two stellar vectors as shown in the equation in Section 4.5. Two possible solutions to this problem were evaluated and discussed below.

6.1.1.1 Storing Cosines of the Angles

The first method that was evaluated was building the vectors in the same way as before, but instead of taking the arccosine of the dot product, the end values would simply be the cosines of the angles as shown in the equation below.

$$cos\,\theta = \frac{\left(\vec{V}_1 \cdot \vec{V}_2\right)}{\left\|\vec{V}_1\right\| \cdot \left\|\vec{V}_2\right\|}$$

This meant that the Look-Up table would have to be rebuilt with the cosines instead of the angles themselves. The range of the Look-Up table can be seen below in Table 4 in the right column, with the old angles associated with the cosines in the left column.

Table 4: Look-Up Table Range for Cosine Storage

Angle	Cos(Angle)
0	1
⋮	⋮
30	0.866

The way that the table was sorted would also have to be changed. The cosines of the angles which defined each triad were sorted horizontally in descending order, and then the entire table was sorted in descending order based upon the first column.

Table 5: Format of the Look-Up Table w/ Cosines

cos(Angle)1-1	cos(Angle)1-2	cos(Angle)1-3
cos(Angle)2-1	cos(Angle)2-2	cos(Angle)2-3
…	…	…
cos(Angle)(n)-1	cos(Angle)(n)-2	cos(Angle)(n)-3

$$\cos(\text{Angle1-1}) > \cos(\text{Angle1-2}) > \cos(\text{Angle1-3})$$
$$\cos(\text{Angle1-1}) > \cos(\text{Angle2-1}) > \ldots > \cos(\text{Angle(n)-1})$$

The search operated in the same manner as it had done in the previous algorithms, except a few modifications had to be made in order to account for the fact that the Look-Up table now was in descending order instead of ascending order.

6.1.1.2 Angles from Pixel Distance & Angular Resolution

The second method used to calculate angles between vectors was to use the distance between the stars on the image, and the angular resolution of the imager. For this method,

the assumption had to be made that the angular resolution was constant across the imager in terms of degrees per pixel. That of course was not true because the angular resolution of the pixels towards the edge of the imager would be less than those in the middle, but the difference was assumed to be negligible.

The change between two stars in the x- and y-direction on the image plane could be calculated by simply subtracting the x- and y-locations of the stars on the image. Then the changes were multiplied by the angular resolution to get the changes in location represented in degrees. By then taking the square root of the sum of the squares of those changes in location, the total angle between the two stars could be found. Figure 6.1 below displays the process used.

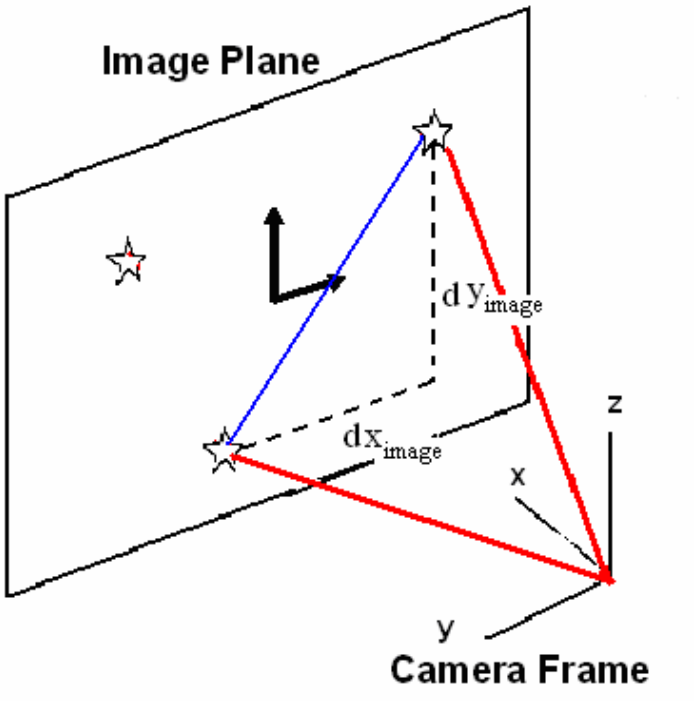

Figure 6.1: Pixel & Angular Resolution Method

The two changes in location discussed above are represented by dx_{image} and dy_{image} in the figure above. The blue line is the total pixel length which was approximately proportional to the angle between the two vectors. In order to calculate the angle, the individual pixel lengths of dx_{image} and dy_{image} were multiplied by the angular resolution, which has units of degrees per pixel. This is shown in the equation below.

$$\theta_x(\text{deg}) = dx_{image} \times \text{Angular Resolution(deg/pixel)}$$
$$\theta_y(\text{deg}) = dy_{image} \times \text{Angular Resolution(deg/pixel)}$$

Then the square root of the sum of the squares of θ_x and θ_y was calculated as displayed below. This value was approximately the value of the angle between the two stellar vectors.

$$\theta_{total} = \sqrt{\theta_x^{\,2} + \theta_y^{\,2}}$$

6.1.2 No Square Root Functions

The other significant function that needed to be removed from the algorithm was the square root function. This function was used in two places in the algorithm. The first instance it was used is during calculation of the angles between the vectors. It was used differently for the two methods described in Section 6.1.1.1 and Section 6.1.1.2. The square root was also used in quaternion extraction.

Polynomial functions were used to fit every square root used in the algorithms. These polynomial fits used the Least-Squared Method. In this section the variable ‘x’ refers to the input of the square root function.

6.1.2.1 Storing Cosine Method

The first method, which used cosines instead of the angles to define a stellar triad, uses the square root function to normalize the vectors when calculating the arccosine. The dot product of the two vectors was divided by the product of the magnitudes of the vectors. The magnitudes were calculated by taking the square root of the sum of the squares of the components of each vector.

A polynomial was used to fit the curve for the square root function. The limits of the curve fit were set to 1 and 1.2. This was because the minimum magnitude of a stellar vector would be 1, which happened when the star was in the center of the image. The maximum magnitude would be at the corner of the image and would be approximately 1.12. The polyfit.m function in MATLAB produced the following third-order polynomial.

$$sqrt(x) \approx 0.0559x^3 - 0.2929x^2 + 0.9181x + 0.3189$$

When this polynomial was compared to the actual square root function, the error was extremely small. The error was plotted across the range of x and can be seen below in Figure 6.2.

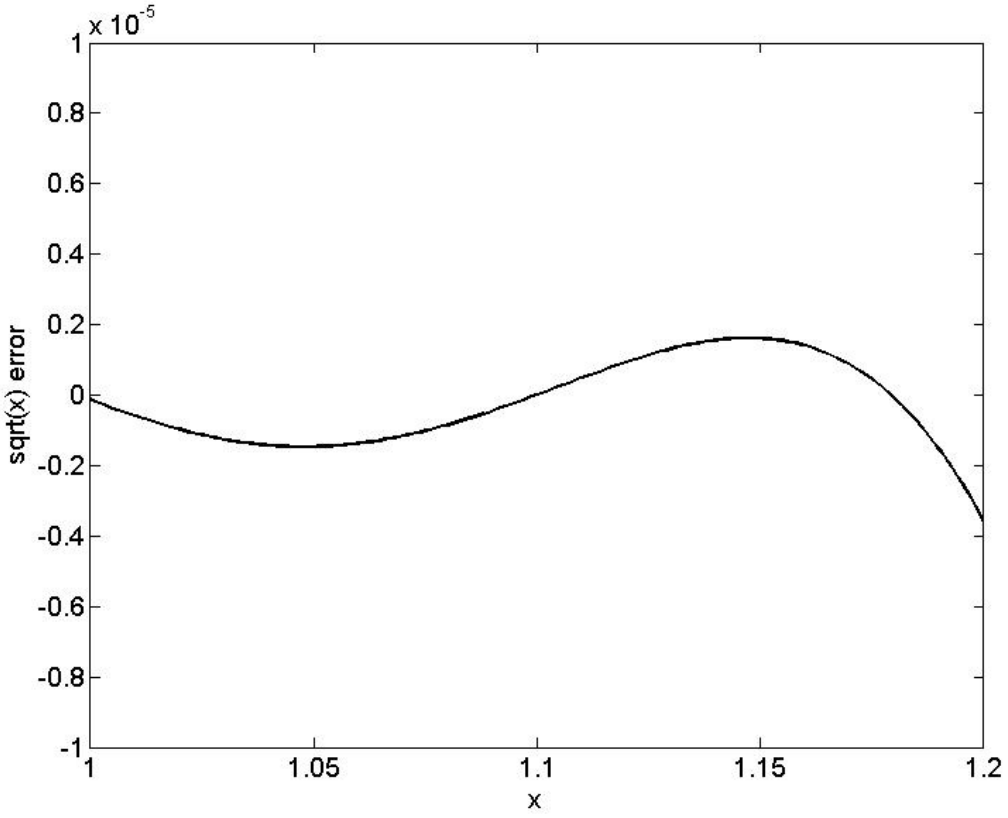

Figure 6.2: 3rd Order Polynomial Fit for Square Root for Cosine Storing Method

6.1.2.2 Pixel Distance and Resolution Method

The second method used to calculate the angles between stellar vectors was to multiply the pixel distances by the resolution of the imager. What this method did was allow distances on the imager itself to represent the angles. However, in order to find the total angle between two stellar vectors using this method, it was necessary to use the square root function when taking the sum of the squares of θ_x and θ_y as described above in Section 6.1.2.1. These angles were calculated and converted to radians. Therefore a polynomial fit would have to be made for the square root function which ranged from x =

0, to x = 0.6. The limit of this range was selected because the maximum angles allowable in the algorithm were 30 degrees. In radians, 30 degrees is 0.5236 rad. It was assumed that the maximum θ_x and θ_y could be were 30 degrees each. Then the maximum value that could be sent into the square root function would be 0.5483 rad^2. So 0.6 rad^2 was used for margin.

It was soon found that a single polynomial fit would not work due to the fact that it could not converge well near x = 0. Therefore two polynomials were used to make this method work. The first was for when x was less then 0.1, and the second was for x greater than 0.1. Fourth order polynomials were used for both functions. The resulting polynomials can be seen below.

$$x < 0.1 \qquad sqrt(x) \approx -1407.12x^4 + 524.788x^3 - 73.656x^2 + 6.287x + 0.038$$

$$x > 0.1 \qquad sqrt(x) \approx -4.098x^4 + 7.008x^3 - 4.844x^2 + 2.355x + 0.121$$

As before, the difference between these two functions and the true polynomial function was plotted. This can be found below in Figure 6.3.

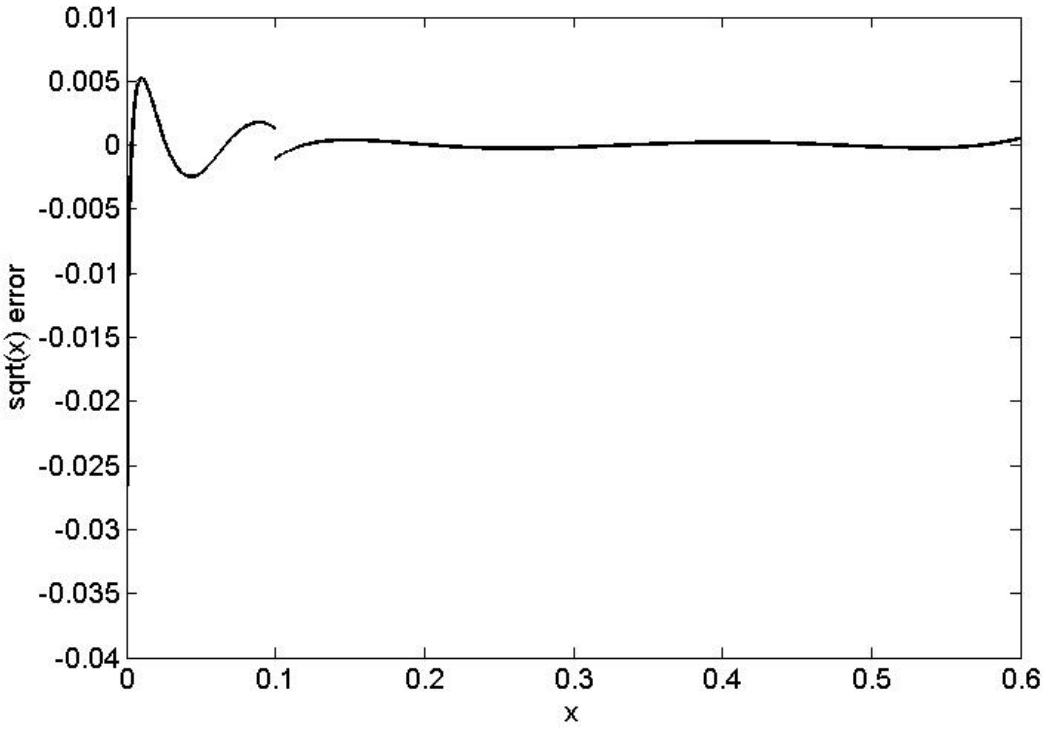

Figure 6.3: 4th Order Polynomial Fit for Square Root for Pixel Method – 2 polynomials

The jump in the figure is at x = 0.1, when one function ends and the other takes over. It can be seen that when x was greater than 0.1, the error is much less than if x were less than 0.1.

6.1.2.3 Quaternion Extraction

The square root function was also used in the process of quaternion extraction as can be seen in the equations of Section 4.5.

$$q_4 = \pm 0.5\sqrt{1 + a_{11} + a_{22} + a_{33}}$$

For the case when the ECI coordinate system and the camera frame were nearly aligned, the diagonals of the transformation matrix (a_{11}, a_{22}, and a_{33}) would all be nearly 1. Therefore the range of the input to this square root function had to be from zero to

approximately 4. Once again a polynomial function was used in place of the square root function. The same two were used as in the previous section. A third polynomial function was added to take the range of x from 0.6 to 4. The polynomial function for this range is the bottom of the three equations found below.

$$x < 0.1 \qquad sqrt(x) \approx -1407.12x^4 + 524.788x^3 - 73.656x^2 + 6.287x + 0.038$$

$$0.1 < x < 0.6 \qquad sqrt(x) \approx -4.098x^4 + 7.008x^3 - 4.844x^2 + 2.355x + 0.121$$

$$0.6 < x < 4 \qquad sqrt(x) \approx -0.0035x^4 + 0.0427x^3 - 0.2190x^2 + 0.8260x + 0.3526$$

Again, the error for the overall set of functions was plotted and can be found below in Figure 6.4.

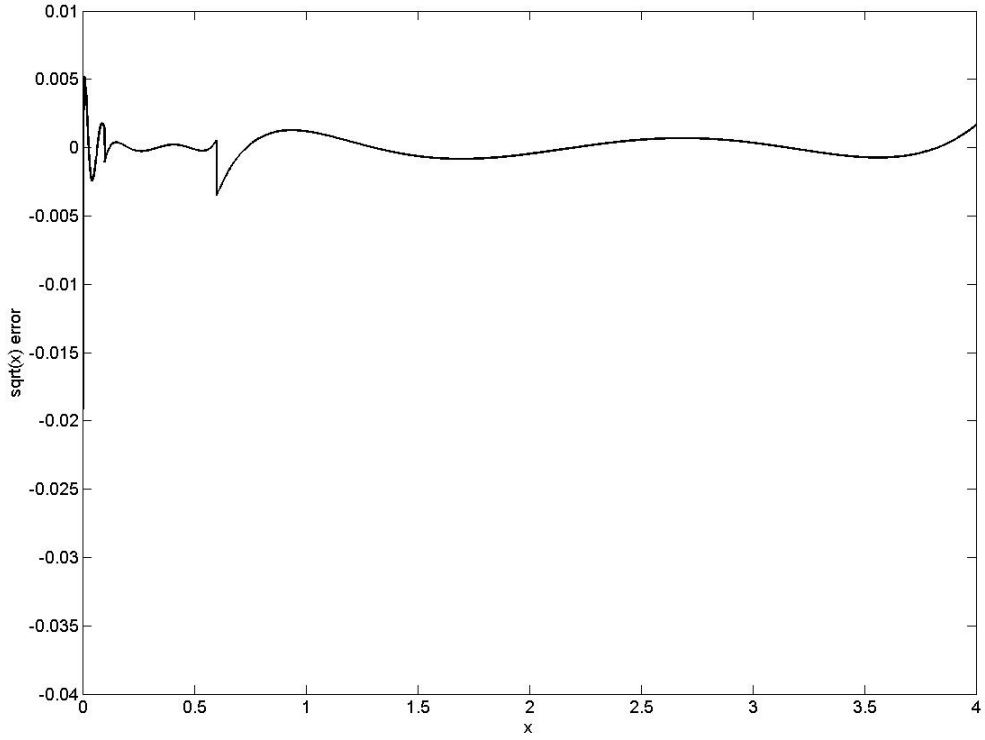

Figure 6.4: 4th Order Polynomial Fit for Square Root for Quaternion Extraction – 3 polynomials

As before, the sharp jumps on the graph represent where the functions end and begin, resulting in different initial errors. Overall the process of using several polynomials to

represent the entire square root function provided much more accurate results than a single polynomial function would have. Evidence of this can be found below. Figure 6.5 shows what the result of a single polynomial would have given for the range of x being 0 to 4.

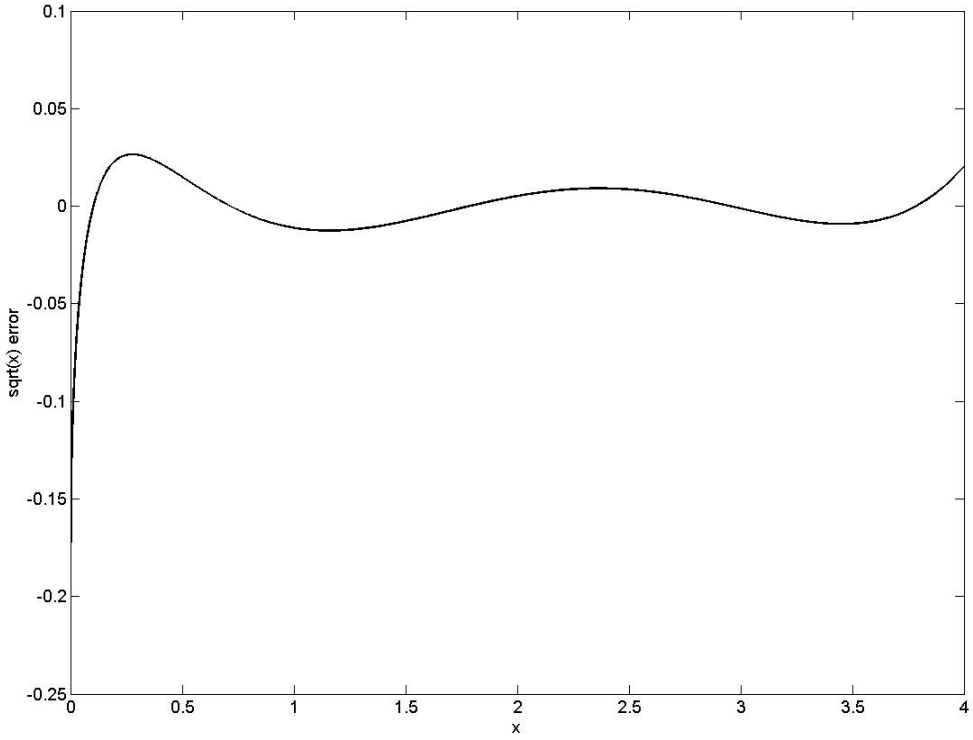

Figure 6.5: 4th Order Polynomial Fit for Square Root for Quaternion Extraction – 1 polynomial

The error was two orders of magnitude greater when a single polynomial was used, validating the idea that multiple functions provides more accurate results.

6.2 Fixed Point Location

The final step in preparation for implementing the algorithm into fixed point was to define the fixed point location. After a thorough search of the algorithm, it was determined that the highest integer value which would ever be used was 2388. This was the conversion factor which put the camera frame in the units of pixels. This value required 12 bits to be devoted to integers alone. If one bit was devoted to sign as well, only 3 bits would be left to devote to fractions on a 16-bit processor. This would not be sufficient given the level of accuracy required for this algorithm. Therefore, 32-bit processing was used for the simulations. This allowed 19 bits to be devoted to the fraction which was more than enough to provide the accuracy required.

6.3 MATLAB Fixed Point Toolbox

MATLAB has a fixed point toolbox which can be used to convert floating point operations into fixed point. MATLAB has to emulate the fixed point process, which made it very time consuming to run Monte Carlo simulations.

A script was written to convert all desired variables to fixed point. The script inputted variable values and outputted the values in fixed point with the 32-bit limit, and 19 integers devoted to the fraction. The ‘fi’ function in MATLAB was used to perform this operation, and can be seen below.

```
y = fi(VAR,1,32,‘PROPERTY’,VALUE,...)
```

The term VAR represented the variable value in floating point which was to be converted. The ‘1’ signified that positive and negative signs had to be maintained. The ‘32’ told the function that the maximum bit-length available was 32. There were several different properties which were added to the function. The properties and their values can be found below in the order that they were used.

FractionLength:	19
ProductWordLength:	32
ProductMode:	(No Value Necessary)
SpecifyPrecision:	(No Value Necessary)
SumWordLength:	32
SumMode:	(No Value Necessary)
SpecifcyPrecision:	(No Value Necessary)
ProductFractionLength:	19
SumFractionLength:	19

6.4 Fixed Point Simulation Results

Both of the methods described above were implemented into floating point and fixed point so that a proper comparison could be made.

6.4.1 Storing Cosines Method

The first method to be evaluated was the method involving building the vectors, searching and matching using the cosines of the angles rather than the angles themselves, in order to avoid using the arccosine function. Initial thoughts were that this method would produce the best results due to the fact that it involved building the actual vectors, and no false assumptions were made as is done in the second method. Nonetheless it was expected that some performance would be lost due to the fixed point limiting the

accuracy of the fraction. The results of the first method can be found below in Figure 6.6. Only 100 simulations were run for this case because MATLAB required much processing time in order to emulate fixed point.

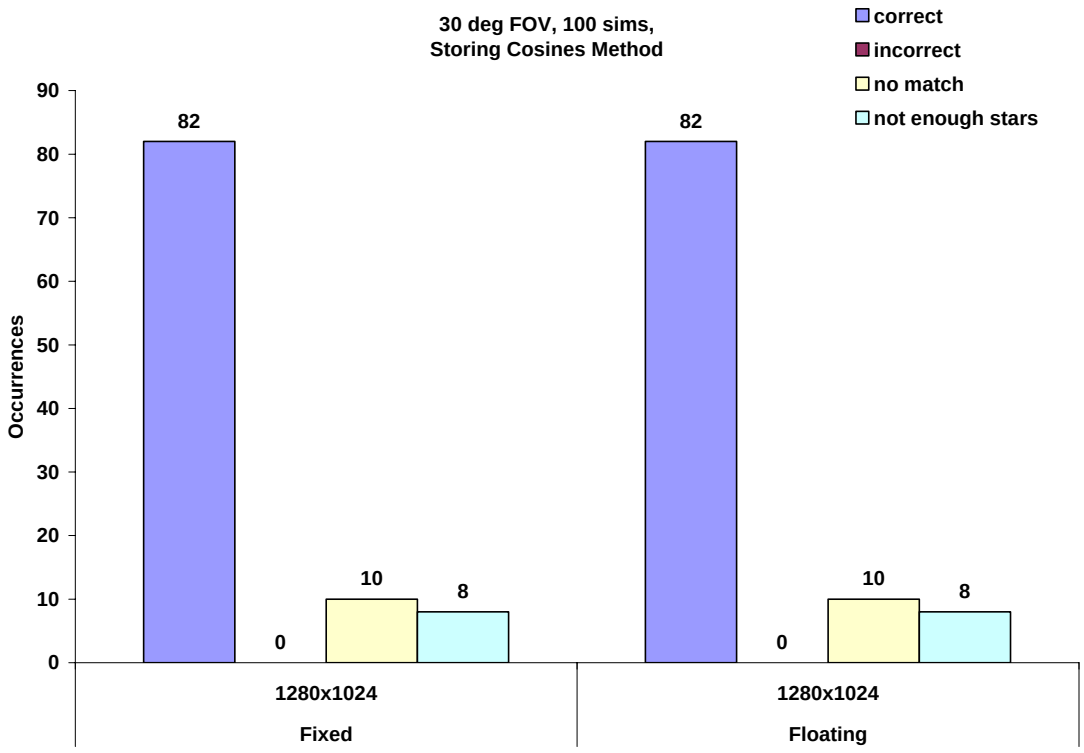

Figure 6.6: Fixed vs. Floating Point – Storing Cosines Method

The assumption that accuracy would be lost due to fixed point implementation was found to be erroneous. This method *did* however prove to be less accurate than the original method of using the actual angles between the vectors. Evidence of this was the fact that when the actual vectors were used, only about 2% of the simulations resulted in no match being found. However when the cosines were used instead of the angles, the number jumped to about 10%. The jump may be attributed to the fact that the range for this method is extremely small. The cosines of the angles range from 0.866 to 1. When the

actual angles were used, the range was 0 to 30 degrees. This may have made it easier to search through the Look-Up table.

Once cosines were implemented instead of angles, though, this method performed exactly the same in fixed point as it did in floating point. The same numbers of instances for all four categories occurred in both fixed and floating point. The average right ascension, elevation, and roll error can be found in the Table 6 below.

Table 6: Avg. Angular Errors for Storing Cosines Method

	Right Ascension (deg)	Elevation (deg)	Roll (deg)
Fixed Point	0.041	0.016	0.097
Floating Point	0.039	0.022	0.097

Nearly the same values for each angle were found for both fixed and floating point.

6.4.2 Pixel Distance and Resolution Method

When the second method was implemented, the results were much less similar than in the previous method. There were a significant number of times that the floating point found a match that the fixed point algorithm did not. The overall results can be found below in Figure 6.7.

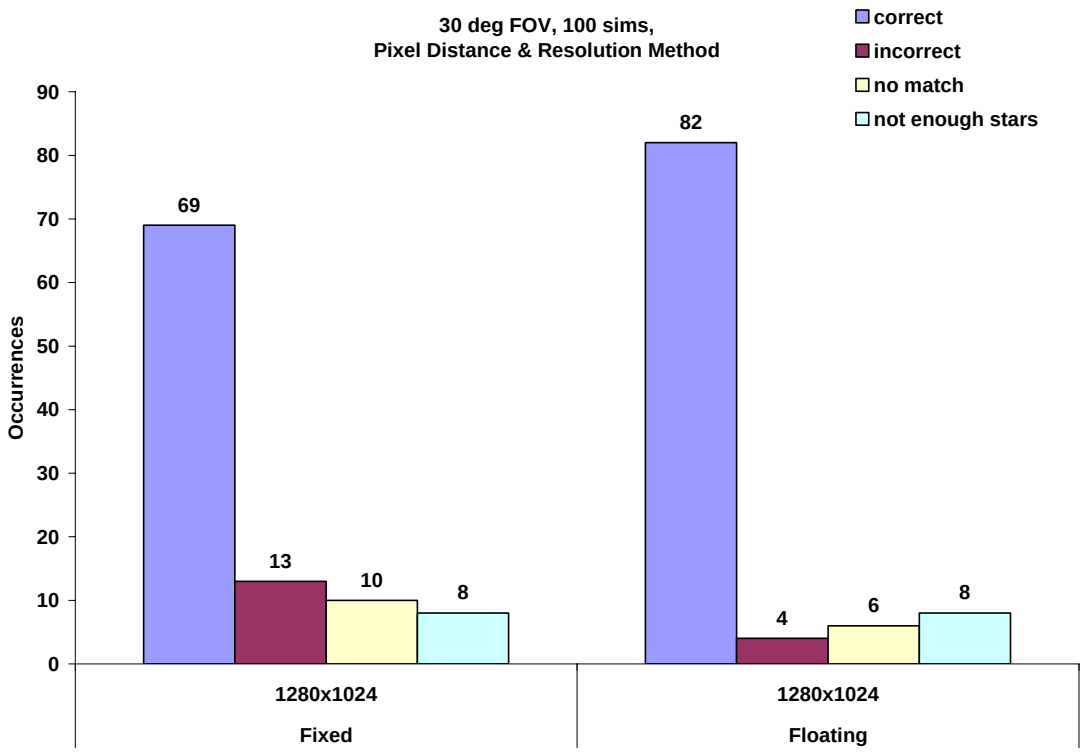

Figure 6.7: Fixed vs. Floating Point - Pixel/Resolution Method

From the figure above it can clearly be seen that the number of correct matches dropped by a count of 13; of which nine became incorrect matches and four became no matches. Table 7 below shows the average angular errors for both fixed and floating point.

Table 7: Avg. Angular Errors for Storing Cosines Method

	Right Ascension (deg)	Elevation (deg)	Roll (deg)
Fixed Point	0.030	0.021	0.050
Floating Point	0.013	0.004	0.047

The errors were less in floating point, which was expected. However the fact remains that too many accurate matches were lost using this method.

This drop in performance was expected because the method itself was the more inaccurate of the two methods. It was based on a false assumption that the angular resolution of the imager was constant.

6.5 Failure Analysis

As was done in floating point, the instances when an accurate match was not found were analyzed for the fixed point implementation. Only the method which was implemented using cosines of the angles was evaluated in this section.

Figure 6.8 shows the 2-D map of the stars with the bore sights highlighted in red for the instances when not enough stars were present on the image.

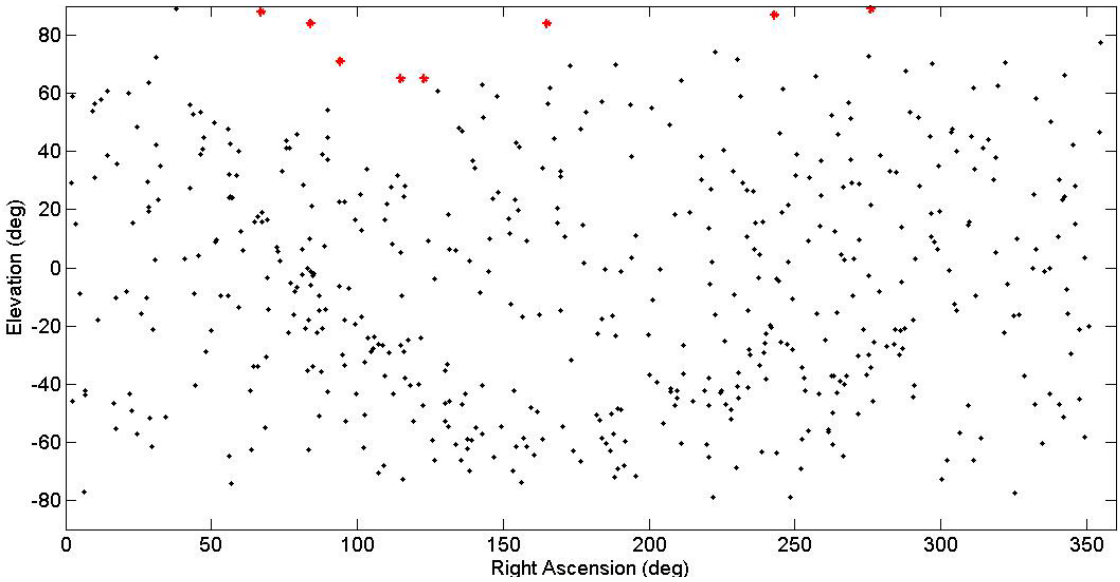

Figure 6.8: 2-D Stellar Map w/ Instances of Not Enough Stars Found in Fixed Point

As was the case when the algorithm was in floating point, the only times when not enough stars were present was in the polar regions. This was expected because the same random orientations were used throughout every simulation. If the field of view and imager aspect ratio did not change, then the instances when not enough stars were present should have been the same as well. There were fewer instances in this figure than in Figure 5.4 because only 100 simulations were run for fixed point as opposed to the 1000 run for the floating point.

Figure 6.9 contains the 2-D stellar map with the instances when no match was found highlighted in red.

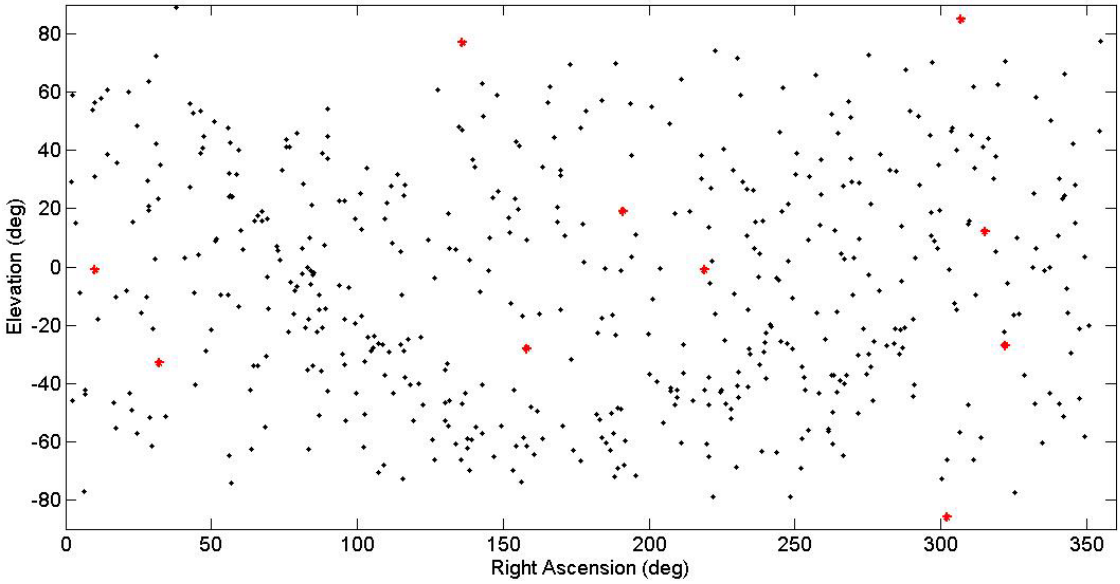

Figure 6.9: 2-D Stellar Map w/ Instances of No Match Found in Fixed Point

As the figure shows, the times when no match could be found all occurred when the camera was pointing in the direction of a relatively empty piece of sky. This meant that at least four stars were present on the image, but the values being extracted for the cosines of the angle were not accurate enough to find a match. This was also seen in Figure 5.3 when the failures were analyzed in floating point.

CHAPTER 7: CONCLUSION

7.1 Summary

This thesis presented two methods for implementing a star tracker pattern matching and quaternion-extraction algorithm in fixed point. Both methods avoided the use of trigonometric functions and square roots, which were the primary functions to be avoided. Initially three different pattern matching algorithms were evaluated in floating point, in order to compare performance and complexity. Tools such as a star catalog, look-up tables, and a camera/centroiding emulator were developed. The least complex pattern matching algorithm was found to be the one which had the best performance. This method used angles between stellar vectors to define stellar triads. Accurate star matches were found from a look-up table 886 times out of 1000 simulations. Using angles between the vectors was selected as the best candidate algorithm form implementation into fixed point.

Two techniques were developed for determining the angles between stellar vectors on an image using fixed point processing. The first involved taking the dot product of the two stellar vectors, and then performing a search based on the cosines of the angles between the vectors rather than the angles themselves. The second technique developed was to multiply the distances between stars on the image by the average angular resolution of the imager. This provided a rough idea of the angle between the vectors. This was the

simpler of the two methods, because it only involved multiplying pixel distances by a scalar value.

Of the two techniques evaluated, the first method performed the best. By calculating the cosines of the angles, this method avoided the use of any trigonometric functions. It also used the square root function fewer times than the second method. Over 100 simulations it found 13 more accurate matches than the other method. Both methods yielded fairly accurate results regarding the error in right ascension, elevation, and roll. These values are three to four orders of magnitude greater than commercial star trackers available to the larger satellites. Nonetheless, pointing knowledge of less the 0.1 degrees for any picosatellite would be a notable accomplishment – which this algorithm was found to be capable of providing.

7.2 Recommendations for Future Work

7.2.1 Centroiding

Most of the commercial star trackers are capable of centroiding stars on an image to less than a pixel of accuracy. If it were found that this could be done for a simple imager that could be used onboard a picosatellite, then the accuracy of the vectors, and angles would improve, making the algorithm even more accurate.

7.2.2 Develop a Smarter Way of Performing the Search

In order to make the search more efficient, several things could be added to the overall architecture. To decrease the possibility of incorrect or no matches found, an onboard orbit propagator could be implemented. This would allow the algorithm to throw out half of the stars right off the bat, by knowing which side of the Earth the vehicle is on. In doing so, it would be known that the Earth is blocking the view of nearly half of the stars, thus allowing half of the table to be thrown out for any given search.

Another way of performing a more optimized search would be to know approximately where the vehicle is pointing before the image is taken. The algorithm could be able to look at the image and know exactly what stars it should be looking for, rather than blindly going about it in the "Lost-in-Space" manner. This could be done by adding more sensors, and implementing an onboard Extended Kalman Filter (EKF). An EKF would allow the satellite to more accurately estimate its attitude prior to star tracker operation.

In order to decrease the number of instances when not enough stars were in the field of view, more stars could be added to the catalog. By adding some of the dimmer stars the chances of finding correct stellar matches in the polar regions would increase. However by adding more stars to the catalog two assumptions would have to be made: that the satellite could store an even larger look-up table; and the imager would be sensitive enough to find the dimmer stars in the sky. Both of these assumptions would have to be carried out once hardware was brought into the picture.

7.2.3 Use Computer Science Techniques to Mature Algorithm

As was discussed earlier, the methods developed for searching and sorting throughout this thesis were very rudimentary, and ran in a "brute force" style. They were not efficient when it came to operational speed; nevertheless the tasks that they were required to do were completed. If a computer scientist were to sit down and dig through the searching and sorting algorithms, there is little doubt that the code could be drastically improved and become more efficient. In doing so the overall operating time for the star tracker would decrease. This may allow the star tracker to operate in some kind of loop such as an EKF, which would provide an extremely accurate way of estimating the vehicle's attitude and rates.

7.2.4 Implement Onboard Processor/Spacecraft

Once the algorithm is completely optimized, the end goal would be to put it onboard a small processor which operated in fixed point. On that platform, simulations could better describe how the algorithm will actually perform, especially if camera hardware were put into the system as well. Once the processor and camera have been setup and the algorithm operates as it should, the system could be integrated into a small satellite for on-orbit validation. The results from orbit could be compared to the results on the ground using MATLAB, if the pictures were capable of being downloaded.

LIST OF REFERENCES

Cole, C.L., Crassidis, J.L., "Fast Star Pattern Recognition Using Planar Triangles", *Journal of Guidance, Control, and Dynamics*, January 2006

Liebe, C.C., "Pattern Recognition of Star Constellations for Spacecraft Applications", Technical University of Denmark, *IEEE AES Magazine*, June 1992

Mortari, D., Junkins, J., Samaan, M., "Lost-In-Space Pyramid Algorithm For Robust Star Tracker Pattern Recognition", Taxas A&M University, 24th Annual AAS Guidance and Control Conference, January 2001

Shucker, B., "A Ground-Based Prototype of a CMOS Navigational Star Camera for Small Satellite Applications", University of Arizona, 15th Annual AIAA/USU Conference on Small Satellites, August 2001

Sidi, M.J., *Spacecraft Dynamics & Control: A Practical Engineering Approach*, Cambridge University Press, 2002.

Sturm II, E.J., "Magnetic Attitude Estimation of a Tumbling Spacecraft", California Polytechnic State University, 2005

Wertz, J.R., (ed.), *Spacecraft Attitude Determination and Control*, Kluwer, 2005.

Wertz, J.R., Larson, W.J., *Space Mission Analysis and Design*, 3rd Edition, Microcosm Press, 1999

Zenick, R., "Lightweight, Low-Power Coarse Star Tracker", AeroAstro, 17th Annual AIAA/USU Conference on Small Satellites, August 2003.