

# Embedding a Star Tracker onto A Microcontroller

*Tom Creusot*

School of Civil & Mechanical Engineering Project Thesis  
Curtin University

24/11/2023

## **Acknowledgments**

I would like to thank my supervisor A/Prof Jonathan Paxman for all his help and advice during this thesis. I also would like to thank my research team Ovik Choudhury and Alena Panferova for designing the hardware for the star tracker. Lastly I would like to thank the Space Science and Technology Centre at Curtin University and the BINAR team for providing me this opportunity.

## **Preface**

The Star Tracker project already existed prior to the commencement of this thesis. Prior and current work is disclosed throughout the paper. All work on the Star Tracker software before and during the thesis was done by the author, T. Creusot and he alone.

Some work in this paper is a direct copy or modified work from a progress report written during the running time of the thesis.

The hardware of this project was mainly designed and chosen by other students including A. Panferova and O. Choudhury. Choices influenced by the author are included in this project including choices on hardware prior to the beginning of the thesis.

## **Abstract**

A Star Tracker is a highly accurate form of Attitude Determination used on spacecraft. This device can provide orientation and pointing positions to sub-degree accuracies. This thesis investigates an open-source star tracker using the Pyramid tracking algorithm. Through investigation, it was found that the Pyramid method would not run effectively without implementing any optimisations or considering the target platform.

With the request of BINAR, a micro-satellite development group based in Perth, this algorithm was to be implemented on a functioning satellite. This thesis is an in-depth analysis of modifications and optimisations required to prepare a star tracker for a mission.

## **Nomenclature**

PC	Personal Computer
FOV	Field Of View
MB	Megabyte
KB	Kilobyte
3D	3-Dimensional
2D	2-Dimensional
IR	Infrared

## Table Of Contents

<b>Acknowledgments.....</b>	<b>2</b>
<b>Preface.....</b>	<b>2</b>
<b>Abstract.....</b>	<b>3</b>
<b>Nomenclature.....</b>	<b>3</b>
<b>Table Of Contents.....</b>	<b>4</b>
1.0 Introduction.....	6
2.0 Literature Review.....	7
2.1 Universities and Cubesats.....	7
2.3 Attitude Determination Methods.....	8
2.4 Star Tracking Methods.....	9
2.5 Voting Methods.....	11
2.6 Background.....	13
2.6.1 Image Acquisition.....	14
2.6.2 Image Processing.....	14
2.6.3 Projection.....	16
2.6.4 Tracking Mode.....	18
2.6.5 Database.....	20
2.6.6 Voting.....	22
2.6.7 Software.....	22
2.6.8 Hardware.....	24
<b>3.0 Technical Design Work.....</b>	<b>26</b>
<b>3.1 Aim.....</b>	<b>26</b>
3.2 Lens.....	26
3.3 Tests.....	28
3.3.1 Simulation Test.....	28
3.3.2 Demo Test.....	28
3.3.3 Integration Test.....	29
3.3.4 Absolute Magnitude.....	29
3.3.5 Initial Performance.....	29
3.4 Image Processing.....	30
3.4.1 Thresholding.....	30
3.4.2 Blob Detection.....	32
3.4.3 Skipping.....	33
3.5 Projection.....	34
3.5.1 Lens Distortion.....	34
3.6 Pyramid Method.....	37
3.6.1 Identifying Bottlenecks.....	37
3.6.2 Star Triangle Iterator.....	37
3.6.3 Region Reduction.....	38
3.6.4 Field of View Reduced.....	39
3.6.5 Chunk Iterator.....	40
3.6.6 Chunk Area Search.....	42
3.7 Platforms.....	43

3.7.1 Raspberry Pi.....	43
6.7.2 Embedded Systems.....	43
3.7.3 Hardware Abstraction Layer.....	44
3.7.4 Memory.....	45
3.7.5 DCMI.....	47
3.7.6 Multi-Core.....	47
3.8 Project Lifetime.....	47
<b>4.0 Results.....</b>	<b>48</b>
4.1 Sky Wipe.....	48
4.2 Test Parameters.....	49
4.3 PC.....	50
4.4 Raspberry Pi 3B.....	51
4.4.1 600MHz.....	51
4.4.2 1200MHz.....	52
4.5 Microcontroller.....	53
4.6 Analysis.....	54
<b>5.0 Conclusion.....</b>	<b>55</b>
<b>6.0 Future Work.....</b>	<b>56</b>
Hardware.....	56
Software Implementation.....	56
Embedded System.....	56
<b>7.0 References.....</b>	<b>57</b>
<b>Appendix A - Sky coverage.....</b>	<b>61</b>
<b>Appendix B - Percent Four Star Sky Coverage.....</b>	<b>62</b>
<b>Appendix C - Setting up Allied Vision Python480 camera for set and forget.....</b>	<b>63</b>
<b>Appendix D - Setting up star tracker library on Raspberry Pi.....</b>	<b>66</b>

## 1.0 Introduction

On Earth, a person is capable of finding their orientation from a variety of sensory cues. The main way of doing this is through using the acceleration of gravity or the inclination of the horizon to find the plane perpendicular to up. They can also find their facing angle on the plane from the sun's position, by using a compass to measure the magnetic fields or by identifying constellations and comparing them with their known positions. When designing autonomous vehicles, it can be an art to make them accurately predict their own orientation. Engineers will tend to combine many of these inputs together to create an accurate map to increase the accuracy and reliability of the vehicle. This process is known as attitude determination.

In space, without the aid of gravity, it can be difficult to provide an accurate map of a body's orientation. When a body is in a low orbit, a *horizon sensor* can be used to find the Earth's plane, when the body is not in shade, a *sun sensor* can be used as a basic reference marker. When the body is within a known magnetic field, a *magnetometer* can be used to provide a vector towards the magnetic North pole. In all of these scenarios, the sensors can only function in specific circumstances and may only provide part of the required information for the body's attitude. A sensor that can accurately find the orientation of a satellite regardless of where and when it is, is the *star tracker*.

A *star tracker* is a device that takes a photo of a set of stars and runs an algorithm that can find unique patterns to identify them. Using the identified stars, a highly accurate attitude can be found. For payloads requiring precise attitude, a sensor like this can be desirable over other sensors as it does not require prior information to run, it has less downtime compared to other sensors and it is highly accurate with its results.

With the introduction of the standardised 1U cubesat, a 10x10x10cm, 1kg satellite, and the cost to launch depreciated, the barrier to entry has decreased. Because of this, space has become an opportunity for enthusiasts to satisfy their hobby and for Universities trying to engage a future generation.

Curtin University, located in Perth, Western Australia, has taken an interest in having access to this new frontier and has set up the group BINAR to design cubesats. With limited funds and a lack of space in the satellite, this team has avoided buying off-the-shelf components and instead has focussed on designing custom boards that are more versatile and affordable. This thesis continues to work on a star tracker which is designed to run on microcontrollers at high refresh rates giving the ability to run on cubesats. Through investigations, it is found that software will not function without substantial optimisations and modifications. Contained are details on the steps required to perform these tasks.

## 2.0 Literature Review

### 2.1 Universities and Cubesats

In 1961, a 100kg surveillance spacecraft from the US Navy was launched, it was named the SOLRAD-3, but unlike its predecessor SOLRAD-2, this satellite was not alone. Connected to this satellite at launch was the University of Iowa's Injun-I. Having the extra space in the Ferring, the US Navy decided to allow a secondary package to travel with their craft for a reduced cost, what is commonly known as *rideshare* [1]. For years, ridesharing involved cooperating with various payload manufacturers to attempt to get a lift, this was difficult, causing time constraints and excessive oversite, however, 2003 saw the beginning of standardised rideshare platforms. Rideshares were specified to be a  $10 \times 10 \times 10 \text{cm}^3$ , 1.33kg cube. This allowed the use of a standardised deployer for a fixed-size *cubesat* allowing for a group of ride shares to all ride together on a well-documented and standardised platform. Rideshares suddenly became far easier and cheaper to run with satellites now costing between \$50,000 to \$200,000 to launch [2, 3]. Lowering the barrier to entry, universities started designing their own cubesats. While many were tech demonstrations, some actually were designed to provide scientific Earth observations. The QuakeSat from Stanford University designed a cubesat that could measure earthquakes with a magnetometer. The CabX-2 was designed to observe CO<sub>2</sub>, humidity and the atmospheric electron content and SwissCube-1 was used to measure oxygen in the upper atmosphere [3].

## 2.3 Attitude Determination Methods

For the previous satellites to perform accurate measurements, they needed to know where to point. To find a spacecraft's attitude, a variety of sensors can be chosen. Mixing different sensors together can achieve a high degree of accuracy, availability and degrees of control. Observing Table 1 shows the accuracy and degrees of freedom of different sensors [4].

*Table 1*  
*Attitude Sensors*

Type	Initial attitude acquisition	Degrees of freedom	Accuracy
Magnetometer	Yes	3	1 arc minute (0.016°)
Radio Frequency Beckon	Yes	2	1 arc minute (0.016°)
Horizon Sensor	Yes	2	5 arc minutes (0.083°)
Sun Sensor	Yes	2	1 arc minute (0.016°)
Solar Panels	Yes	2	1 degree
Star Tracker	No	3	1 arc second (0.00028°)

*Accuracy of different attitude sensors adapted from [4].*

Magnetometers are 3D compasses that can provide roll, pitch and yaw. As a magnetic field exists on Earth and is consistent, magnetometers are used widely in a variety of applications. BINAR-1, 2, 3 and 4 are all using the MMC5983MA, a \$7 AUD sensor with a maximum accuracy of  $\pm 0.5^\circ$ , however, the team are expecting a lower accuracy as the accuracy test was performed on earth [5].

Horizon sensors are infrared sensors that through observing the angle of the horizon can determine the pitch and roll of the spacecraft. It cannot determine the yaw as every direction on the surface of the atmosphere appears the same. Using this sensor, the satellite can accurately point a payload perpendicular to the earth's surface to an angle of between  $0.393^\circ$  and  $0.075^\circ$  [6].

Sun sensors can measure the azimuth or direction of the sun, providing yaw and pitch but not the spacecraft's roll. These sensors come in many types. A solar panel can be used as a form of attitude determination by having the panels on adjacent faces and measuring the current returned by each panel. By measuring the amount of light entering each face, a rough estimation can be calculated, K. Bolshakov found that accuracy is on average around  $\pm 3^\circ$  [7]. A Photodiode can be used to get a basic attitude with an error of  $\pm 17^\circ$  [7]. However, to get accurate measurements, a grid or array sun sensor must be used. This sensor uses a set of narrow slits that projects the sun's light onto an array of thermocouples, as the spacecraft rotates, the position of the light moves over the sensors. A proposed approach by K. Bolshakov shows a way to get an accuracy of  $\pm 2.5$  arc seconds [7].



Star Trackers are one of the most accurate attitude sensors available for space travel. These sensors use a camera sensor to take photos of stars and identify them. The accuracy comes from the size of the image sensor, the constant nature of the stars and the ability to use each star identified as its own 3D vector. Each 3D vector can be considered as its own high-quality azimuth sensor. This provides the accuracies shown in Table 1 of around one arc second [4].

## 2.4 Star Tracking Methods

In 1976, STELLAR, the first satellite star tracker was designed, this star tracker was able to identify stars by taking a photo and using prior information from another sensor. With the prior knowledge, it was capable of finding a sub-catalogue database which would be approximately where the sensor was looking. By rotating the image, the stars in the image and the database would line up. At this point, the algorithm would pattern match and return a true result. Requiring the process of rotating, the star tracker took a substantial amount of time to the point a match would occur once every few minutes [8].

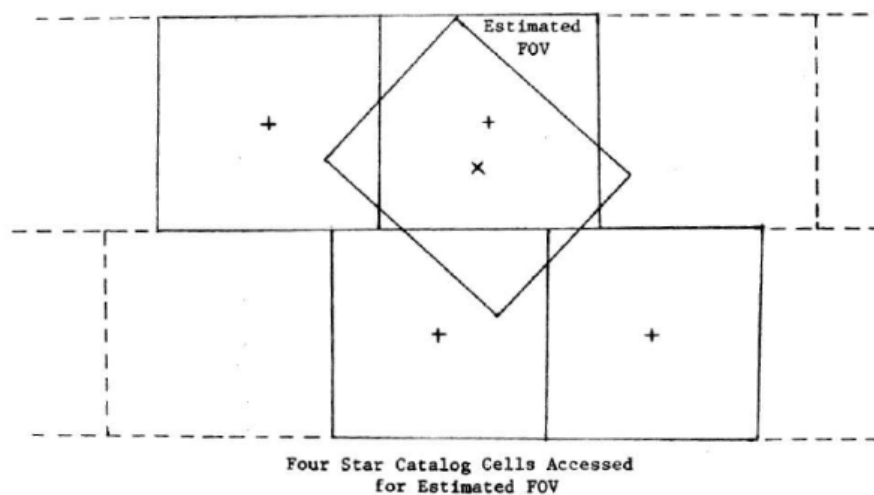


Figure 1: The STELLAR satellite Adapted from [8]

From the success of the STELLAR star tracker, more efficient star tracking modes were investigated. In 1977, Junkins designed a star tracker that used the angles between three stars in the image to pattern match instead of having to rotate the image to plate solve. This algorithm was capable of generating an accuracy of less than 20 arc seconds with a response rate of 15 seconds running on a Motorola 6800 at 2MHz [8, 9]. It was also at this time that the star tracker team noticed blurring the image slightly causes the blob detection to be more reliable and reduces the chance of a pixel being overloaded in a long exposure they called this hyperacuity.

It was found that while Junkins tracking algorithm was far more efficient, the database could still be improved, by sorting the database elements by a unique feature such as the star perimeter, and the lookup time was reduced [8, 10]

In 1995, Liebe developed a star tracker which grabbed three stars and found the arc angle between two of them (1, 2) and the angle between those two arcs (3) this can be seen in figure 2. Leibe also acknowledged how adjusting the field of view modified the sky coverage, the database size and the time to look up an image. He found that he could design a high-precision star tracker with a single-digit degree field of view and get 1-arcsecond resolution at the cost of weighing 5 times larger, consuming 4 times more power, increasing the database size by over double and with that, the response time. He concluded that a wide field of view lenses between 15 and 40 degrees would provide an accuracy of 20 arcseconds, however, they have a far higher performance [4]. By reducing the size of the database and increasing the field of view, Leibe was now the desirable star tracker [8].

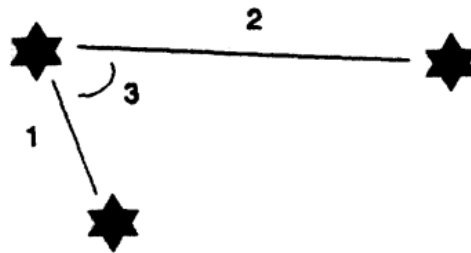


Figure 2: Leibe triangle. Adapted from [4]

In 2004, the pyramid algorithm by D. Mortari was developed. This method like the others would form a triangle with a set of three stars. The arc angle between each side would then be found, however, instead of searching for a triangle in the database, it would find each individual arc angle separately. This was revolutionary as using a database called a *k-vector*, resulted in near instantaneous database search times. Unlike other methods, it also allowed the star tracker to add an extra fourth star for extra verification making it both faster and more accurate [11, 12, 8]. A representation of the pyramid method can be found in Figure 3.

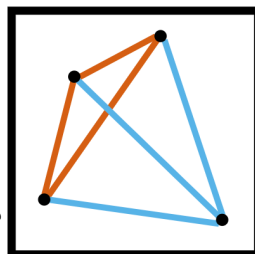


Figure 3: Star Pyramid from the Pyramid Star Tracker.

## 2.5 Voting Methods

Voting is the process of combining a set of observed sensor values with its corresponding actual value. This can be used to combine different sensors together to get a more accurate result or more degrees of freedom [13]. The TRI-axial Attitude Determination *TRIAD* algorithm by H. Black was the first algorithm to attempt to fuse multiple sensors together to find a satellite attitude in 1964. It was used to find a semi-accurate attitude by combining sensor data from a set of sun sensors all pointing perpendicular and parallel to each other. [14]. The proposed method was used to combine two known vectors together. Observing Figure 4, and the corresponding equation 1, using the triad method the sun sensor vector  $\mathbf{s}$  is assigned an axis  $\mathbf{t}_1$ , the next axis  $\mathbf{t}_2$  is given the direction that is perpendicular to the first axis and the magnetic field vector  $\mathbf{m}$ , and then to make a perpendicular 3D coordinate system, the third axis  $\mathbf{t}_3$  is the vector perpendicular to  $\mathbf{t}_1$  and  $\mathbf{t}_2$  [15].

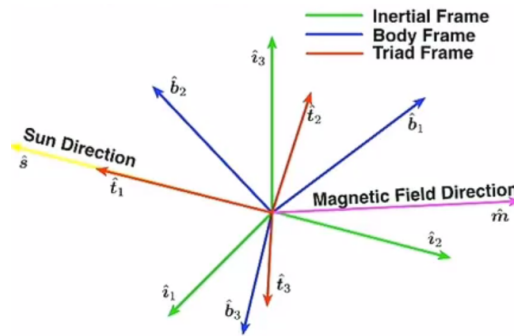


Figure 4: Triad algorithm working with a sun sensor and magnetometer, Adapted from [15].

$$\begin{aligned}
 \hat{t}_1 &= \hat{s} \\
 \hat{t}_2 &= \frac{\hat{s} \times \hat{m}}{|\hat{s} \times \hat{m}|} \\
 \hat{t}_3 &= \hat{t}_1 \times \hat{t}_2
 \end{aligned} \tag{1}$$

While this method was intuitive, it could only work with two vectors and it only uses half the information provided by the magnetometer [15]. Since the conception of this voting method, many permutations have been created, however, Shuster comments on how with new methods, all of the TRIAD voting methods have now become obsolete [16].

In 1965, Grace Wahba proposed a problem stating that to make an optimal attitude determination algorithm, the least squares of every body frame rotated by a rotation matrix should be 0 when the output of the rotation is the world frame. This can be seen in equation 2 where  $\mathbf{W}_i$  is the reference frame,  $\mathbf{V}_i$  is the body frame,  $\mathbf{A}$  is a rotation matrix,  $w$  is the weight of the current vector and  $n$  is the number of vectors. When  $\mathbf{A}\mathbf{V}_i$  is always equal to  $\mathbf{W}_i$ , the system is considered perfectly noise-free and has a perfect set of data [17, 18].

$$L(A) = \frac{1}{2} \sum_{i=1}^n w_i |\hat{W}_i - A\hat{V}_i|^2 \quad (2)$$

In 1968, P. Davenport solved Wahba's problem by using eigenvalues to calculate a quaternion rotation known as the Davenport q-method [18, 19]. Unfortunately, this method required calculating all the eigenvalues and eigenvectors for a high-degree matrix to produce the quaternion rotation and was deemed too computationally expensive to be used.

In 1981, M. D. Shuster modified Davenport's algorithm to instead of calculating eigen values and eigen vectors, to iterate to find the solution. This was called the Quaternion-ESTimator *QUEST*. By iterating over a simpler equation, it took far less time and only took a handful of iterations to get to an accuracy comparable to Davenport's. However, this method does not directly calculate a quaternion, however, instead, it calculates a quaternion from a classical Rodriguez parameter which is a different type of vector [20, 18].

Since then many variants that offer faster methods and ones without singularities such as the ESOQ-2 designed by D. Motari [21]

## 2.6 Background

*Star Tracker Microcontroller* is an open-source *Github* project created by a university student to provide an inexpensive alternative to the off-the-shelf star tracker [22]. This star tracker was designed using the pyramid tracking method and with a cut back design allowing it to be run on an embedded system. The system, however, was set up “as-is” with no optimisations or modifications. Without improvements to the software, it runs too slow and unreliable to be considered a feasible candidate to run on a microcontroller or to be run in space. The background chapter will focus on the design of the essentials of a pyramid star tracker and the technical design work chapter will focus on the improvements to make it flight-ready.

The design of the algorithm can be seen in Figure 5, to simplify the task and make the library more versatile, the project was separated into separate modules. Firstly the image must be captured from the sensor in the module *Image Acquisition*, then the image must be processed to find the locations of the stars through the module *Image Processing*. The star points are then undistorted if the lens has any error and then the stars are projected onto a unit sphere in the module *Projection*. With the 3D coordinates, the algorithm tries to identify unique features of the sample of stars in the module *Tracking Mode*, to match them to a database in the module *Database*. Where finally a module uses the database and image points of each star to find an accurate attitude through the *Voting* module.

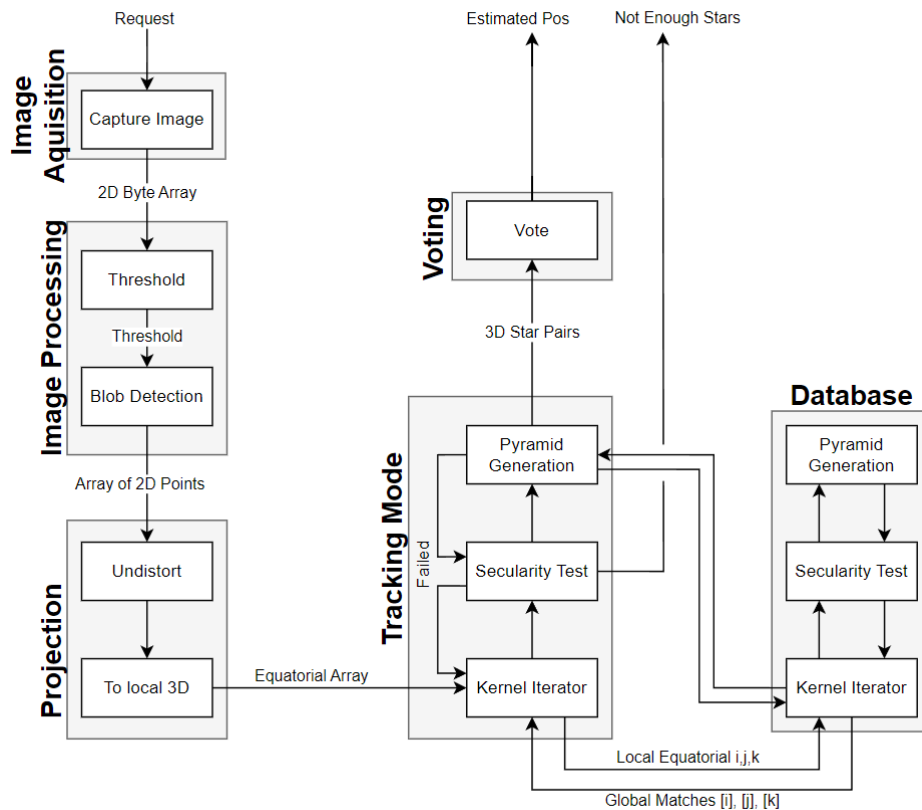


Figure 5: Algorithm Design

### 2.6.1 Image Acquisition

When running a program on a microcontroller, any code related to the peripherals is platform-dependent. At the time of starting this thesis, the hardware was not prepared, and thus the image capture module was not ready. Instead, a demo program designed to run on a PC was used to simulate the image acquisition. Through this images were capable of being read into the library through a Linux file directory.

### 2.6.2 Image Processing

The tracking mode module in the star tracker requires the location of stars in the captured image. To obtain these points, some machine vision manipulations are required. To get the location of stars in the image, the image was first *thresholded* and underwent *blob detection*.

Thresholding is the process of identifying the *foreground* and *background* pixels in an image, in this case, it is the stars and sky respectively. The setup of the threshold used a basic binary threshold as it was expected to perform faster than an adaptive threshold. The chosen method generates a histogram which is the pixel intensity to the number of pixels with that intensity, the generation of this histogram can be seen in Equation 3 and in Figure 6. Using this, it would find the brightest  $n$  percentage of stars using this histogram. To identify the percent-based threshold, it shall be called *Percent Threshold*.

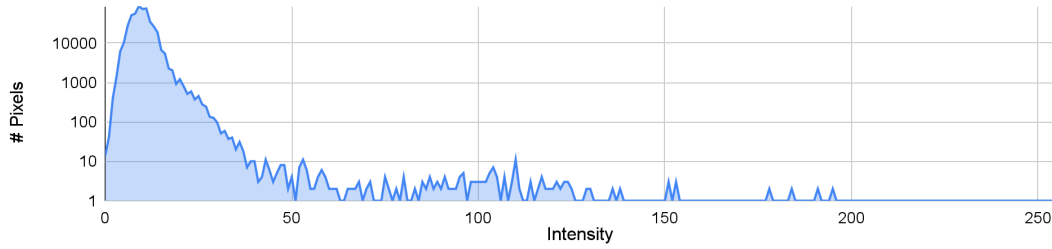
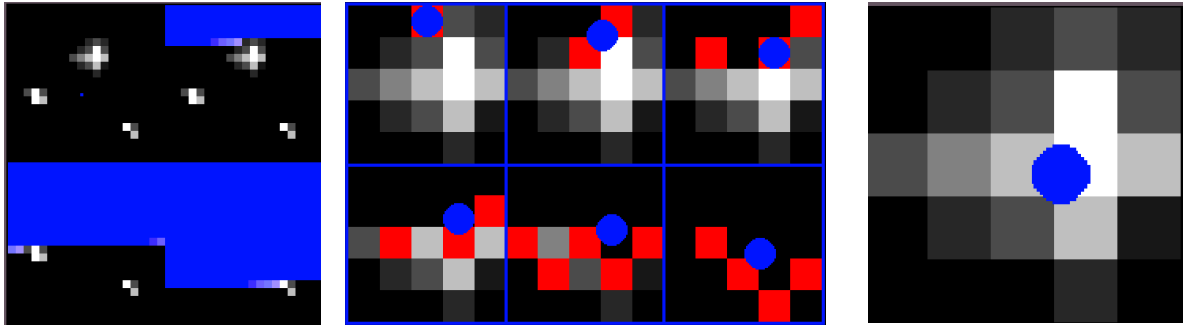


Figure 6: A threshold histogram where each pixel is investigated and added to the intensity it is valued.

$$hist(i) = \sum_{y=0}^{height} \left[ \sum_{x=0}^{width} \begin{cases} 1 & img(x, y) = i \\ 0 & img(x, y) \neq i \end{cases} \right] \quad (3)$$

Blob detection entails finding clusters of “foreground” pixels and combining them into a single object or *blob*. In this case, the foreground pixels in a cluster resemble a star. The method implemented was the grassfire method [23]. The grassfire method wipes across the image row by row until it finds a foreground pixel. The function then fans out to find all the pixels connected that are also foreground. As it fans out it consumes the pixels and calculates the centroid by using Equation 4 where  $I$  is the brightness of the pixel,  $u$  is the  $x$  position and  $v$  is the  $y$  position. An example of the grassfire spreading algorithm can be shown in Figure 7. This method was originally chosen as instead of storing all the prior pixels it examined, it sets them to black. This saves memory and time at the expense of consuming the image.

$$\begin{bmatrix} u \\ v \end{bmatrix}_{centroid} = \sum_{i=0}^n \left\{ \begin{bmatrix} u \\ v \end{bmatrix}_i I_i \right\} \cdot \frac{1}{\sum_{i=0}^n I_i} \quad (4)$$



*Figure 7: Grassfire Blob detection. The image on the left shows the scanner finding a blob, the middle shows how it consumes a blob and the right shows the centroid.*

Using thresholding to identify the pixels that are stars and then blob detection which groups the pixels into a set of clusters, the known position of the stars on a local 2D sensor frame of reference can be identified

### 2.6.3 Projection

After the image processing outputs the sensor locations of the stars, the stars must then be projected onto a unit sphere. This unit sphere represents how the stars are observed from the satellite's frame of reference. This is done in a similar way to how many computer games and rendering engines present an image from a 3D environment. *OpenGL* is a library that implements these methods and has documented it extensively, the equations in this section are sourced from there [24]. The equation to project the image onto a 3D sphere and back is known as the *pinhole intrinsic matrix*, this matrix can be seen as the 3 by 3 matrix in Equation 5. This matrix takes in the focal length of the lens  $f_x$ ,  $f_y$ , the *principal point* or centre-aligned point of the image  $c_x$ ,  $c_y$  and the scale of the sensor  $s$ . These parameters are used to project 2D image points onto a sphere with the reference frame directly in front of the image with  $z = -1$  being forwards.

While the focal length can be obtained by the lens, it will not be accurate depending on the size of the sensor and the distortion of the lens. To calculate the focal length, equation 6 was used, this equation takes in the diagonal field of view of the sensor  $FOV$  and the diagonal number of pixels  $s$  and provides a pseudo focal length to accommodate. The principal point  $c$  should be the centre of the image.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \frac{1}{s} \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad (5)$$

$$f_x = f_y = \frac{\frac{1}{2}\cos(FOV)s}{\sin(FOV)} \quad (6)$$

Using an intrinsic matrix to get the 3D points will provide a set of points in the frame of reference of the camera. If the star tracker is on the *front* of the satellite in the correct orientation, the camera and satellite frames of references are aligned, however, if it is not mounted in this orientation, an *extrinsic matrix* must be used.



An extrinsic matrix is a rotation and translation matrix that transforms between the global and local reference frames, in this case, the satellite's and camera's frames respectively. The matrix to do so is shown as the 3 by 4 matrix in Equation 7, where X,Y,Z are the satellite's frame of reference, x,y,z is the camera's frame of reference,  $\mathbf{r}$  is the rotation component of the extrinsic matrix and  $\mathbf{t}$  is the translation part of the extrinsic matrix.

Calculating the parameters for the extrinsic matrix is done by using OpenGL's *LookAt* function [24]. Distances in space can be considered infinitely large compared to the size of the spacecraft so the translation vector can be considered as 0. As for the rotation matrix, it can be calculated with Equation 8 where  $\mathbf{Q}$  is up and  $\mathbf{P}$  is forwards in the spacecraft's reference frame.

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} r & r & r & t_x \\ r & r & r & t_y \\ r & r & r & t_z \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \quad (7)$$

$$\begin{bmatrix} r & r & r & 0 \\ r & r & r & 0 \\ r & r & r & 0 \end{bmatrix} = \begin{bmatrix} \text{norm}(\mathbf{Q} \times \mathbf{P}) & 0 \\ \text{norm}(\mathbf{P} \times (\mathbf{Q} \times \mathbf{P})) & 0 \\ \text{norm}(\mathbf{P}) & 0 \end{bmatrix} \quad (8)$$

Combining the extrinsic and intrinsic matrices allows the user to convert a 2D point in the sensor's reference frame to a 3D point in the spacecraft's reference frame. This can be seen in equation 9.

$$\underset{1}{s} \underset{2}{\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}} \underset{3}{\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_x & c_y \\ 0 & 0 & 1 \end{bmatrix}}^{-1} \underset{4}{\begin{bmatrix} r & r & r & t_x \\ r & r & r & t_y \\ r & r & r & t_z \end{bmatrix}}^{-1} = \underset{5}{\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}} \quad (9)$$

s: image scale, u,v: pixel position, f: focal length, c: principal point or centre offset,  
r: rotation matrix, t: translation, X,Y,Z: spacecraft position

### 2.6.4 Tracking Mode

Tracking mode is the core part of the algorithm, this section inputs the local 3D star and finds a global 3D star match. The method used by this star tracker library is the *Pyramid Method* developed by D Mortari, A. Samaan and C. Bruccoleri [11]. This method tries to identify a set of stars based on their distance from each other. This distance is known as the *arc* or *great circle angle* which is the angle created when both points are on the plane which cuts through the center of a sphere at the origin. This angle can be calculated by equation 10. Where  $\mathbf{P}$  and  $\mathbf{Q}$  are the 3D local star vectors to calculate the angle of.

$$\theta = \cos^{-1} \left( \frac{|\mathbf{P} \bullet \mathbf{Q}|}{|\mathbf{P}||\mathbf{Q}|} \right) \quad (10)$$

To perform the algorithm, three stars are chosen from the image and formed into a triangle. The distances or arc angles of the triangle are then found. Each angle is then individually searched in the database for similar matches. The database should return a set of star pairs with a similar angle to what is requested. The algorithm will then try to reform the triangle with the given star pairs, this can be seen in Figure 8. A star triangle can only be formed iff each star from the corresponding star pair matches up with one other star from another star pair.

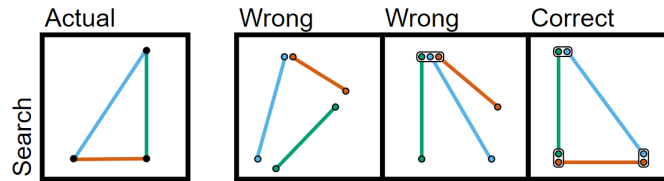


Figure 8: Formation of a Star Triangle from a database search. A star triangle can only be valid if it forms a triangle with the stars.

The star triangle formed will not be unique and further assessments would need to be made to ensure the match is valid, one of these tests is called *specularity* which ensures the triangle is not flipped. If the triangle is a flipped clone of the original, it cannot be a match, figure 9 shows how a triangle of the same size flipped. Specularity can be calculated by Equation 11 where  $\mathbf{Q}$ ,  $\mathbf{R}$  and  $\mathbf{P}$  are the vectors of each star.

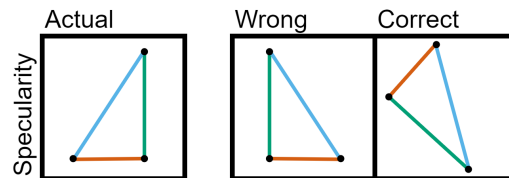


Figure 9: Testing for Specularity. A triangle is not valid if it is flipped.

$$s = \hat{\mathbf{Q}} \times \hat{\mathbf{R}} \bullet \hat{\mathbf{P}} \quad (11)$$

Once the specularity for the star triangle is correct, a fourth star known as the *pilot* star is searched for in the database, as in the first step, the arc angle is calculated between the pilot star and each point on the star triangle. If a match for each arc angle or star pair is found and the star pair matches the desired stars, the star pyramid is considered valid.

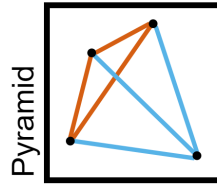


Figure 10: Creating a pyramid.

The pilot star and stars forming the triangle tend to be chosen based on their brightness. The brighter the star, the more pixels are available to find the centroid. However, sometimes a bright star may be unreliable so iterating through a standard triple loop to create the triangle may yield many failures. To overcome this, an alternative method was proposed, the *kernel iterator* by D. Arnas, M Fialho and D. Mortari [25]. This iterator ensures that the first stars in the sequence will not be prioritised in the case of a failure. This iterator can be seen in Figure 11 where in the left algorithm, the value for *ii* will remain the same until every star has been examined.

```

for ii in (0, n]:
    for jj in (ii + 1, n]:
        for kk in (jj + 1, n]:

for dj in (1, (n-2)):
    for dk in (1, n - 1 - dj]:
        for ii in (0, n - dj - dk - 1]:
            i = ii - 1
            j = ii + dj - 1
            k = j + dk - 1

Where i,j,k ∈ (0, ℤ)

```

Figure 11: Algorithm of kernel iterator. Left is the standard approach, right is the kernel-iterator. Adapted from [25]

### 2.6.5 Database

The database for the pyramid method was designed for speed. The performance comes from the implementation of a *k-vector* designed by D. Mortari and B. Spratling [12]. To better understand this system, it is recommended to refer to Figure 12 as an example while reading this section.

The *k-vector* is a linear equation that inputs an arc angle and outputs an index to where angles of similar size are located in the pair list database. The pair list database is a list sorted by the size of arc angles for every possible pair of stars. Contained inside of this list are sets of references to the star catalogue.

The catalogue is a list of stars in equatorial format in an arbitrary order.

As the pair list will not have a linear relationship between the index from the *k-vector* and the arc angle, a lookup table is inserted between the *k-vector* and the pair list. This lookup table has a set of *bins*, these bins convert the linear *k-vector* into the non-linear address of the pair database.

The aim of this database is to input an arc angle with a tolerance slightly above and below the target arc angle. With this, a range of pairs from the pair list are returned. Each pair returned will be an angle within the desired tolerance.

Take for example, an angle of  $6^\circ$  is requested with a tolerance of  $2^\circ$ :

1. The *k-vector* calculates the location in the lookup table representing  $4^\circ$  and  $8^\circ$ .
2. At the provided bins in the lookup table, the ranges 40 to 43 and 43 to 44 are found.
3. Inputting these values as an index into the pairs list will output a set of pairs representing star pairs with an angle between the ranges  $6^\circ - 8^\circ$  and  $8^\circ - 10^\circ$ .
4. The numbers contained in each of these pairs are addresses to the catalogue database.
5. Inputting the pair into the catalogue will return two stars in equatorial form with an angle between  $8^\circ$  and  $10^\circ$ .

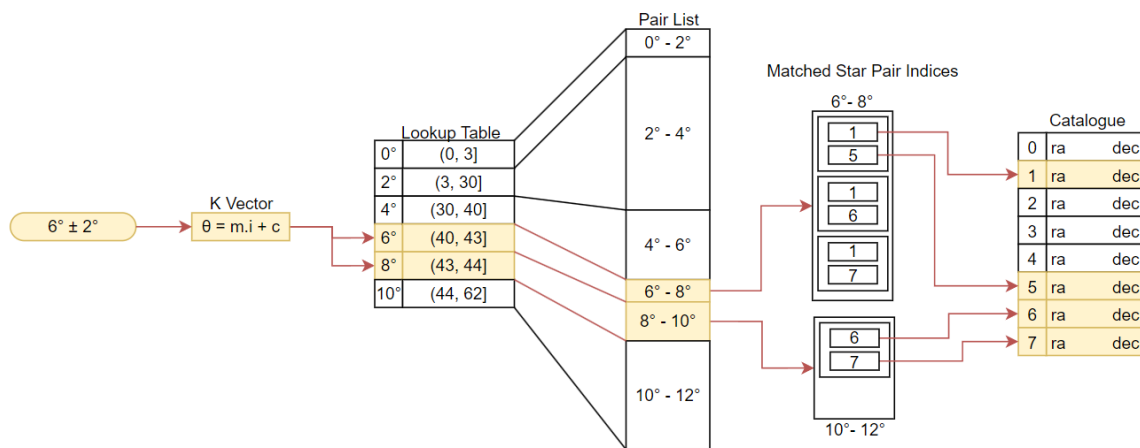


Figure 12: Pyramid database.

The k-vector equation is a simple linear equation where the index of the search bin in the lookup table is the x-axis and the input angle is the y-axis. Equation 12 shows how the k-vector is created, in this equation,  $i$  is the index of the look-up table,  $\theta$  is the input angle,  $n$  is the number of bins or entries in the lookup table and  $e$  is the smallest possible floating point number which is known as *decimal precision*. The decimal precision variable is necessary as if the absolute highest or lowest angle is provided, there could be a rounding error which causes an overflow.

$$\begin{aligned}\theta &= mi + c \\ m &= (\max(\theta) - \min(\theta) + e) \frac{1}{n} \\ c &= \min(\theta) - e\end{aligned}\tag{12}$$

The number of bins required can be calculated by finding the range of star pair angles and dividing it by the minimum angular tolerance. This can be seen in equation 13 where  $\phi$  is the pairs list maximum and minimum angles and  $t$  is the minimum angular tolerance.

$$n = \left\lceil \frac{\max(\phi) - \min(\phi)}{t} \right\rceil\tag{13}$$

Each bin in the lookup table is then set. This is done by looping through each index of the lookup table and finding the target angle by running Equation 12 with  $i$  being the current index. The angle is then found from the pair list and the bin is given the corresponding address.

To search the database, the desired angle and the tolerance must be provided, with this the k-vector equation is run with the target angle plus and minus the minimum tolerance. This will provide a range with all the angles within the tolerance of the angle provided. This can be seen in Equation 14.

The output from Equation 14 is then fed into the lookup table which will provide the max and min bounds for the pairs list.

$$\begin{aligned}i_{max} &= \left\lceil \frac{1}{m} \left( \theta - c + \left( \frac{m}{2} + e + t \right) \right) \right\rceil \\ i_{min} &= \left\lfloor \frac{1}{m} \left( \theta - c - \left( \frac{m}{2} + e + t \right) \right) \right\rfloor\end{aligned}\tag{14}$$

### 2.6.6 Voting

Once the local and global 3D-matched star vectors are found, the orientation of the spacecraft can be determined. This is done using the Quest method [20]. This method was implemented as there was open-source sample code online that could be ported to the library.

### 2.6.7 Software

To run the program on a microcontroller in space, it was essential to have a high-speed, robust language that is commonly used. This limited the project to C, C++ or Rust. After investigating each language, Rust was chosen as it enforces strict typing and memory management. For a critical component that cannot be repaired, it was important to ensure that an error could not occur.

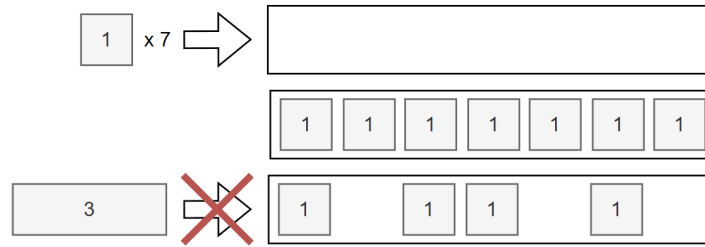
As well as using Rust as the language, it was decided that the *Stack-only Code* practice was to be used. This practice entails completely avoiding the *heap* and *dynamic memory*.

Dynamic memory and heap allocation is the process of declaring variables of unknown size at compile time. For computers with large amounts of memory, this may be beneficial as it provides versatility in storing complex information. The heap also allows users to more easily store data without having to remember where the memory was defined. Stack-based variables are defined at a specific location in code and it can be difficult to access by different parts of the program.

Unfortunately, for devices with limited memory size, the heap is not a viable option for numerous reasons. Firstly, the heap when not managed correctly can lead to memory not being freed when the data is no longer required. Secondly, running a program on a system for an extended period of time can lead to heap fragmentation where free space may not be accessible. Thirdly requesting to store and remove data from the heap is time-consuming and fourthly, having a dynamically sized variable can lead to unpredictability such as using more memory than the system can provide.

In languages such as C, the user is required to manually specify memory to be input onto the heap. The user is also responsible for manually removing memory when it is not needed. If the user forgets to do this or removes the memory address twice, it could lead to data being accidentally deleted or unused, corrupted data slowly reducing the available space on the controller. This can lead to bugs that are hard to reproduce and catch.

When storing data on the heap, the heap will find the first available slot that can fit the memory, during the runtime of the program, some of the used blocks will be freed as they are no longer needed. Over time the heap will look less like a contiguous block and more of a fragmented set of loose memory slots. At this point, trying to add a large variable may not be possible even though there is enough space available. This is because there is no single slot large enough to fit it as small fragments are in the way, figure 13. shows an example of this.



*Figure 13: Heap fragmentation example. A heap of size 7 has seven, 1-sized memory allocations. Three memory allocations are freed, however, a size 3 memory address cannot be stored.*

When trying to store a variable on the heap, the compiler actively has to search the heap for an available slot to fit the memory. This can take unnecessary time to run. Dynamic memory is also considered unsafe as at compile time, there is no way to guarantee that the memory allocated will be smaller than the available space. Using stack memory, this is not a problem as the data is always at a fixed size and the used program memory will always be the same throughout each run.

### **2.6.8 Hardware**

The BINAR team lacked the power constraints and time to set up the satellite controller to be run on a computer. To be compatible with the target satellites, it was decided that the star tracker would be run on its own embedded system.

The design of the hardware of the star tracker was mainly handled externally, this reduced the requirement to design the PCB and decide on the connectors and power stability. However, the choice of the sensor, lens and microcontroller required knowledge from the designer of the software. Prior to this thesis, the microcontroller and sensor had already been chosen.

#### **Microcontroller**

For the cubesats, the BINAR team were using the *STM32H7* series microcontrollers; these embedded systems were chosen for a number of reasons. The STM series has flown in space and has been investigated for power usage, temperature range and radiation shielding by Open Source Satellite [26]. The team found that the controller passed their requirements for power efficiency and ability to survive in space. The controller also had software support with *STM32CubeIDE* which provided easy C and C++ hardware abstraction support [27]. Trusting the research of the group, it was decided to use a microcontroller from the same series the chip chosen for the micro was the *STM32H747* [28]. This chip was chosen as it has 480 MHz clock speed, 2 MB of Flash, 1 MB of RAM and an interface that can automatically read in parallel image sensor data called DCMI. The extra RAM and FLASH allowed for the image to be stored on the chip, the fast clock allowed the controller to run at speeds closer to what was expected from a computer and the parallel interface would reduce the power constraints and response time of the controller. For testing purposes, it was chosen to use the *STM32H755* as it has similar features but comes with a development board to prototype [29].



## **Sensor**

When deciding on the sensor, many considerations were made including the size, processing time requirements, the shutter type and the ability to operate in low light conditions.

Due to the size of the RAM, when excluding the database, the size of the image that could be stored was 1000x1000 where one Byte is one Pixel. To process this image, the program would have to iterate over each individual pixel multiple times. Considering the squared growth of the processing time and the storage requirements of the rest of the program including the database, it was decided to use a sensor with sub HD resolution.

To justify the decision to use the specific microcontroller, the sensor was required to be capable of running with a parallel interface to connect to the microcontroller's DCMI port and support for 8-bit reading to constrain the size of the image. The sensor was required to take photos of stars and thus operate in low light conditions. Sensors with larger pixel sizes tend to be capable of absorbing more light making them more desirable for the project. The final decision was the choice of the shutter, sensors have global shutters and rolling shutters. Rolling shutters capture the image over time whereas global shutters capture the image at the same time. If the satellite were tumbling, a rolling shutter would reduce the accuracy of the sensor. With this in mind, a global shutter was deemed a requirement.

With the requirements specified, it was decided to use the Python480. This sensor is 608 by 808 pixels with large 4.8 $\mu$ m pixels and a global shutter [30]. Using 8-bit pixels, this sensor occupies 491 KB of memory leaving 509 KB for the database and program.

## **External memory**

Without knowing the size of the database, the required memory was unknown. To address the lack of knowledge about database size, using external storage was suggested. This however was rejected by the BINAR group as they were concerned with adding additional points of failure through radiation and vibration damage. This creates the requirement for the database to fit in less than 509 KB of space.

## 3.0 Technical Design Work

### 3.1 Aim

The aim of the thesis was to finish the star tracker project by producing software that can fit and run optimally on a microcontroller. To be a viable option for BINAR and other interested parties, it would need to be more desirable than alternative attitude determination methods.

Currently, BINAR uses a magnetometer and gyroscope to find the satellite's attitude. The magnetometer has an accuracy of  $\pm 1.0^\circ$  [5]. And the gyroscope has an inaccuracy of  $0.49^\circ/\text{s}$  [31]. To produce results that are desirable over this setup, it was decided that the star tracker should have an accuracy of less than one degree. Due to the lack of external forces in space and the uncertainty of the capabilities of the star tracker, the response time was accepted at 10 seconds. To fit on the chosen microcontroller, the program was required to run smoothly on a 480MHz CPU with 1MB of RAM and 2MB of flash.

### 3.2 Lens

The lens was not chosen at the commencement of this thesis as the PCB was not finalised and the part was hot-swappable. The choice of the lens can affect the amount of light reaching the sensor which would require a higher exposure time and thus a higher response time. By reducing the field of view of the lens, the accuracy could be increased as the known point of the stars would be to a higher degree of resolution. Reducing the field of view may also increase the required size of the database as larger fields of view would require fewer stars to be capable of having full coverage of the sky. Inquiring with BINAR also raised the concern that a wide field of view lens would be more likely to have celestial bodies in the image or if the satellite were to have deployable solar panels in the future that may obstruct the lens.

At the time of investigating lenses, BINAR was preparing three cubesats to be launched with each running a star tracker. As opportunities to test in space were limited, it was decided for each star tracker to have lenses with different fields of view. The pixel size of the sensor was an uncommon format of 1/3.6" optical format [30]. Many websites do not accommodate this format when expressing the field of view of their lenses. The field of view of a lens can be calculated by finding the angle between the focal length and the diagonal length of the sensor as shown in Equation 15. The optical format provided by the sensor is not the diagonal length of the sensor but a way of identifying the size of the 2D sensor; to find the actual field of view, a lookup table such as Machine Vision Direct must be used to convert [32].

$$FOV = 2 \tan^{-1} \frac{(\text{sensor diagonal})}{2 \times (\text{focal length})} \quad (15)$$

With the need to test vast amounts of lenses and conform to a budget, affordable lenses were acquired from Arducam [33]. The selected lenses were the 16mm ( $18^\circ$ ), the 8mm ( $34^\circ$ ) and the 3.6 ( $68^\circ$ ) lenses. These lenses were not ideal as the aperture was narrow at f/2 and the target sensor size was 1/2.5 inch meaning that less light could get through and some of the light admitted would not land on the sensor. The lenses also had an IR filter on them, so it was decided to order two 8mm lenses and remove the IR filter from one to see the effects. An extra set of lenses was also chosen from a factory that designs security camera lenses the 6mm ( $44^\circ$ ) and the 4mm ( $64^\circ$ ), these lenses have an f/1 aperture and are labelled as *starlight* as they could admit far more light than a standard f/2 [34].

Tests were performed in Gingin, Perth, Western Australia which is labelled as *Bortle* class 3 [35] performed on moonless, cloudless nights. The images were captured through an industrial machine vision camera with the Python480 sensor from Allied Vision. The output images were to be run through the software *nova.astrometry.net* to get the actual field of view, the distortion from the lens and the number of stars that could be seen. By taking photos of the same region of the sky, the lenses could be compared with each other.

The first observation made was that all lenses achieved full sky coverage. For a specific field of view, there will be a certain magnitude of brightness required so that there are no possible orientations that would have less than four stars. Observing Appendix A, 18 degrees require at least 5.44 and the rest have higher margins. Appendix B also provides information about the percentage of sky covered with four stars at a given magnitude and field of view.

It was also found that the IR filter does not provide any substantial advantage or disadvantage to seeing stars. The starlight lenses were also tested and the gain in magnitude was not significant compared to the amount of error they create.

It was also observed that the 16 mm lens is  $\frac{1}{3}$  the error but only half the FOV of the 8mm. Because the accuracy and ability to block out external bodies goes up as the field of view goes down, it was decided to run the rest of the tests with the 16mm lens.

*Table 2: Observations with varied lenses observing the southern cross with max gain and an exposure time of 0.2 seconds.*

Focal Length	16	8	8	6 - starlight	4 - starlight	3.6
FOV	17.8°	34.7°	34.7°	45.2°	64.0°	69.5°
Aperture F/x	2	2	2	1	1	2
IR Filter	Y	Y	N	N	N	Y
Found Stars	28	31	32	38	41	35
Magnitude	5.59	4.86	4.86	5.43	5.58	4.71
Average Error (°)	0.012	0.038	0.041	0.12	0.116	0.2
STD Error (°)	0.014	0.04	0.027	0.112	0.04	0.616
Max Error (°)	0.074	0.227	0.046	0.474	0.089	0.17

### 3.3 Tests

This thesis required the implementation of the pyramid algorithm; however, it did not provide any additional functionality or optimisations. To achieve the performance in the requirements, a set of tests was designed to compare the improvements of each optimisation. In each set of tests, the computer, hardware or input parameters may have been changed to better respect the purpose of the test and provide more accurate comparisons. Because of this, each optimisation should not be compared with other optimisations.

#### 3.3.1 Simulation Test

To examine the reliability and average response time of the pyramid algorithm and database without having the camera and lens projection prepared, a simulation was designed that bypassed the initial image acquisition. This simulation generated a set of points on a unit sphere resembling the centre of each test image. For each test, the 3D local position of the visible stars was provided as if the star recognition was successful. The pyramid algorithm and database could then run in a vacuum without any errors in the other sections.

Input variables could then be modified to vary the tolerance of the database searches, the field of view, distortion to the input star positions, absolute magnitude range and the accepted timeout. Outputs include the average time, the number of positives, negatives, false positives and false negatives.

#### 3.3.2 Demo Test

Using simulation testing provides numerical data showing the reliability of the pyramid algorithm, however, it does not consider unexpected errors or the acquisition and processing of the images. The demo test reads actual images taken from a camera and attempts to run the entire algorithm like it would be run in a real scenario. The images were taken through a camera with the same sensor proposed to be used in the star tracker and ran through the plate solver *nova.astrometry.net* to identify the exact pointing position of the camera [36].

This test had many input parameters such as the tolerance, absolute magnitude range and of the database and field of view. These values were tweaked to get the most desirable result for the most images. The output results were the reliability and accuracy of the algorithm. The desired result is disclosed in each test.

### ***3.3.3 Integration Test***

The integration test involves running the Demo Test on the target hardware. The target hardware includes the NEUCLEO-H755ZI-Q which uses the STM32H755 chip [29] and the Raspberry Pi 3B. The Raspberry Pi was considered a potential target piece of hardware as it is a low-power computer that could be used as the main controller for a CubeSat. Due to the ease of use of having an operating system, the Raspberry Pi was tested as a candidate.

The Integration test was expected to produce a similar result to the Demo test so the purpose of the integration test was to ensure that the constraints for the star tracker were met on a potential candidate.

### ***3.3.4 Absolute Magnitude***

The absolute magnitude range was automatically chosen for most tests based on the least number of stars required to have full sky coverage. Calculating this allowed the database to remain at the lowest size and the exposure of the camera to be reduced. Finding the required magnitude was done by using the Monte Carlo simulation to observe random points around the entire sky. If any of these points had less than 4 points within the target field of view, the magnitude was not high enough. The results of this can be found in Appendix A and Appendix B.

### ***3.3.5 Initial Performance***

On starting the thesis, the algorithm ran but was not practical. Testing with a realistic set of parameters on a PC returned 23 failures and 1 false positive out of 24 simulation tests with an average response time of 4 minutes. The tests were performed using an AMD Ryzen 5 3600 @ 3.6GHz [37] with a field of view of  $18^\circ$ , error tolerance of  $0.03^\circ$  and error of  $0.01^\circ$ . These tests did not consider the exposure time or time to process the image.

To ensure the algorithm worked, the error was reduced to  $0^\circ$ . This had a 100% success rate taking 4 minutes per test. Testing with the error tolerance also reduced to zero yielded 100% success rate with a response time of 2.75 milliseconds.

For these tests, the database was calculated at 750 KB which while within the memory size of the microcontroller, limits the available space for other variables and forces the database to be kept in Flash instead of used in RAM.

The results from this show that the algorithm works, however, it fails to meet the constraints as it both exceeds the response time and the memory requirements.

### 3.4 Image Processing

Image processing entails thresholding the image and then performing blob detection on the stars. Using the 608x808 *Python480* camera, the thresholding took 870 ms, and the blob detection took 847 ms with a total of 1717 ms. This section was designed hastily, giving many opportunities to optimise and improve. When investigating this section, it should be noted that it is impossible to test for realistic images with the current sensor as atmospheric distortion and light pollution affects the accuracy and noise of the sensor. The images used for testing were performed in a rural area on a moonless night with low wind to avoid unnecessary disturbances.

#### 3.4.1 Thresholding

In the original algorithm, the *percent threshold* was used. Shown in Figure 14 is a sample histogram made from an image of stars. With the Percent threshold, it can differentiate the stars from the background, however, the Percent threshold does not consider any potential interference such as the glow from the atmosphere or blooming from the sun.

The Otsu's threshold was investigated for a solution to this problem [38]. The purpose of this method is to find the “well” between the foreground and background and as seen in Figure 14, the algorithm can identify the difference between a bright foreground object and a dull background object. However, this method does not consider dull stars or the glow of stars. It also does not consider how to handle if part of the image is illuminated by a celestial body.

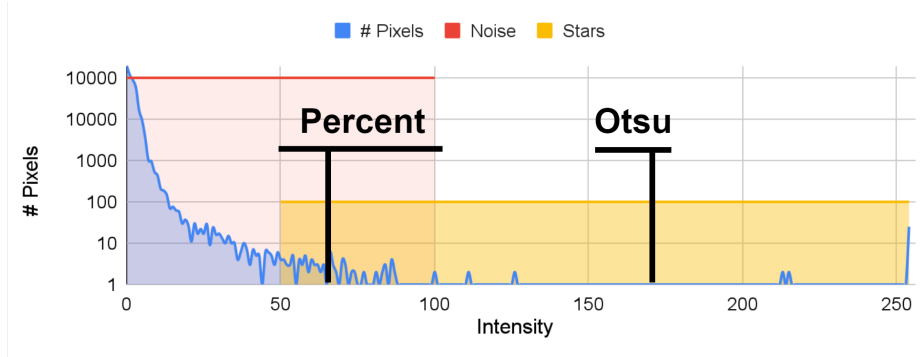
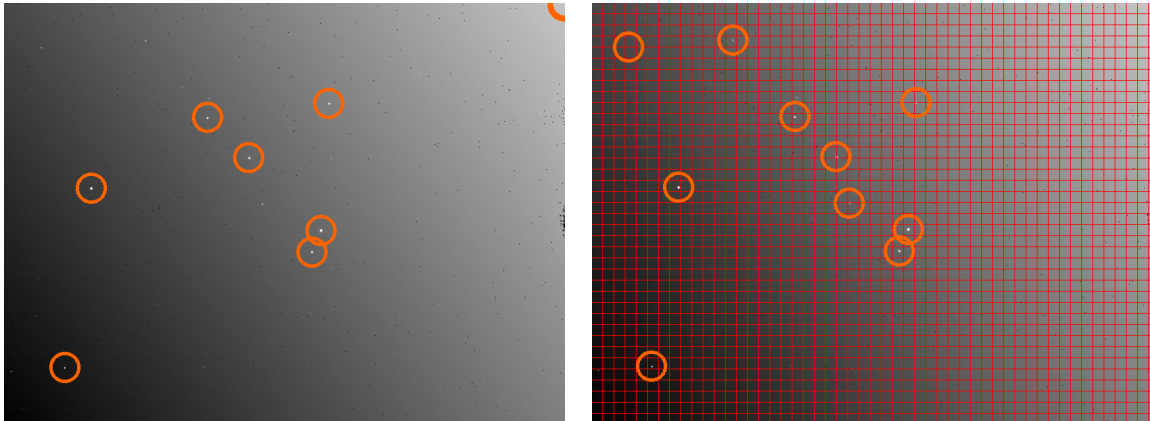


Figure 14: Number of Pixels vs Intensity of star image, The area in red contains noise and the area in yellow contains stars.

Because of this, alternative thresholding methods were considered and the Niblack method was implemented [39]. A comparison between Percent and Niblack methods can be shown in Figure 15. In the image, a bright object such as the sun is illuminating the lens, the Percent threshold fails as it misidentifies the glowing part of the image where the Niblack threshold is more reliable and capable of detecting duller stars.

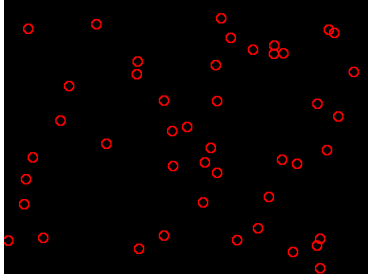
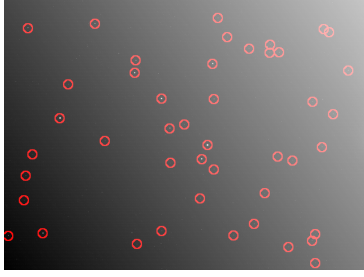


*Figure 15: Artificial example of Percent Threshold (left) vs Niblack Threshold (right). Percent threshold has 4 misidentifications in the top right corner, the red lines in Niblack threshold represent the boundaries to cells.*

A time comparison can be seen in 3 with skip set to 0, which tests both methods on a microcontroller. While the Niblack method takes slightly longer, it produces more positive matches and fewer false positives. For the Percent threshold, the input percentage parameter had to be modified to correctly identify each image. With a more refined approach, the Niblack was able to correctly identify the stars without changing the input parameters.

Using this information, the loss of time was insignificant to the improvements in the identification of stars. By having misidentified stars, time errors could accumulate further in the program.

Table 3: Comparing Percent and Grid Threshold of a photo centred on the star Kappa Scorpii. Tests were performed on an STM32H755 microcontroller.

	Kappa Scorpii normal			Kappa Scorpii Bloom			
							
Method	Percent @ 0.9998%	Niblack @ Cells: 50x50 Add: 50 Skip: 0	Niblack @ Cells: 50x50 Add: 50 Skip: 3	Percent @ 0.9998%	Percent @ 0.9999%	Niblack @ Cells: 50x50 add: 50 skip: 0	Niblack @ Cells: 50x50 Add: 50 Skip: 3
Time	859 ms	993 ms	92 ms	859 ms	859 ms	997 ms	100 ms
Positives	12	12	12	8	6	10	10
False Positives	0	0	0	8	0	0	0

### 3.4.2 Blob Detection

The chosen blob detection algorithm was the grass fire method [23]. Although this method consumed the image, there were no requirements stating that the image must be viewed after running. The method was found to be fast and memory efficient so alternatives were not investigated.

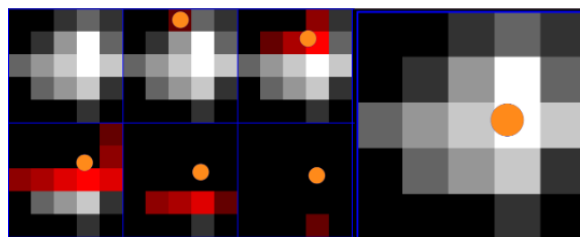
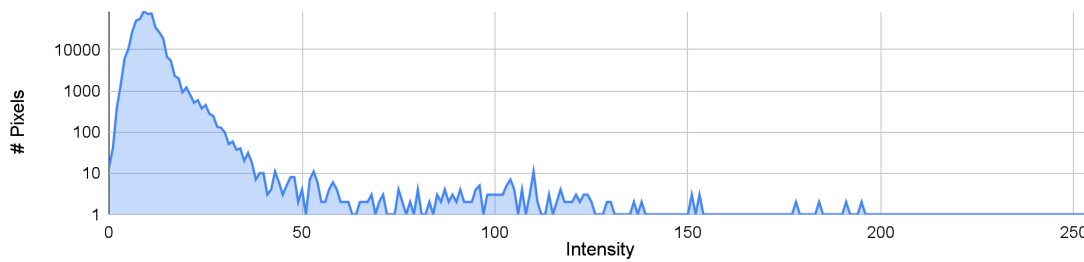


Figure 16: Grass fire blob detection consuming foreground pixels. The orange circle is the calculated centroid and the red pixels are the pixels being observed.



A problem observed with the blob detection was that bright pixels appeared in the image when there were no stars. This observation seen in 17 is known as hot pixels. In this diagram, all the pixels should be in a cluster near 0 intensity, however, some pixels are as bright as 200. As the pixels are significantly brighter than the surroundings, they can be misidentified as stars. A paper [40] observes the effects of a high radiation environment in space damaging CMOS image sensors causing them to form hot pixels. They found that once damaged, they cannot be corrected and tend to be unusable. They also found that hot pixels tend to form in areas of weak pixels, this could imply that hot pixels tend to appear near each other. While this was not observed with the sensor, it has not been exposed to radiation expected from space.

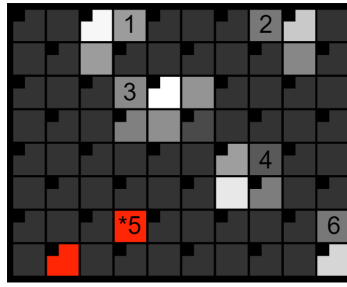


*Figure 17: Number of pixels vs Intensity of dark frame. This image was taken in full darkness.*

Due to time constraints, the solution to stopping hot pixels from affecting the image was to only use blobs containing at least 2 pixels. Many of the brightest stars in any image would not be removed, however, singular pixels would be removed. This also helps with accuracy as having a star take up multiple pixels allows the identification of the star to have sub-pixel accuracy. For singular hot pixels, this method works, however, for clusters of hot pixels, this method would fail. A way to more accurately reduce hot pixels could be to store a list of hot pixels and replace them with the average brightness of the neighbouring cells. The stars could be identified by taking a photo of the Earth's ocean at night time or when identifying stars in an image, to identify bright pixels which are not stars. This method would be more robust, however, it would require memory to run. Due to time constraints, these approaches were not implemented or tested.

### 3.4.3 Skipping

As the blob detection was to ignore single-pixel blobs when wiping the image to find foreground pixels, the algorithm could skip every second pixel without affecting its ability to identify blobs. This can be seen in Figure 18 where any blob bigger than one is identified and some single pixels are not investigated. This method was also found to work with Niblack thresholding, the performance effects of skipping can be seen in Table 3 in the column where skips are 3.



*Figure 18: Example image of skipping. Every pixel with a black box in the top left corner is skipped. The pixel with a number represents what pixel the algorithm begins to identify the blob. The red pixels are hot pixels.*

Using skipping, thresholding is reduced from 993 ms to 100 ms and blob detection is reduced from 847 ms to 647 ms. By applying all the modifications, the total time has been brought down from 1717 ms to 747 ms on a microcontroller.

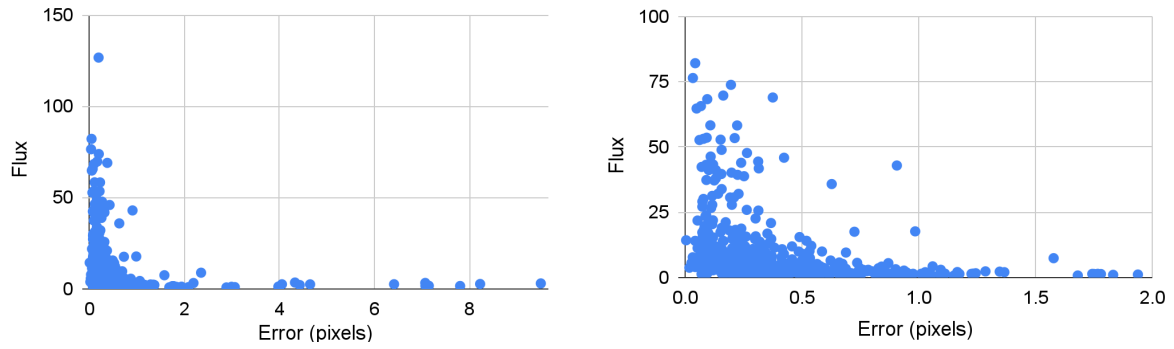
### **3.5 Projection**

On starting the thesis, lens projection was already completed and did not need modification, however, an investigation into the accuracy of a pinhole camera model on a standard lens needed to be analysed. One area of concern was the distortion of the lens,

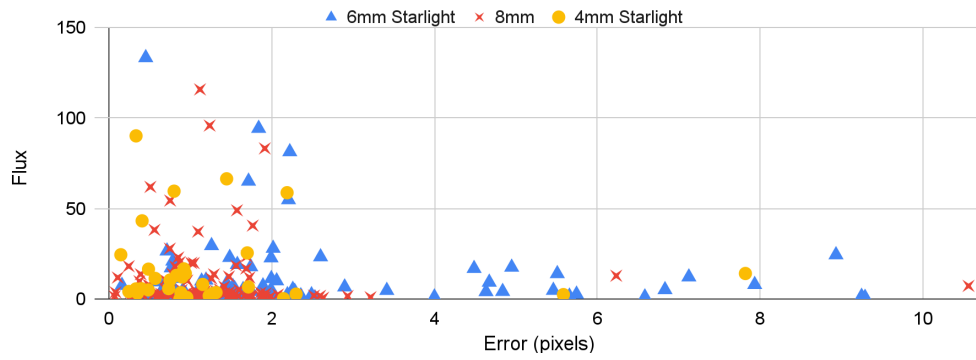
#### **3.5.1 Lens Distortion**

If a lens is too distorted, the arc angles calculated for the pyramid method will be wrong, causing misidentifications. Through simulation tests, it was found that the time to identify constellations and the chances of a false positive were impacted by the increase in variation.

Attempts were made to run sample images from the Allied Vision camera through the demo test. Doing this was largely successful for the 16mm 18° lens, however, it struggled with the larger field-of-view lenses. This was unexpected as the simulation tests showed they were capable of running every lens with their given error parameters. After investigating the error further, it was found that the algorithm was highly dependent on brighter stars and tended to ignore the duller ones. Because of this, it was important that the brightest few stars needed to be the most accurate. Figure 19 shows the error compared to the brightness of stars, observing these graphs shows that the brightest stars are the most accurate. Looking at figure 20, it can be seen that the other lenses were far less accurate with their brightest stars which caused the star tracker to be inaccurate.



*Figure 19: Brightness of star compared to error in pixels of a 16mm lens. The right is a zoomed-in version excluding the outliers of the left.*



*Figure 20: Brightness of star compared to error in pixels for a range of lenses.*

To improve the accuracy of the lens correction algorithms were implemented. The two approaches were to use the error of stars from nova.astrometry.net [36] to generate a map to calibrate the lens, the other attempt was to use a checkerboard. The astrometry.net approach would be more desirable as it would allow in-orbit calibration to account for damage caused during launch. To test whether the lens was corrected, the output from the distortion algorithm was to be run through astrometry.net, if the error was less than the undistorted image, it would be considered corrected. Undistortion was attempted using both OpenCV and Matlab as they both had the tools required to perform an undistortion [41, 42]. Using the star error attempt was run, however, this produced worse results than the undistorted image for every lens. The checkerboard approach was then attempted, this approach was difficult as the lens had to be focused at infinity to take clear images of the sky, however, the checkerboard was not at infinity. To make this work, the checkerboard was held 10 meters away from the camera, the checkerboard can be shown in Figure 21. As the checkerboard was still not in focus, OpenCV was unable to identify the points on the checkerboard, however, Matlab succeeded. Undistorting the image and taking photos of the stars, the error was still larger than without undistortion.

After analysing a set of photos taken at different times pointed directly up at the zenith, it was noticed that the error varied greatly, having the undistortion method also fail to improve the images implies that the error is not due to the lens and is likely due to atmospheric distortion. It is likely running tests on Earth would never produce clean lens distortion values. While there may be more ideal ways to test for undistortion, there was not enough time in the duration of the thesis so the 16mm lens was used in tests as it did not require undistortion.



*Figure 21: Slightly out of focus checkerboard.*

### 3.6 Pyramid Method

The pyramid method is the main part of the algorithm which inputs 3D point vectors as stars, identifies them in a database and outputs the match.

#### 3.6.1 Identifying Bottlenecks

The algorithm was written to implement the pyramid tracking algorithm without alterations or optimisations. Knowing that the algorithm was correct, heuristic approaches were required to optimise the system.

The approach to optimising the system was to time each individual part of the algorithm and identify the sections that took the longest. An investigation of features of the identified section would then be analysed to find the likely cause. Knowing the desired field of view and memory requirements, the tests were mainly focused on using a 18° degree field of view whilst ensuring the database was smaller than 500 KB.

#### 3.6.2 Star Triangle Iterator

When observing the code of the pyramid algorithm, it was found that searching the database took the longest. This was attributed to the number of star pairs found per database search which tended to yield over 100 results. With the known culprit, the database structure was analysed. Observing the number of star pair angles within the accepted angle tolerance range showed that a single search could yield hundreds to thousands of star angles using the 18° lens. The blue *without region reduction* series in Figure 22 shows the bins of the k-vector and how many star pairs are within a single field of view. The design of the database requires that a bin from above and below the desired angle must be chosen. Because of this, a point on the observed line in the figure is half what needed to be searched.

Examining the database search code found that the algorithm would attempt to find all solutions to a set of 3 stars before performing the specularity and pyramid tests. While this was easier to implement, it resulted in many combinations of star triangles being formed when the first had a chance of being correct. To fix this an iterator was created which would iterate over every possible triangle orientation when requested rather than all at once. After implementing this, the failure rate did not improve, however, the time to fail was reduced drastically, this can be shown in Table 4 between None and Triangle Iterator.

Table 4: Test results with and without the triangle iterator at 18 degrees field of view and 0.03-degree angle tolerance.

Optimisation	FOV	Error	Error Tolerance	# Tests	# Failures	Average Time (ms)
None	18	0.01	0.03	24	24	240,000
Triangle Iterator	18	0.01	0.03	24	24	20,000
Region Reduction	18	0.01	0.03	24	22	71
FOV Reduction	18	0.01	0.03	24	24	80

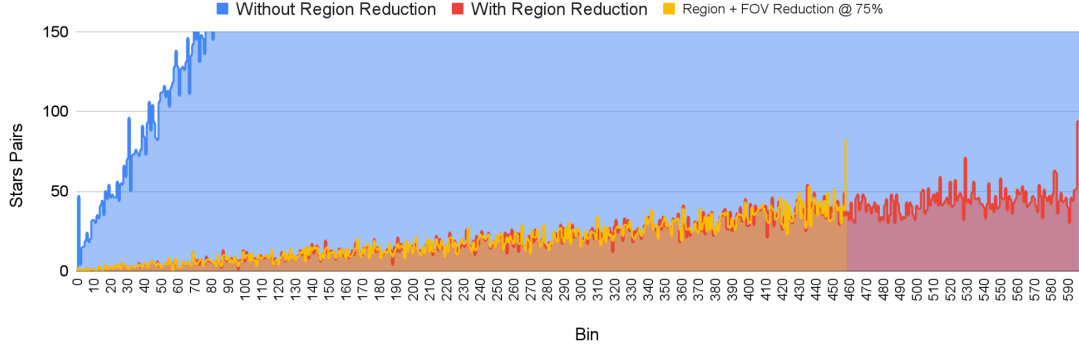


Figure 22: Star Pairs per K-Vector Bin for an 18 degree field of view lens.

### 3.6.3 Region Reduction

In Figure 22 *without region reduction*, the number of star pairs returned for a single search is large. In the case that the first three chosen stars were wrong, the algorithm would have to run through thousands of false matches before moving to the next set. To improve the speed of the algorithm, it was required to reduce the number of star pairs.

The star pairs are generated with any two stars in the star catalogue within a field of view angle of each other. As the number of stars increases, the number of star pairs exponentially increases, by reducing the number of stars in the catalogue, the number of star pairs would be reduced.

Figure 23 shows the projection map of the sky, to highlight the hotspots of stars, a low opacity circle around each of the stars is drawn. In the area of the Milky Way, the concentration of stars is significantly larger than the poles. A decision was made to reduce the number of stars in the catalogue based on the number of stars in its neighbourhood. This was done by generating a set of points equally distributed over a sphere and ensuring only the  $n$  brightest stars are within that range. The points should be spaced significantly closer than the field of view of the camera so that if the brightest stars are all in the same area of their neighbourhood, there would be no gaps. To distribute  $n$  points on the sphere, the Fibonacci lattice was used. This is a way of getting near evenly spaced points on a sphere with little computation power [43].

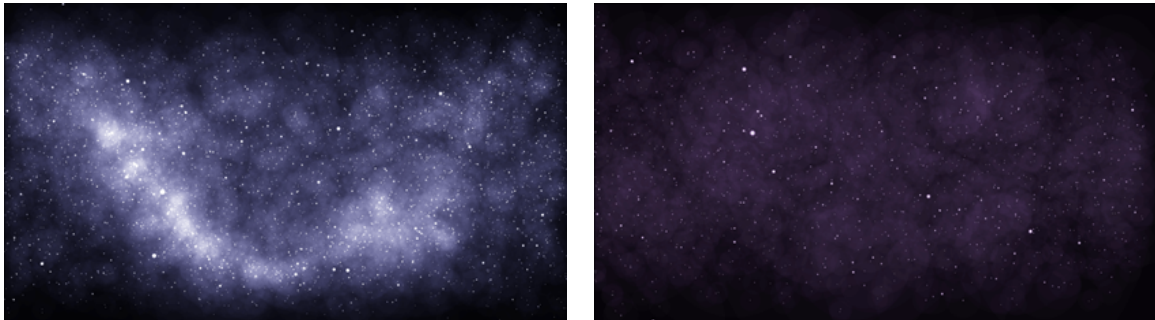


Figure 23: Visualisation of star database without region reduction (left) and without region reduction (right).

Using *region reduction*, it was found that a region slightly larger than half the field of view and 8 stars in each region yielded the best result for every field of view while not impeding the algorithm's reliability or accuracy. Observing figure 22 *with region reduction*, it is clear the number of search results has significantly decreased. Table 4 shows the increased performance where the reliability and speed has drastically improved.

Implementing region reduction also drastically reduces the size of the database which can be seen in Figure 24 where at low fields of view, the database size exponentially increases.

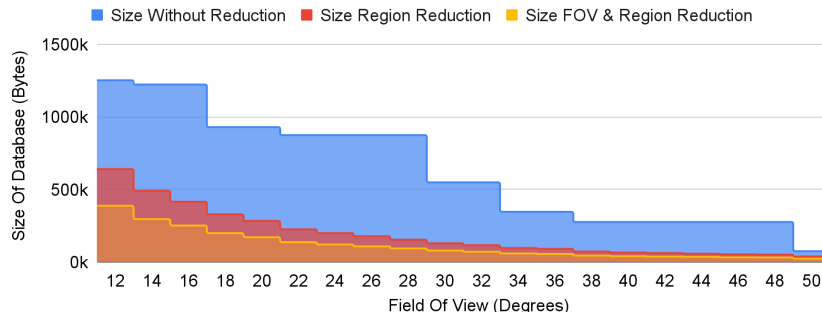


Figure 24: Database Size to FOV showing with and without region and field of view reduction.

### 3.6.4 Field of View Reduced

Observations shown by passing test images through nova.astrometry.net [36] found that stars on the outside of the images were duller and thus were less likely to be identified. Referring to Figure 22, it can be seen that there are more combinations of stars at higher fields of view. Having an angle greater than what is necessary significantly increases the number of star pairs which increases the search time and database size. By reducing the allowed arc angle between the stars in the database, the size is dropped, however, the number of potential matches also diminishes, reducing the reliability. The tradeoff can be seen in figure 25. It was found that having the star pairs only able to form if their arc angle is below 75% of the field of view does not significantly impact the performance success rate or response time.

Implementing this, it was expected that the response time would go down, however, it slowly increases as the reduction increases. Observing Figure 22, it is clear why, instead of lowering the number of bins per search, the bins on the end were removed, for any star pair that has not been culled, the response time should remain unchanged, however, if a selected star pair was one that was culled, the algorithm would fail on the current search and have to search again taking up time.

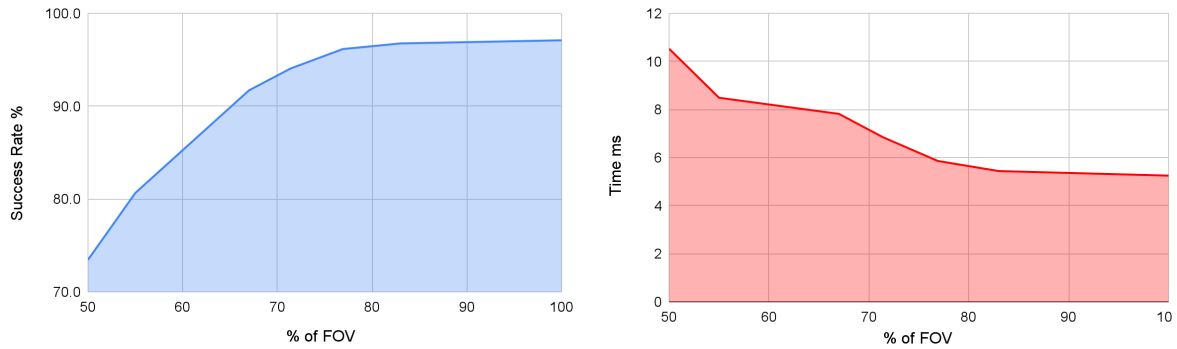


Figure 25: Reduction of Field of View vs the success rate and completion time of the algorithm.

### 3.6.5 Chunk Iterator

Reducing the number of stars was approaching diminishing returns, however, the bottleneck was still the excessive number of search results. Because of this, other ways of reducing the search time were investigated.

In software engineering, a common way of comparing algorithm speeds is the Big-O, which specifies how long an algorithm will take as the number of elements approaches infinity. It is used to compare an algorithm which may be  $O(n)$  to one which is  $O(n^2)$ . An algorithm of  $O(n)$  is expected to loop as many times as there are elements in it while an algorithm of  $O(n^2)$  will have a double loop and as more elements are added, the number of iterations exponentially increases.

For the algorithm to compare each star pair with each other to form a triangle it takes  $O(n^3)$  time, this result occurs as a triple loop is used where each loop represents one of the three stars. To reduce the time of combining star pairs, the number of search results must be reduced.

Instead of finding ways to reduce the number of star pairs, the focus changed to how to categorise the star pairs more appropriately. Currently, the star pairs returned when they were within a tolerance of the target orientation, however, this did not consider other factors. After investigating, it was found that a vast number of matches were in completely different sections of the sky. By categorising the stars by location in the sky, the number of searched elements is reduced which reduces the instability of the  $O(n^3)$  triple loop.

It was found that by iterating over *chunks* of the sky and only taking stars that fit within the target chunk, the order of complexity became  $O(n^3) + O(m)$  where  $n$  is the number of star matches and  $m$  is the number of regions to iterate over. Adding more chunks  $m$  does add time to the search process, however, as  $m$  increases,  $n$  decreases which makes the entire process faster. Implementations of chunk iterator proposals can be seen in 26.



To implement this new approach many methods were theorised. Firstly it was decided to divide the regions of the sky up by points on the Fibonacci lattice as done in *region reduction* this method was called *regional chunk iterator*. To identify the region each star pair was in, extra information was stored in the database, this was an integer specifying what point the pair belonged to. Considering that a pair could span over multiple regions, it was decided that the integer was a bitfield so it could be specified in multiple regions.

This method was successful and greatly reduced the search time of the algorithm, however, storing the bitfield in the star pair substantially increased the size of the database. It was found that storing the bitfield on the star did not reduce the search time of the algorithm but reduced the size of the database as there are more star pairs than stars in the database.

While this method worked successfully, it did have the downsides of adding complexity to the database, increasing the size of the database and having arbitrary points, it was difficult to debug. It also had the problem where the bitfield was 32 bits meaning that there could only be 32 regions, and lacked versatility as the smaller the field of view, the more regions were required.

In the search for a method that did not modify the database, it was decided to use the Fibonacci lattice but instead of calculating the stars region at compile time, to do it at runtime, this method was called the *crunch chunk iterator*. It was found that this method was substantially slower than not having an iterator, thus this method was disregarded.

From this information, it was decided to find other unique features of the database. While calculating the distance between stars and a centre point was too time-consuming, a heuristic approach was considered. The database stored stars in equatorial form with latitudinal and longitudinal angles, this was done as storing a 3D vector would take up unnecessary space and not add extra resolution. By segregating the stars in ranges of declination (north, south ranges), a pseudo region could be created this was called the *declination chunk iterator*. With this, only stars in a small vertical band would be considered. When tested this method worked with similar success to the *regional chunk iterator*, however, without the extra database size. The method also had an added bonus that iterating each chunk could be randomised. This was useful as to ensure full sky coverage, the chunks needed to overlap slightly. By skipping every second chunk, after half the chunks have been investigated, there would be no overlaps.

While the *declination chunk iterator* was successful, a less novel approach was investigated where instead of just considering the declination, the right ascension (east, west) was also considered this was called the *equatorial chunk iterator*. This iterator was more complex to design and resulted in far more regions, however, it did not offer a substantial improvement over the *declination chunk iterator* and due to its complexity, the randomised iterations were not possible.

Table 5: Performance comparison of different chunk iterators at 18 degrees, all optimisations on with a PC. Chunk was attempted, however, the tests were aborted with 0 successes after an hour.

Optimisation	Tests	Successes	False Negatives	False Positives	Average Time Success (ms)
None	1484	1382	101	1	60.14
Regional	1484	1477	6	1	10.44
Chunk	1484	-	-	-	-
Equatorial	1484	1480	4	0	6.13
Declination	1484	1479	4	1	7.93
Declination Random Parity	1484	1480	4	0	5.90

Observing table 5, the difference between not having a chunk iterator and having a chunk iterator is significant. Considering random error, false negatives and the run time are reduced greatly. In this test, declination was the fastest when using random skipping, however, both Equatorial and Declination appear to have the same error. From the results, *declination random parity*, will be used for future testing.

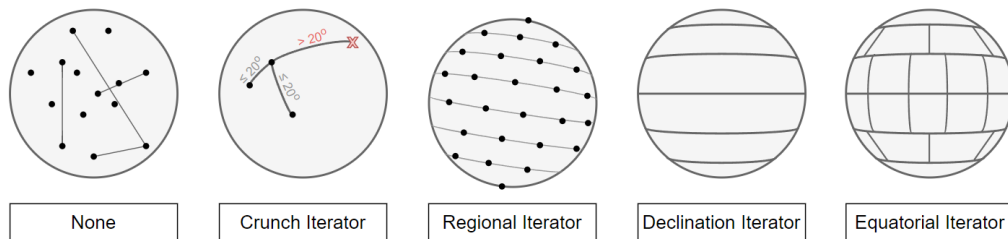


Figure 26: A visual representation of each type of how chunks are defined in each iterating method.

### 3.6.6 Chunk Area Search

Designing the architecture for the chunk iterators allowed the design of an alternative approach to the star tracker. By specifying the expected right ascension and declination bounds, such as in the *equatorial chunk iterator*, the search window and search time would be reduced. Knowing a rough estimate of the satellite position can be done by using the star tracker normally and using a gyroscope to measure the roll of the spacecraft. It can also be done by the use of extra, lower-accuracy sensors. Having a magnetometer will provide a rough approximation, however, that rough approximation would produce a highly accurate input for the star tracker to run. Table 6 shows that this chunk search is substantially faster and with similar accuracy to the other iterator methods.

Table 6: Performance of chunk search error when using an inaccurate sensor to provide a rough approximation of attitude.

Secondary Sensor Error	Tests	Successes	False Negatives	False Positives	Average Time Success (ms)
0°	1484	1475	9	0	0.34
1°	1484	1478	6	0	0.35
2°	1484	1479	5	0	0.36
5°	1484	1478	6	0	0.29
10°	1484	1479	4	1	0.32
20°	1484	1481	3	0	0.49

### **3.7 Platforms**

Writing code for different device architecture can require changes to a software's code and design philosophy. By changing the architecture; library availability, peripheral support, memory layout and performance must be considered. For the program to be installed on an embedded system, many considerations must be made.

#### ***3.7.1 Raspberry Pi***

While the software is to be designed for a microcontroller, it was important to have a potential fallback option. In a future BINAR cubesat, it may be possible that a low-power computer such as a Raspberry Pi may be used as the controller. Had the program not been finished and the launch window was missed, future opportunities may present themselves.

The Raspberry Pi tested was an RP 3B, implementing the program on a Raspberry Pi was trivial as many libraries required for the program were cross-compatible.

Complications were found though when installing and compiling. Raspberry Pi does not have OpenCV pathing correctly set up for the language Rust. While OpenCV is not to be used on the microcontroller, it provides many effective functions such as adaptive threshold which runs on the GPU [41]. To fix the OpenCV install, the dev library needs to be installed and compiled, the required steps can be found in Appendix 4.

However, fixing the pathing problems allows OpenCV to be used. Compiling was also a noticeable problem, while the software ran effectively, compiling the code took far longer than expected, if this computer is to be used in the satellite, it would be important to push firmware updates as executables instead of raw code. The code ran on the Raspberry Pi in the same way as the PC.

#### ***6.7.2 Embedded Systems***

Embedded systems lack memory and storage. Because of this, they lack the capabilities for complex operating systems. A microcontroller is to only run a single binary program in its storage when turned on, it does not have a UI or way to compile code. This causes complications such as the lack of library support and the effects of cumulative bugs.

### 3.7.3 *Hardware Abstraction Layer*

Interfacing with a microcontroller requires the user to manually modify registers and memory addresses to gain access to the system's peripherals. For high-end systems, this may be difficult to achieve depending on the availability of the documentation. This is why *Hardware Abstraction Layers* or *HAL's* are made. A *HAL* is a piece of code that hides all the complex register management behind a set of human-readable functions making it easier for the writer and maintainer of the program. *HAL's* also can allow cross-compatibility between similarly designed chips such as microcontrollers of the same series. The star tracker library was written in the Rust programming language, however, the designated *HAL* for the STM32 series *STM32CubeMX* [27] was only designed for C and C++. Third-party *HAL's* were then investigated which were designed for Rust. These were the *STM32h7xx-hal* and *Embassy* [44, 45], both of these *HAL's* worked to some extent, however, they did not work when the clock speeds were configured, they also could not access the peripherals unless they were manually configured through a dummy program in *STM32CubeMX*. *Embassy* is currently in Beta and not explicitly ready for public use and *STM32h7xx-hal* lacked the documentation and support required. In the future *Embassy* may be useful, however, until then, these *HAL's* were deemed unsafe to run the program on. As *STM32CubeMX* was required for the execution of the program, it was decided to write the interfacing code in C and have the Rust library accessed through a wrapper. This approach worked allowing for fine control over the microcontroller while still using the Rust library.

### 3.7.4 Memory

The memory of a microcontroller is not handled automatically by an Operating System and cannot be disregarded like high-end hardware. Instead, it must be manually managed to ensure optimal memory management. The memory can be divided into the Flash and RAM.

Flash is read-only memory, this is used to store the program and any other information which is not to be readily modified. For the STM32H7, the memory can be manipulated, however, to do so it must first be erased which can take up to 20 seconds. The flash can also only be written 10 thousand times before it begins to degrade which limits the ability to run complex operations [28]. Open Satellite specifies that flash is unsafe in high radiation environments due to heightened degradation, however, it also states that the STM32H7 series controllers are proficient and that they should not be a concern [26].

SRAM is the system RAM, this memory can have every memory address read and written individually at high speeds without requiring the erasure of data. This type of memory unlike Flash memory, will not degrade when used and is considered safer in high radiation environments. This memory tends to be used for storing system variables. [26].

The chosen microcontroller has 1MB of SRAM and 2MB of Flash, however, this memory is fragmented, which can be seen in Table 6.

In this microcontroller, the Flash is divided into two, 1 MB chunks. As this is long-term storage, the image does not need to be stored here meaning it would only need to store the database and main program. With a  $\sim 18$ -degree lens, a region size of 10 degrees, a max number of stars in a region of 8 and a cut field of view of 1.3, the database size is reduced to 85 KB. With the hardware abstraction layer and main program, the required flash storage reaches 154 KB. This is 15% of the available memory and thus would be trivial to store in a single flash bank.

However, it may be ideal to store the database in a separate flash bank as the program or database may need an update during operations and it would take less bandwidth to update a single section than both at the same time.

It may also be ideal to store the database in flash storage and not move it to RAM during runtime as the flash operates at the same speed and the database should not be regularly written to. Moving the database to RAM would increase complexity, and startup time, and reduce the available storage of other variables. This would allow for the program to take advantage of the 256-bit reads as when reading the star pair database, the required information tends to be in a cluster.

The SRAM however, is far more fragmented than the flash, in RAM, there is only one bank that is large enough to store the image which is AXI-SRAM with 21KB to spare. The main program memory is then defaulted to be stored in DTCM which is likely capable of storing stack on. If the database was to be stored in RAM, it could then be stored in SRAM1 or SRAM2.

To accurately store the program in RAM, a linker file must be set up which tells the program what address to store variables. As Rust is being used as a static library, it cannot be attached to a linker file. Instead, the C code attached to a memory linker should reserve parts of its program for the Rust code and provide the reserved address. With this both the C and Rust code will have the instructions required for where to store information.

**Table 6**  
**Memory Locations STM32H7**

<b>Memory</b>	<b>Size</b>	<b>Domain</b>	<b>Speed</b>	<b>Notes</b>
AXI-SRAM:	512 KB	D1	240 MHz	Read/Write address: 64 bits
SRAM1	128 KB	D2	240 MHz	Read/Write address: 8, 16, 32 bits
SRAM2	128 KB	D2	240 MHz	Read/Write address: 8, 16, 32 bits
SRAM3	32 KB	D2	240 MHz	Read/Write address: 8, 16, 32 bits
SRAM4	64 KB	D3	240 MHz	Read/Write address 8, 16, 32 bits
Backup	4 KB	D3	240 MHz	Retained in Standby and Vbat mode.
ITCM	64 KB	D1-CPU	480 MHz	Instruction RAM
DTCM	2x64	D1-CPU	480 MHz	Interrupts or stack/heap.
FLASH - 2	1 MB	D1	240 MHz	Read address: 256
FLASH - 1	1 MB	D1	240 MHz	Read address: 256

*D1: High Performance (half max cpu speed).*

*D2: Communication Peripherals and Timers.*

*D3 reset/clock power management.*

*Adapted from [28]*

The size of each memory address must also be considered, for the image, the pixels are stored in 8-bit bytes, while the architecture is 32-bit words. By storing the image as a byte array, the image would be 4 times bigger than it should be at 1.97 MB, far exceeding the RAM potential. To reduce this, the pixels were stored as a bit field where each word stored 4 bytes. This method stops the memory issue, however, it requires extra clock cycles to execute the code as it needs to pull the specific byte out of the word as well as access the memory. This could be combated by a custom hardware abstraction layer which as the image is investigated, each word is read into DTCM memory and the word is iterated on.

### **3.7.5 DCMI**

The microcontroller also has Digital CaMera Interface or *DCMI*. This is a way of reading an image directly into memory using a parallel interface. This can read 140 MB/s and thus read the image into memory in 3.5 ms, however, the sensor can only run at 68MHz which would result in a running time of 7.2 ms per image. This has yet to be set up, however, it may improve the efficiency of the program, reducing the power requirements and making the read speed slightly higher depending on the peripheral clock speeds.

### **3.7.6 Multi-Core**

The microcontroller is a multi-core Cortex M7 with a Cortex M4 with the M7 running at 480MHz and the M4 running at 240MHz. Unlike multithreading, these cores run separate code and only communicate through common memory. Having the SRAM clock speed slower than the M7 makes the memory a bottleneck, making running both cores on a simultaneous task counterproductive [28]. However, running them separately where one is doing blob detection and the other is searching the database with the found blobs may save time. Other ways to optimise for multicore could include segmenting the database in half into multiple SRAM chips. By doing this, they both would be able to look at one-half of the memory. However, if all the searches are to be performed on a single side of the database, this would not offer any improvements.

## **3.8 Project Lifetime**

With the star tracker software ready to be integrated with the hardware, there were unexpected complications, the team working on the hardware had a power fault that they were unable to identify. Nearing the launch window, the BINAR team deemed the star tracker to be unsafe for this round of satellite launches. To secure the star tracker on the next launch window, attempts were made to commune with a student group who is to design a satellite that may be launched. While the star tracker is likely unnecessary for the payload, the team in charge of attitude determination have expressed interest as running a star tracker on the satellite may add a reputational advantage for sponsorships.

## 4.0 Results

Many tests needed to be performed to test the capability of the star tracker, these were decided to test reliability by running a long sample set and to test accuracy and response times through simulation. This was to be run on each platform to produce useful results.

### 4.1 Sky Wipe

To test the star tracker in a real-world example, in this test, the camera would point at the zenith and take photos as the stars moved. This *sky wipe* test would reduce the amount of atmospheric distortion making the test more reliable. To do this, a Raspberry Pi was configured with the Allied Vision Python480 camera and a power bank. The Pi was to take photos once every minute during the night, (see Appendix C for the setup guide). Once complete, the images were offloaded onto the PC to run the algorithm. Using 500ms exposure, full gain and a 16mm lens in the same way as the other tests, it was noticed that the success rate was less than 50% and the time to identify stars surpassed one minute. After some analysis, it was found that the error in the lens was far higher than it was tested for the single photo tests. It was so high that even the reliable plate solver *nova.astrometry.net* [36] was unable to identify most images.

The images were then viewed and compared with the single image tests, in these tests, the images were found to have far less noise than the images from the sky wipe. All tests were performed with max gain or voltage to reduce their exposure time, however, it is likely that increasing the gain increases the temperature of the sensor, thus making it more noisy [46]. In this circumstance, increasing the noise floor decreased the gap between foreground and background images, this can be seen in Figure 26. Having such a noisy image overcame the improvements made by swapping the Percent threshold to the Niblack threshold.

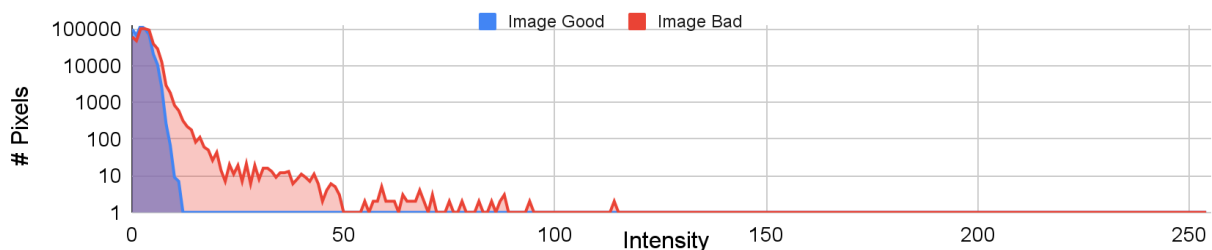


Figure 26: Pixel intensity histogram comparing an image on a prior test to an image on the sky wipe.

At the time of completion, the weather was not ideal to test the sensor, because of this it cannot be known the cause of the noise. It is possible that the sensor was damaged in previous testing or that as the season moves to Summer, the camera is warmer overall. When testing in Winter with the single photo shots, the camera would take multiple high gain, high exposure shots in quick succession and the noise was low. It is likely that the sensor on the sky wipe test was constantly capturing photos meaning after the first photo (1 minute), it would have taken 120 photos with high gain.



While this test was inconclusive and a complete failure, it highlighted an overlooked part of the star tracker hardware. In space, the star tracker would not be capable of regulating its heat. Knowing that the sensor cannot run at temperatures above a certain limit, it may be required to cool the sensor with an active cooling element. Cooling cameras is a method used by many machine vision projects. ThorLabs, an industrial camera manufacturer has products designed specifically for cooling cameras [46].

## 4.2 Test Parameters

To test the accuracy of the star tracker and speed, a set of test suites has been set up. Platforms with operating systems have undergone simulation tests that test the star tracker under each operation. The tests will follow 7 unless otherwise stated.

As there is no sky wipe test available to test the extremities of the algorithm, another test suit was set up. This test runs through a set of hand-picked images which have known errors. The star tracker is expected to fail at identifying the first triangle provided. This will provide a deeper insight into the loss of time when an error occurs.

There is also a test provided to the microcontroller which is designed to test its capabilities in standard circumstances to test the general performance of the microcontroller.

In all the tests, the exposure time was set to 500ms as this is what was the time duration of the test images provided.

*Table 7*

### *Test Criteria*

Field Of View	18 degrees
Region Size	10 degrees
Angle Tolerance	0.04 degrees
Region Num	8
FOV Reduced	1 / 1.3
Chunk Iterator	None
	Regional
	Equatorial 50 degree separation, x1.2 overshoot
	Declination 10 degree separation x2 overshoot
	Declination Random Pararity 10 degree separation, x2 overshoot

### 4.3 PC

Tests on the PC were mainly used to ensure that tests on the other target devices were running successfully as it was an easier platform to set up on. In this, all the optimisations were tests, however, all tests prior to the chunk iterators had their tests modified to ensure they were completed in a timely manner. All tests returned an accuracy below 0.01

*Table 8:*

*Early optimisations on a PC.*

Optimisation	Angle Tolerance	Induced Error	Success Rate %	Time (ms)
None	0	0	100	2.75
None	0.04	0	0	240,000
None	0.04	0.12	0	-
Triangle Iterator	0.04	0	0	20,000
Triangle Iterator	0.04	0.012	-	-
Region Reduction	0.04	0.012	91	71
FOV Reduction	0.04	0.012	100	80

*Contained is a set of optimisation which were tests with varying conditions.*

*Table 9:*

*Chunk iterators ran on PC*

Optimisation Tests (#)	*Exposure (ms)	*Thresholding (ms)	*Blob Detection (ms)	Tracking (ms)	Total (ms)	Success Rate (%)	
None	1484	500	10	8	58	576	92
Regional	1484	500	10	8	10.29	528	99
Crunch	1484	500	10	8	-	-	-
Equatorial	1484	500	10	8	5.63	523	100
Declination	1484	500	10	8	8.02	526	100
Declination	1484	500	10	8	6.59	525	99
Random Parity							

*Contained is a comparison of all the types of chunk iterators.*

## 4.4 Raspberry Pi 3B

It was decided that since the Raspberry Pi was a potential candidate for the star tracker, that it should be tested, specifically the 3B as it was available. As the Raspberry Pi has an operating system, it can run the program as if it were the PC (see Appendix D). This gives the ability to run side-by-side comparisons seeing how different clock speeds affect the performance of the program. Simulation tests were run to test the pyramid tracking mode performance, all parameters were set to the same as the PC test. A set of demo tests were run to find the time taken to perform blob detection and thresholding, when run on the same device, the performance of these are constant. The exposure time was set to 500ms which is the time tested in all of the demo tests. The success rate in these tests was ignored as they will have the exact same conclusion as the tests run on the PC.

The Raspberry Pi clock speeds can be modified by the user from between 600MHz to 1200MHz depending on the power saving mode, (see Appendix C). The tests were performed on both clock speeds giving another sample point examining how the clock speed affects the performance. The early optimisations were not included as they were already addressed in the PC section and it was clear that they would not be adequate methods.

### 4.4.1 600MHz

At 600MHz and idle, the Pi consumes approximately 5.26V and 0.2A totalling 1.1W of power. When running the simulation test, the current increases to approximately 0.3A which is 1.6W. Table 10 shows the results of the optimisations.

*Table: 10*

*Raspberry Pi 3B test at 600MHz*

Optimisation	Tests (#)	*Exposure (ms)	*Thresholding (ms)	*Blob Detection (ms)	Tracking (ms)	Total (ms)
None	1484	500	312	165	831.52	1809
Regional	1484	500	312	165	159.06	1136
Crunch	1484	500	312	165	-	-
Equatorial	1484	500	312	165	90.69	1067
Declination	1484	500	312	165	122.15	1099
Declination Random Parity	1484	500	312	165	97.57	1074

*Table showing the time in milliseconds of the star tracker simulation and demo tests on the raspberry pi @ 600MHz*

#### 4.4.2 1200MHz

When running at 1200MHz, the Pi consumes approximately 5.25V at 0.26A totalling at 1.365W. When the simulation is run, the current increases to 0.39A totalling 2.05W. Tests were run to test all the optimisations, these can be seen in Table 11.

*Table: 11*

*Raspberry Pi 3B test at 1200MHz*

Optimisation	Tests (#)	*Exposure (ms)	*Thresholding (ms)	*Blob Detection (ms)	Tracking (ms)	Total (ms)
None	1484	500	154	81	486	1221
Regional	1484	500	154	81	73.41	808
Crunch	1484	500	154	81	-	-
Equatorial	1484	500	154	81	42.26	777
Declination	1484	500	154	81	65.13	800
Declination Random Parity	1484	500	154	81	45.36	780

*Table showing the time in milliseconds of the star tracker simulation and demo tests on the raspberry pi @ 1200MHz*

## 4.5 Microcontroller

At the time of writing this thesis, the program has been ported to a microcontroller. However, the memory has not been correctly set up, and the camera is not configured. Instead, the microcontroller is sent an image through USB serial and told to run, the microcontroller then will run the star tracker algorithm with the given image and send it through serial when it has completed a section. While communicating progress takes time, it was deemed insignificant. It was also decided not to include the time to send the image through serial and instead use 7.2ms which is the time to run the DCMI interface when set up. The time to take a photo successfully was decided on 500ms, however, without an atmosphere this number is likely to be lower.

The tests taken were using the test conditions specified in Table 7. Each test was run at a different success rate. The fails to represent the number of star triangles that failed to be found in the test. The images were hand-picked to show what happens if a bright star is poorly identified by the blob detection.

*Table 12:*

*Microcontroller speed to failures.*

Fails	Exposure	Reading	Thresholding	Blob Detection	Tracking	Voting	Total
1	500	7.2	141	593	235	0	1, 476
3	500	7.2	140	597	4117	0	5, 361
4	500	7.2	141	597	568	0	1, 306
5	500	7.2	141	597	5, 355	2	6, 600
7	500	7.2	140	596	16, 926	-	18, 171
9	500	7.2	140	594	12, 932	2	14, 173
10+	500	7.2	141	602	50, 323	-	51, 573

*Table showing the how long a search takes when  $n$  failures occur*

## 4.6 Analysis

The results in these tests were to show the reliability, accuracy and response times, however, due to the failure of the sky survey, it is hard to make assumptions about real-world applications. In the sample photos taken when the sensor was working appropriately, the star tracker was able to recognise the image, however, this is also an inconclusive test as the images were taken of bright stars and would be an unfair representation of the dark areas of the sky.

An observation made from the tests was the accuracy of the star tracker, in the simulation tests, the error of the final output was never over the random error input into the system. In both the simulation and demo tests for the star tracker, the error never exceeded 0.1 degrees.

Although the complete star tracker was not fully set up, real-world analysis was performed and it found, the microcontroller can return a value with a response time of 1476 seconds. The results are considered to be reliable with the tests taken earlier in the year and the accuracy was consistently below 0.1 degrees. This has passed the expectations of a 10-second response time and 1-degree accuracy.

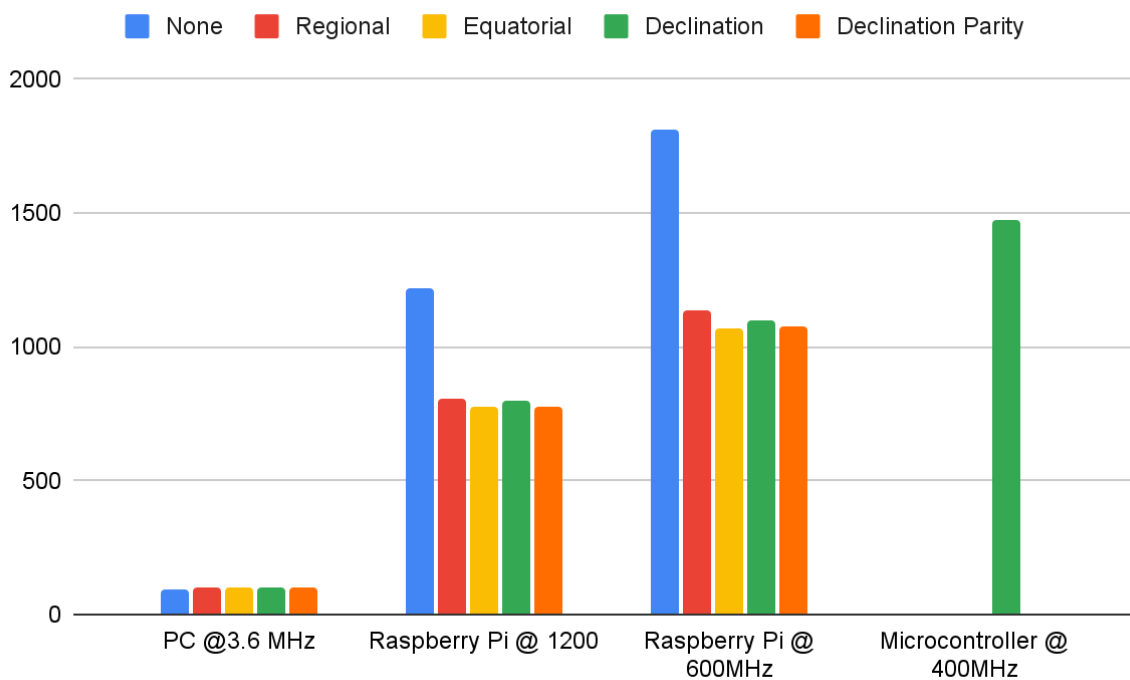


Figure 27: Performance of chunk iterators on different target platforms..

## 5.0 Conclusion

Contained in this thesis is the required steps to modify a Pyramid Star Tracker so that it is capable of running on low-end hardware such as a microcontroller. This is done through optimisations in bottlenecks in the code.

To achieve an acceptable performance, many optimisations must be performed. It was found that separating the code into separate modules and deciding what parts of the program should be improved is an ideal approach. It was also found that, although unintuitive, heuristic approaches were capable of providing significant gains in performance and storage.

For processing and recognition of the images, finding ways to avoid investigating every pixel can affect the performance greatly. Whether it is by skipping every second pixel or running the thresholding algorithm at the same time as blob detection, looping over a 2D array takes unnecessary time, where possible it should be avoided.

For running the star tracking algorithm, finding ways to reduce the size and order of complexity of the database will exponentially reduce the time of the system. Reducing the database can be through removing redundant stars and identifying regions where excessive stars are stored. Reducing the order of complexity could be through reducing the number of database lookups required by using sub-databases or prior knowledge.

It is also important to consider what type of hardware is available, if there is access to tools such as a DCMI parallel reader or a form of graphical hardware accelerator, time to solve images would be significantly reduced. The clock speeds and the choice of memory could impact the speed of reading the database or manipulating the image.

Ultimately for any program, if it needs to be optimised, investigation should start by seeing what takes the longest and finding an alternative to make it simpler.

With the optimisations designed in this thesis, the star tracker library went from an unusable program to one that can run on a microcontroller. With minor modifications, the code will be capable of providing useful attitude information. However, due to the unfortunate events occurring with the hardware, the star tracker will remain grounded indefinitely. To ensure the star tracker flies, the project will be continued so it will be ready when requested.

## **6.0 Future Work**

While the star tracker is now functional on a microcontroller and capable of fulfilling all the constraints, it should be improved before launch.

### ***Hardware***

As found in the results of the sky wipe, the sensor must be investigated to find the best way to reduce its noise in high heat, high gain environments.

The 16mm lens has been thoroughly tested and has promising results, however, to achieve this thesis, the other lenses were neglected. It would be important to run realistic space simulations to simulate how each lens performs without atmospheric distortion. It may also be ideal to investigate lenses in a higher price range which would be more accurate.

The CMOS sensor has not been tested for how radiation can cause damage to the pixels, it may be ideal to test how long the sensor lasts in a challenging environment. It may also be ideal to test if the sensor would survive capturing the sun with a high exposure.

### ***Software Implementation***

Considering the hot pixels from the CMOS sensor, it may be ideal to find a way to nullify the pixels so that cumulative error does not lead to an early failure of the spacecraft.

It may also be useful to investigate lens distortion in more depth. Considering atmospheric distortion, it was difficult to know how much error was accountable from the atmosphere and how much was attributed to the lens.

### ***Embedded System***

While the star tracker is now functional on a microcontroller, it is not designed optimally. The memory is not correctly set up for the microcontroller. At the present moment, the database is stored in the memory and without performing proper memory management, it is hard to identify where it is located. By having the database stored with program memory, the RAM is likely to overflow causing catastrophic failure. Because of this, the database must be moved to FLASH storage.

Interfacing with the camera is also not correctly set up, whether this uses the DCMI protocol or just peripherals, this needs to be implemented to run standalone.

The hardware is also not complete, to program the desired star tracker board, the hardware must be correctly set up and ready.



## 7.0 References

- [1] M. A. Swartwout, "A brief history of rideshares (and attack of the CubeSats)," 2011 Aerospace Conference, 2011. doi:10.1109/aero.2011.5747233
- [2] J. Puig-Suari, C. Turner, and W. Ahlgren, "Development of the standard cubesat deployer and a CubeSat class picosatellite," 2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542), Mar. 2001. doi:10.1109/aero.2001.931726
- [3] D. Selva and D. Krejci, "A survey and assessment of the capabilities of Cubesats for Earth Observation," *Acta Astronautica*, vol. 74, pp. 50–68, 2012. doi:10.1016/j.actaastro.2011.12.014
- [4] C. Leibe, "Star Trackers for Attitude Determination" *IEEE AES Systems Magazine*, vol. 10, no. 6, pp. 10-16, June 1995. doi: 10.1109/62.387971
- [5] MEMSIC, "±8 Gauss, High Performance 3-axis Magnetic Sensor," MMC5983MA datasheet, 2019, p. 2.
- [6] J. Wessles, "Infrared horizon sensor for CubeSat implementation," Stellenbosch University, uri: <http://hdl.handle.net/10019.1/103564>
- [7] K. Bolshakov, F. K. Diriker, R. Clark, R. Lee, and H. Podmore, "Array-based Digital Sun-sensor design for CubeSat Application," *Acta Astronautica*, vol. 178, pp. 81–88, 2021. doi:10.1016/j.actaastro.2020.08.005
- [8] B. Spratling and D. Mortari, "A survey on Star Identification Algorithms," *Algorithms*, vol. 2, no. 1, pp. 93–107, 2009. doi:10.3390/a2010093
- [9] J. Junkins, C. C. White III, and J. D. Turner, "Star pattern recognition for real time attitude determination," *The Journal of the Astronautical Sciences*, vol. 25, no. 3, pp. 251–270, 1977.
- [10] E. J. Groth, "A pattern-matching algorithm for two-dimensional coordinate lists," *The Astronomical Journal*, vol. 91, p. 1244, May 1986. doi:10.1086/114099
- [11] D. MORTARI, M. A. SAMAAN, C. BRUCCOLERI, and J. L. JUNKINS, "The Pyramid Star Identification Technique," *Navigation*, vol. 51, no. 3, pp. 171–183, 2004. doi:10.1002/j.2161-4296.2004.tb00349.x
- [12] B. B. Spratling and D. Mortari, "The K-vector ND and its application to building a Non-Dimensional Star Identification Catalog," *The Journal of the Astronautical Sciences*, vol. 58, no. 2, pp. 261–274, 2011. doi:10.1007/bf03321168

- [13] M. A. S. MOHAMMED, H. BOUSSADIA, A. BELLAR, and A. ADNANE, "Performance comparison of attitude determination, attitude estimation, and nonlinear observers algorithms," *Journal of Physics: Conference Series*, vol. 783, p. 012017, Jan. 2017. doi:10.1088/1742-6596/783/1/012017
- [14] H. D. BLACK, "A passive system for determining the attitude of a satellite," *AIAA Journal*, vol. 2, no. 7, pp. 1350–1351, May 1964. doi:10.2514/3.2555
- [15] E. Rangel Enger, 'Spacecraft attitude determination methods in an educational context', Dissertation, 2019.
- [16] S. Tanygin, and M. D. Shuster. "THE MANY TRIAD ALGORITHMS\*," *Avd. Astronaut, Sci* 127 (2007): 81-99.
- [17] G. Wahba, "A least squares estimate of satellite attitude," *SIAM Review*, vol. 7, no. 3, pp. 409–409, Jul. 1965. doi:10.1137/1007077
- [18] M. D. Shuster, "The generalized Wahba problem. The Journal of the Astronautical Sciences," *The Journal of the Astronautical Sciences*, vol. 54 no. 2, pp. 245-259, April. 2006.
- [19] P. B. Davenport, "A vector approach to the algebra of rotations with applications," *National Aeronautics and Space Administration*, vol. 4696, July. 1968.
- [20] M. D. SHUSTER and S. D. OH, "Three-axis attitude determination from vector observations," *Journal of Guidance and Control*, vol. 4, no. 1, pp. 70–77, Apr. 1981. doi:10.2514/3.19717
- [21] D. Mortari, "ESOQ-2 Single-Point Algorithm for Fast Optimal Spacecraft Attitude Determination," *Annual AAS/AIAA Space Flight Mechanics Meeting*, Feb. 1997.
- [22] Star\_Tracker\_Microcontroller. (2023), T. Creusot, Accessed: Jan 2023. [Online]. Available: [https://github.com/TomCreusot/Star\\_Tracker\\_Microcontroller](https://github.com/TomCreusot/Star_Tracker_Microcontroller)
- [23] S. N. D. A. Meera, "Enhanced Algorithm For Tracking number Plate From Vehicle Using Blob Analysis," vol. 3, no. 4, 2015. [Online]. Available: <https://ijournals.in/wp-content/uploads/2017/07/18.3415-Shashi.compressed.pdf>.
- [24] opengl. (2023), KHRONOS GROUP. Accessed: Nov 2023. [Online]. Available: <https://www.opengl.org/>
- [25] D. Arnas, M. A. A. Fialho, and D. Mortari, "Fast and robust kernel generators for star trackers," *Acta Astronautica*, vol. 134, pp. 291–302, 2017. doi:10.1016/j.actaastro.2017.02.016

- [26] P. Madle, "STM32H7 Radiation Test Report," Open Source Satellite, Hampshire UK, 2021 Accessed: Nov, 17, 2023. [Online]. Available: [https://www.opensourcesatellite.org/downloads/KS-DOC-01251\\_STM32H7\\_Radiation\\_Test\\_Report.pdf](https://www.opensourcesatellite.org/downloads/KS-DOC-01251_STM32H7_Radiation_Test_Report.pdf)
- [27] STM32CubeIDE. (2023), STMicroelectronics. Accessed: May 2023 [Online]. Available: <https://www.st.com/en/development-tools/stm32cubeide.html>
- [28] STMicroelectronics, "STM32H7474/757," STM32H747 datasheet, March 2023, p. 1.
- [29] STMicroelectronics, "STM32H755xI," STM32H755xI datasheet, Feb 2023, p. 1.
- [30] ONSemiconductor. "Python480," Python480 datasheet, May 2018, p. 1.
- [31] InveSense, "MPU-6881 Product Specification Revision 1.0," MPU-6881 datasheet, 2014, p. 9.
- [32] "Image Sensor Formats," Machine Vision Direct. <https://machinevisiondirect.com/pages/image-sensor-formats> Accessed: August, 2023
- [33] "Raspberry pi camera modules, camera solutions for Arduino and Jetson Nano. UVC camera modules and M12 lenses.," Arducam. <https://www.arducam.com/>
- [34] "help you protect you," HUPUU. <https://hupuu.com/>
- [35] "Light pollution map," www.lightpollutionmap.info, 2015. <https://www.lightpollutionmap.info/#zoom=14.31&lat=-29.1740&lon=114.9204&state=eyJiYXNlbWFWIjoiTGFiZXJCaW5nUm9hZCIIm92ZXJsYXkiOiJ3YV8yMDE1Iiwib3ZlcmxheWNvbG9yIjpmYWxzZSwib3ZlcmxheW9wYWNpdHkiOiJwLCJmZWV0dXJlc29wYWNpdHkiOjg1fQ==> (accessed May 26, 2023).
- [36] nova.Astrometry.net. (2023), US National Science Foundation, Canadian National Science and Engineering Research Council. Accessed: Nov 2023. [Online]. Available: <https://nova.astrometry.net/>
- [37] AMD, "AMD Ryzen 5 3600" Advanced Micro Devices Inc. <https://www.amd.com/en/product/8456>
- [38] Image Thresholding. (2023). [Online]. Available: [https://docs.opencv.org/4.x/d7/d4d/tutorial\\_py\\_thresholding.html](https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html)
- [39] "APPLICATION OF THRESHOLD TECHNIQUES FOR READABILITY IMPROVEMENT OF JAWI HISTORICAL MANUSCRIPT IMAGES," Advanced Computing: An International Journal vol. 2, no. 2, pp. 63, 64. [Online]. Available: <https://arxiv.org/ftp/arxiv/papers/1103/1103.5621.pdf>.

- [40] B. Liu, Y. Li, L. Wen, X. Zhang, and Q. Guo, "Effects of Hot Pixels on Pixel Performance on Backside Illuminated Complementary Metal Oxide Semiconductor (CMOS) Image Sensors," *Sensors*, vol. 23, no. 13, p. 6159, Jul. 2023, doi: 10.3390/s23136159.
- [41] opencv. (2023), OpenCV team. Accessed: Nov 2023. [Online]. Available: <https://opencv.org/>
- [42] matlab. (2023), Mathworks. Accessed: Nov 2023. [Online]. Available: <https://matlab.mathworks.com>
- [43] R P Stanley, "The Fibonacci lattice," *Fibonacci Quarterly*, vol 13, no. 3, pp. 215-232, Oct 1975
- [44] stm32h7xx-hal. (2023), stm32-rs. Accessed: Nov 2023. [Online]. Available: <https://github.com/stm32-rs/stm32h7xx-hal>
- [45] embassy. (2023), embassy-rs. Accessed: Nov 2023. [Online]. Available: <https://github.com/embassy-rs/embassy>
- [46] "Thermal noise and temperature control," [https://www.thorlabs.com/newgrouppage9.cfm?objectgroup\\_id=10773](https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=10773), Accessed: Nov, 2023

## Appendix A - Sky coverage

Field Of View Deg	Mag @ 2.00	Mag @ 2.31	Mag @ 2.62	Mag @ 2.94	Mag @ 3.25	Mag @ 3.56	Mag @ 3.88	Mag @ 4.19	Mag @ 4.50	Mag @ 4.81	Mag @ 5.12	Mag @ 5.44	Mag @ 5.75	Mag @ 6.06	Mag @ 6.38	Mag @ 6.69
10	0	0	0	0	0	0	0	0	0	0	0	0	0	1	3	4
12	0	0	0	0	0	0	0	0	0	0	0	0	1	3	5	10
14	0	0	0	0	0	0	0	0	0	0	0	1	4	7	8	11
16	0	0	0	0	0	0	0	0	0	0	1	2	5	9	16	20
18	0	0	0	0	0	0	0	0	0	1	1	4	8	12	19	25
20	0	0	0	0	0	0	0	0	0	1	2	6	10	17	26	36
22	0	0	0	0	0	0	0	0	0	1	3	8	14	20	34	48
24	0	0	0	0	0	0	0	0	1	1	3	12	19	25	40	57
26	0	0	0	0	0	0	0	0	2	3	6	14	21	31	50	70
28	0	0	0	0	0	0	0	0	2	3	6	17	26	39	59	86
30	0	0	0	0	0	0	0	0	2	6	10	21	31	47	69	103
32	0	0	0	0	0	0	0	1	4	6	11	23	35	56	83	118
34	0	0	0	0	0	0	0	1	4	7	13	27	43	62	96	135
36	0	0	0	0	0	0	1	2	6	11	17	33	49	75	110	153
38	0	0	0	0	0	0	1	3	7	14	21	37	52	80	121	174
40	0	0	0	0	0	1	2	3	8	15	23	42	63	94	137	193
42	0	0	0	0	0	1	2	3	10	17	26	49	69	103	150	214
44	0	0	0	0	1	2	3	5	11	19	30	53	76	113	163	234
46	0	0	0	1	1	2	4	6	13	21	32	58	87	126	180	256
48	0	0	0	1	1	2	4	6	13	22	35	64	92	133	195	280
50	0	0	0	1	1	4	7	9	17	26	39	71	98	147	218	316
52	0	0	0	1	1	4	8	11	20	30	44	78	105	159	236	338
54	0	0	0	1	1	4	8	12	22	34	49	85	120	173	257	372
56	0	0	0	1	2	5	9	14	24	38	53	92	133	189	275	401
58	0	0	0	1	2	6	10	15	28	39	56	99	148	208	297	423
60	0	0	0	1	3	7	12	17	31	45	65	106	157	229	320	459
62	0	0	0	1	3	7	13	20	34	49	70	114	171	244	347	495
64	0	0	1	2	3	8	15	22	37	54	76	126	180	259	365	530
66	0	1	1	2	4	9	15	23	39	58	80	133	191	279	389	560
68	0	1	2	3	4	10	17	26	42	63	87	140	202	293	412	598
70	0	1	2	3	4	11	19	30	45	69	95	150	215	311	434	635

*Least number of visible stars in a sky photo with a given field of view and magnitude.*

## Appendix B - Percent Four Star Sky Coverage

Field Of View																	
Deg	2	2.31	2.62	2.94	3.25	3.56	3.88	4.19	4.5	4.81	5.12	5.44	5.75	6.06	6.38	6.69	
10	0	0	0	0	0	0	1	1	2	6	13	28	47	75	93	99	100
12	0	0	0	0	1	1	2	3	6	13	25	48	74	94	99	100	100
14	0	0	0	1	1	3	6	11	22	44	70	92	99	100	100	100	100
16	0	0	0	2	3	5	10	19	38	64	88	99	100	100	100	100	100
18	0	1	1	2	4	7	16	30	56	80	96	100	100	100	100	100	100
20	0	1	1	4	6	13	25	44	72	91	99	100	100	100	100	100	100
22	0	1	2	5	10	17	34	57	84	97	100	100	100	100	100	100	100
24	0	1	3	8	13	24	45	69	92	99	100	100	100	100	100	100	100
26	0	2	4	11	17	32	56	81	96	100	100	100	100	100	100	100	100
28	1	3	6	14	23	41	67	89	97	100	100	100	100	100	100	100	100
30	2	4	8	18	29	51	76	92	99	100	100	100	100	100	100	100	100
32	2	6	10	22	36	60	84	95	100	100	100	100	100	100	100	100	100
34	3	7	13	26	44	67	90	97	100	100	100	100	100	100	100	100	100
36	4	9	15	31	50	75	93	98	100	100	100	100	100	100	100	100	100
38	5	13	18	36	58	83	97	99	100	100	100	100	100	100	100	100	100
40	6	16	22	42	65	88	99	100	100	100	100	100	100	100	100	100	100
42	8	18	25	49	73	92	99	100	100	100	100	100	100	100	100	100	100
44	10	20	28	56	79	95	100	100	100	100	100	100	100	100	100	100	100
46	12	24	34	63	83	98	100	100	100	100	100	100	100	100	100	100	100
48	14	27	39	70	86	99	100	100	100	100	100	100	100	100	100	100	100
50	16	30	45	76	90	100	100	100	100	100	100	100	100	100	100	100	100
52	18	35	51	80	92	100	100	100	100	100	100	100	100	100	100	100	100
54	20	38	56	84	94	100	100	100	100	100	100	100	100	100	100	100	100
56	21	42	61	87	94	100	100	100	100	100	100	100	100	100	100	100	100
58	23	47	67	90	96	100	100	100	100	100	100	100	100	100	100	100	100
60	25	51	72	93	97	100	100	100	100	100	100	100	100	100	100	100	100
62	28	56	77	95	97	100	100	100	100	100	100	100	100	100	100	100	100
64	30	62	82	97	98	100	100	100	100	100	100	100	100	100	100	100	100
66	31	66	86	98	99	100	100	100	100	100	100	100	100	100	100	100	100
68	34	70	88	100	100	100	100	100	100	100	100	100	100	100	100	100	100
70	36	73	92	100	100	100	100	100	100	100	100	100	100	100	100	100	100

*Percentage of photos of the sky which will have at least four stars given a field of view and magnitude.*

## Appendix C - Setting up Allied Vision Python480 camera for set and forget.

With a Raspberry Pi, a USB power bank and the Allied Vision Python480 camera, an autonomous program can be uploaded that takes a photo once every  $n$  seconds without needing a monitor to setup.

To prepare, first install VimbaX and install the Python package.  
Second, copy the following code into a python file.

### Sample Code: vimba\_runner.

```
'''

import sys
import os
import cv2
import time
from vmbpy import *

EXPOSURE = 600_000    # 0 to 1_000_000 where 1_000_000 is 1 second.
GAIN = 11.3           # 0 to 11.3 where 11.3 is the most gain
TIME = 60             # Delay between shots
capture = False
directory = 0         # The directory has a number to stop overwriting.
image = 0             # Each image is numbered in ascending order..
opencv_display_format = PixelFormat.Bgr8

def setup_camera(cam: Camera):
    with cam:
        try:
            stream = cam.get_streams()[0]
            stream.GVSPAdjustPacketSize.run()
            while not stream.GVSPAdjustPacketSize.is_done():
                pass
        except (AttributeError, VmbFeatureError):
            pass

def set_values(cam: Camera, exposure_time, gain):
    with cam:
        exposure = cam.ExposureTime
        exposure.set(exposure_time)
        cam.Gain.set(gain)

# Called when the photo is taken.
def frame_handler(cam, stream, frame):
    try:
        global image
        time.sleep(1)
        cam.queue_frame(frame)
        print(image, " ", frame)
        frame = frame.convert_pixel_format(PixelFormat.Mono8)
        cv2.imwrite("{}{}.png".format(directory, image), frame.as_opencv_image())
    except:
        -
```

```

def main():
    global directory # configuring global variable.
    global image # configuring global variable.
    with VmbSystem.get_instance() as vmb:
        cams = vmb.get_all_cameras()
        # Stops overriding folders by accident
        if 0 < len(cams):
            while os.path.exists(str(directory)):
                directory = directory + 1
            os.mkdir(str(directory))
            # Loop through a set of different gain and exposure settings
            while True:
                with cams[0] as cam:
                    set_values(cam, EXPOSURE, GAIN)
                    cam.TriggerSource.set("Software")
                    cam.TriggerSelector.set("FrameStart")
                    cam.TriggerMode.set("On")
                    cam.AcquisitionMode.set("Continuous")
                    cam.start_streaming(frame_handler)
                    cam.TriggerSoftware.run()
                    time.sleep(1)
                    cam.stop_streaming()
                image += 1
                time.sleep(TIME) # Wait for the next photo.

if __name__ == "__main__":
    main()
'''

```

## Low Power Mode

To prevent the computer from draining the power bank before morning, the CPU clock should be reduced to save power, this can be done with the following commands.

```

# Limit CPU speed.
# Remember to reset once done.
sudo vim /boot/config.txt

# Change this line from 1200 to 600 to get 600MHz:
arm_freq=1200

```



## SSH

To connect to the Raspberry Pi's SSH terminal, the Pi must setup a hotspot on launch. This can be done by clicking on the Wifi symbol on the top left of the desktop,

Advanced Options > Create Wifi Hotspot

Name the hotspot and press create.

Advanced Options > Edit Connections

Click on the hotspot name and click the gear symbol at the bottom of the window.

General > "Connect automatically with priority" <= tick this box.

Download *PuTTY* on the client computer and connect to the wifi.

Go to the wifi settings and copy the code under *IPv4 DNSservers*.

Enter this code under *Host Name (or IP address)* and 22 for *Port*.

Click *Open*.

Sign in with the user and password of the Raspberry Pi.

Find the target folder

```
nohup python3 vimba_runner.py
```

Press *control-c*. After the camera takes a photo, type *ls* to see if a photo appears.

This setup was tested with a Raspberry Pi 3B with a 10 Ah battery bank running at 5V 2.4A.

## Appendix D - Setting up star tracker library on Raspberry Pi

To setup the star tracker on a Raspberry Pi, the project must first be downloaded.

```
git clone https://github.com/TomCreusot/Star\_Tracker\_Microcontroller.git
```

To install Rust, from the root project directory, run:

```
Star_Tracker_Microcontroller$ sh setup.sh          # Select the automatic installation.
Star_Tracker_Microcontroller$ sudo sh setup.sh      # Sometimes this is also requiredvg.
```

OpenCV is installed on the Raspberry Pi, however, it is not setup to work with Rust OpenCV. The libraries for OpenCV Rust need to be setup, the fastest way is to use the following two lines:

```
sudo apt-get install libopencv-dev python3-opencv
sudo apt install python-opencv
```

To run the `star_tracker_nix` library, a set of large libraries need to be compiled. OpenCV requires 2GB's of RAM to install on the device. If the target device does not constrain enough memory, *Virtual RAM* must be setup. This uses some of the storage on the SD card as pretend RAM, it is also known as *Swap*. The following instructions set up the *Virtual RAM*:

```
sudo dphys-swapfile swapoff # Turns off Virtual RAM so it can be modified.
sudo nano /etc/dphys-swapfile # The file contains a variable specifying the swap size.
```

In `/etc/dphys-swapfile`:

```
Change this (100MB):
CONF_SWAPSIZE=100

To this (1000MB):
CONF_SWAPSIZE=1000
```

After file saved and quit:

```
sudo dphys-swapfile swapon # Turns on Virtual RAM after reboot
reboot
```

All relevant libraries and tools should be installed, running the following command should run a binary.

```
cargo run --bin {program name} {input parameters}
```