

UNIVERSITY OF TARTU

Faculty of Science and Technology

Institute of Technology

Andreas Ragen Ayal

# STAR DETECTION ALGORITHM FOR ESTCUBE-2 STAR TRACKER

Bachelor Thesis in Computer Engineering  
(12 ECTS)

Supervisors: Tõnis Eenmäe, MSc  
Margus Rosin, MSc

Tartu 2016

# Abstract

Attitude determination is a very important aspect in the mission of spacecraft. There are various methods of determining the attitude of a spacecraft, including, but not limited to, magnetometers, beacons and gyroscopes.[1]

Star Trackers are systems consisting of one or multiple image sensors connected to a processing unit. The role of the processing unit is to detect stellar bodies and identify the patterns made by the stellar bodies in the images captured and determine the attitude of the spacecraft by comparing the patterns to those found in a pre-compiled database.

This thesis describes an algorithm developed to identify stellar objects, improving a previously developed and implemented algorithm developed at the KTH Royal Institute of Technology.[2]

Using a running average, the new algorithm is able to adapt to a changing background level when identifying stars. This is important as a set constant background level can become obsolete over time as the brightness of the image background changes with the movement of the spacecraft.

With the use of a weighted average system, the new algorithm is able to calculate the Cartesian coordinates centers of stars within an image with sub-pixel accuracy. This improves on the 1 pixel accuracy from the old algorithm. This increase in accuracy allows for the calculation of centroids even if they occupy a small area on an image. Furthermore, an increase in the precision of the coordinates in stars may lead to an increase in the accuracy when determining attitude.

**CERCS:** P160 Statistics, operation research, programming, actuarial mathematics; P520 Astronomy, space research, cosmic chemistry; T111 Imaging, image processing; T320 Space technology.

**Keywords:** cubesat, FPGA, digital image processing, nanosatellite, running statistics, star detection, Star Tracker, weighted average

# Resümee: Tähetuvastus algoritm ESTCube-2 Tähevaatluskaamerale

Asendimääramine on satelliidi töös väga oluline. Selle määramiseks kasutatakse erinevaid seadmeid, näiteks magnetomeetreid, majakaid ja güroskoope.[1]

Tähetuvastuskaamerad on süsteemid, mis koosnevad ühest või enamast pildisensorist, mis on ühendatud arvutussüsteemiga. Arvutussüsteem tuvastab saadud piltidel tähtede kujutisi ja koostab neist kindla algoritmi järgi kujundid. Võrreldes tuvastatud tähtede mustrit sama algoritmi järgi tähekataloogi põhjal koostatud andmebaasiga, tuvastab süsteem satelliidi asendit.

Käesolev uurimustöö kirjeldab pildiandmete voogtöötlust võimaldavat algoritmi, millega tuvastada tähti pildi pealt, arendades edasi varem välja töötatud algoritmi.[2]

Algoritm arvutab jooksvalt pildi fooni, et eristada tähti taustast. Seda on vaja, kuna foon võib sõltuvalt satelliidi liikumisest ajas muutuda ning jäik lävend vananeb aja jooksul sensori kiirguskahjustuste kogunedes.

Algoritm arvutab tähtede koordinaate pildil piksli murdosa täpsusega, erinevalt varasemast algoritmist, mis tegi sama ühe piksli täpsusega. Satelliidi asendi määramise täpsus kasvab ligikaudu lineaarselt tähtede asukoha määramise täpsusega.

**CERCS:** P160 Statistika, operatsioonanalüüs, programmeerimine, finants- ja kindlustusmatemaatika; P520 Astronoomia, kosmoseuuringud, kosmosekeemia; T111 Pilditehnika; T320 Kosmosetehnoloogia

**Võtmesõnad:** digitaalne pilditöötlus, FPGA, jooksev statistika, kaalutud keskmine, kuupsatelliit, nanosatelliit, tähetuvastus, tähevaatluskaamera

## Table of Contents

Abstract	2
Resümee: Tähetuvastus algoritm ESTCube-2 Tähevaatluskaamerale	3
1. Introduction	5
2. Attitude Detection Methods and Systems	7
2.1. Magnetometers.....	7
2.2. Sun Sensors and Beacons.....	7
2.3. Gyroscopes.....	8
2.4. Star Trackers.....	8
3. Field-Programmable Gate Arrays	9
3.1. Architecture.....	9
3.2. Programming.....	11
4. Detection of Stellar Sources	13
5. Detection of stellar sources	15
5.1. Pixel Thresholding.....	15
5.2. Star Detection.....	19
5.3. Algorithm Assessment.....	20
6. Results and Discussion	22
6.1. Square-Root Algorithm Comparison.....	22
6.2. Algorithm Assessment.....	22
6.3. Synthesis for FPGA choice.....	26
7. Conclusion	27
8. Kokkuvõte	28
9. Acknowledgements	29
10. References	30
Appendix 1: Algorithm Block Diagram	32
Appendix 2: Source Code - Newton-Raphson and Binary Method Comparison	33
Appendix 3: Source Code - Star Tracker in Python	37
Appendix 4: Source Code – arith.vhd	43
Appendix 5: Source Code – sensor.vhd – VHD Package	46
Appendix 6: Source Code – thresholding.vhd – VHD Source	47
Appendix 7: Source Code – blobber.vhd – VHD Source	50
Appendix 8: Source Code – thresh_sim.vhd – VHDL Simulation	56
Non-exclusive license to reproduce and make this thesis public	59

# 1. Introduction

Attitude determination is a key part of every spacecraft. It allows the spacecraft to correctly point its high gain antenna towards the Earth, and for spacecraft with controllable solar arrays, it helps point them in the optimum attitude.

Traditionally, Attitude Determination Systems (ADSs) consist of an absolute reference sensor and an inertial sensor. The absolute attitude sensor determines the direction of the spacecraft a spacecraft is pointing at, relative to an origin, at regular intervals, while the inertial sensor is used to measure changes between the previously mentioned intervals. Among the sensors used in ADSs are magnetometers, radio frequency beacons, horizon sensors, Sun sensors, gyroscopes, orbit models and star trackers.[1]

The focus of this thesis, the star tracker, is essentially an electronic camera connected to a processing unit. The role of the processing unit is to locate and identify stars from the electronic image and determine the orientation of the spacecraft.[3]

During the initial stages of development, the conventional star tracker was a fairly large component, weighing tens of kilograms and using many watts of power.[4][5] However, as more smaller and more affordable components are developed, and with the rising popularity of smaller forms of satellites, such as the nanosatellite, there is a need for smaller, more power efficient star tracker systems.

This thesis describes the development of one part of the star tracker system (STS) which will be used in the ESTCube-2 satellite. The proposed STS for the satellite contains a camera, a field-programmable gate array (FPGA), and, depending on the particular use case, a microprocessor. In this particular setup, the role of the FPGA is to detect stars from the image of the celestial sphere and send their coordinates to the microprocessor, which will determine the attitude of the spacecraft.

It is important to mention that a similar system has been developed, at the KTH Royal Institute of Technology, in the year 2014. The system in question uses an image sensor similar to others used in the ESTCube and Tartu Observatory satellite projects, an Actel Proasic 3 FPGA and a Smartfusion2 System-on-a-Chip (SOC). The system uses the FPGA to control the image sensor and parse through the image data to obtain star coor-

dinates, with an accuracy of one pixel. The SOC uses a star database in memory to calculate the attitude of the satellite.[2]

This thesis uses the STS developed at the Royal Institute of Technology as a base and improves on it, by obtaining the coordinates with an improved accuracy.

## **2. Attitude Detection Methods and Systems**

Attitude, as defined by the online Merriam-Webster Dictionary, is the position of an aircraft or spacecraft determined by the relationship between its axes and a reference datum (as the horizon or a particular star).[6] Attitude detection, therefore, has to do with using sensor data in order to calculate and determine the attitude of a craft. In this thesis, attitude and attitude detection shall be in the context of spacecraft.

Following are some of the methods and sensors used to determine the attitude of a spacecraft.

### **2.1. Magnetometers**

Magnetometers are instruments used to measure the strength and direction of low strength magnetic fields (less than 1 mT). [7]

Given that the Earth can be considered to be a very large dipole magnet, it is possible to determine the attitude of an object in space by measuring the magnetic field of the Earth in relation to the given object. There are, however, factors that change the shape and strength of the magnetic field, such as currents in the Earth's core and solar wind, to name a few. These phenomena limit the accuracy of magnetometers. They are, however, attractive solutions for low budget missions and missions which do not require a high degree of accuracy. They can also be used as backup methods for more complex ADS solutions.[8]

### **2.2. Sun Sensors and Beacons**

Sun sensors determine the direction a spacecraft is orientated, in relation to the Sun. Radio beacons, just like the Sun, are used as a reference point, though they are considerably less accurate. With only one point of reference, only a single pointing direction can be determined.[1] One can deduce that with increased points of reference, the attitude of a satellite can be calculated with increased precision.

## 2.3. Gyroscopes

These are devices that can measure angular velocity. The technology behind this is varied, ranging from piezoelectric plates to laser rings[9]. Angular velocity,  $\omega$ , is given by the equation:

$$\omega = \frac{d\theta}{dt}, \quad (1)$$

where  $d\theta$  is the change in attitude over the period of time,  $dt$ . Knowing the angular velocity, it is possible to calculate the attitude of a spacecraft, in relation to a point of origin.

## 2.4. Star Trackers

The main focus of this thesis, the star tracker determines the attitude of a spacecraft based on images taken of the celestial sphere. This is done by mapping the celestial bodies detected from an image, constructing patterns according to a set algorithm and comparing these patterns to known stellar patterns in a database, finding the best match. The components necessary for this are an image source, most commonly a dedicated camera system, a processor, to detect stars in the image, and a star-pattern database, with which the processor can identify the star patterns detected. This thesis focuses on the feasibility of using an FPGA as the processor used to identify and determine the properties of stars in an image.



## 3. Field-Programmable Gate Arrays

### 3.1. Architecture

Commonly known by their acronym, FPGAs are integrated circuits designed to be configured after manufacture. They are made up of logic blocks, also known as cells or slices, placed in an infrastructure of interconnections. These can be programmed at three levels: the function of the logic cells, the interconnections between cells and the inputs and outputs. This makes them very powerful tools, as they are programmed on a hardware level, creating specialized circuits within the semiconductor chip.[10] Following is a brief overview of how programming is carried out on the three levels.

The programmable logic data for the FPGA is contained in its Configurable Logic Blocks (CLBs), commonly known as logic cells. Each manufacturer uses different architecture for the CLBs used in their FPGAs, with different CLB architectures offering different functionality. As such, CLBs can be classified into two: fine-grain and coarse-grain logic blocks.

Fine-grain CLBs are small in size, with some made up of only one transistor pair. These CLBs are used to implement relatively small or simple functions. In order to implement more complex functions, more CLBs would need to be used, making the speed of the process more dependent on inter-cell connectivity. In most cases, fine-grain architecture is considered slower than coarse-grain architecture.[11]

Coarse-grain CLBs are larger in area, made up of multiple larger logic elements, for example the Xilinx 4000 logic block, developed by the Xilinx Corporation contains flip-flops, for clocked storage, multiplexers, to route the logic within the block, two 4-input lookup tables (4-LUT), feeding their output into a 3-LUT. The 4-LUTs can be used to produce any possible 4-input Boolean equation. The system of three LUTs can therefore produce a wide variety of outputs for up to nine inputs.[12]

As individual logic cells, depending on their coarseness, offer simple functionality, it is necessary to combine them, in order to develop more complex functions. The connections between logic cells, that enable such combinations, as well as connecting the logic

cell to input/output pins controlled by transistors. The links can be programmed by various methods, with the more conventional methods being SRAM, Antifuse and Floating Gate programming. This is commonly referred to as *routing*.

SRAM programming utilizes static bits of RAM to control the routing. This allows for the FPGA to be reprogrammed as needed, a very useful feature in development and debugging. As RAM is volatile, this means that the FPGA must be reprogrammed with each power cycle. In most cases, this is solved by loading the instruction set from an external memory source, such as EEPROM. SRAM, however, requires a large area, requiring around 5 transistors per SRAM cell.[11]

Antifuse programming utilizes devices called *antifuses*. Conventional fuses conduct electricity until a large enough current causes them to melt from the heat generated. In the case of antifuses, there is a small, non-conductive link between two conductors, separated by an insulator. On applying a large voltage across the link, it melts. Conductor material migrates across the link, creating a conductive path connecting the two conductors.[12] This process is not reversible. While the link areas can be very small, in order to melt the fuse, high power transistors need to be placed into the circuitry, as well as isolation features to protect low-voltage components.[11]

Floating gate programming uses technology found in erasable EPROM and EEPROM devices. On programming, transistors are “disabled”, by injecting charge onto the floating gate of the transistor (See Figure 1). This increases the threshold voltage of the transistor, effectively turning it off.

In the case of UV erasable EPROM transistors, this injected charge can be removed by exposing the gate to UV light. As with SRAM technology, this can be reprogrammed. An advantage over SRAM technology is the fact that floating gate FPGAs do not need to be reprogrammed after each power cycle.[11]

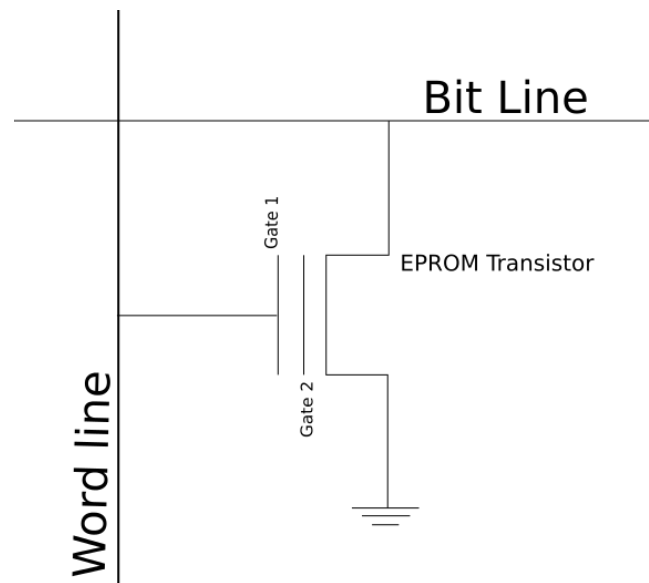


Figure 1: EPROM Transistor[11]

The technologies used play a key role in the selection of FPGA for an application. Some applications, such as military and space technology, require that a device be non-volatile and not susceptible to changes from radiation or power fluctuations.[12]

### 3.2. Programming

The fact that FPGAs are programmed at a hardware level allows for designers to create independent and functionally different components and modules within the FPGA. As such, the first process of programming an FPGA requires the developer to describe the hardware to be implemented, both in terms of the elements and modules to be created and used, as well as how they connect to each other. This is done with a hardware description language (HDL). HDLs allow for the description of multiple levels of hardware, ranging from switch and gate level programming up to algorithm description. It is important to note that HDL code is essentially an accurate description of every aspect of the design, meaning the design can be simulated without having to commit to a specific FPGA.

The most common HDLs are VHDL (VHISC Hardware Description Language; VHSIC – Very High Speed Integrated Circuit), Verilog and SystemVerilog. VHDL and Verilog are open standards and maintained by groups of the Institute of Electrical and Electronic Engineers as IEEE-STD-1076 and IEEE-STD-1364 respectively. While the languages

are different, they all offer similar functionality.[12]

For a hardware description to be implemented and programmed on an FPGA, the HDL code needs to be synthesized. This creates a gate level equivalent of the high level HDL code. After determining constraints, such as inputs, outputs and timing, this can be converted to a bit stream, to be programmed onto the FPGA. Many synthesis software solutions also optimize the HDL code for the particular vendor and FPGA chosen.[12]

## 4. Detection of Stellar Sources

A star tracker can be considered to have two key parts: the camera and the processor. The camera takes images of the celestial sphere, containing stars. The processor takes the image files from the camera and determines the attitude. This process can be further divided into two parts: star detection and stellar pattern recognition. The former takes the aforementioned images and detects the stars, calculating their centroid coordinates in pixels as well as their intensities. The latter takes these coordinates and intensities and matches them to star patterns found in its database. The more accurate the coordinates calculated, the easier it is to match the stars and the more accurate the estimation of attitude. Another way of improving the accuracy of the system is to increase the amount of stars detected in an image. This can be accomplished by increasing the field of view or by creating a system that is capable of detecting fainter stars, given that there are more faint stars in the celestial sphere than there are bright ones (approximately 2.5 times per unit magnitude, see the Yale Bright Star Catalog[13]). It is important to stress that though both parts are equally important, this thesis will not handle the latter.

The camera module for the ESTCube-2 star tracker is still under development and the choice of which image sensor to use has not been made. The likely candidates are the 1.2 mega-pixel “MT9M001C12STM”[14] and the 5 mega-pixel “MT9P031”[15], both developed by ON Semiconductor. Bearing this in mind, some assumptions were made to define this thesis:

1. The input image would be transferred to the processor in a stream, pixel-by-pixel, row-by-row, starting from the top left.
2. The image sensor uses monochrome pixels, with pixel values ranging from 0 to 65535 (in other words, up to 16-bit ADC resolution).

During development of the algorithm, it was important to make sure that the image size and pixel resolution could be changed with relative ease, as the algorithm could be used with various sensors. Thus the number of pixels per row, the number of rows as well as the maximum pixel value should be easily changeable, to cater for different sensors. In addition, it is worth bearing in mind that the optics and sensors chosen will determine

the angular resolution of the image, as well as the field of view.

During the initial stages of development, it was discovered that a star tracker similar in architecture to the one being developed had already been created at the KTH Royal Institute of Technology. It was therefore decided to use it as a base model and make improvements in specific areas, in order to increase the accuracy.

The first improvement was with the thresholding of pixels. Thresholding in this particular context has to do with the intensity level a pixel has to exceed in order to be considered as part of a star. The aim being to define which pixels were to be considered as part of a star and which to be considered as part of the background. The star tracker developed by KTH used a constant threshold, that could be set and changed on command. All pixels were compared with the threshold and pixels not exceeding the threshold were ignored.[2] The algorithm analyzed in this thesis uses running statistics to detect pixels that are brighter than the image background. This should improve on the detection of stars, especially those with low intensities, as well as ensure that the algorithm can still work despite a change in the background level. This can come into play when a very bright celestial body, such as the Sun or the Moon, is close enough to considerably change the brightness of the background.

The second improvement implemented had to do with improving precision of the coordinates calculated, in order to improve the accuracy of the attitude determination. The KTH tracker gives star coordinates with a precision of one pixel.[2] This should be improved on, calculating the coordinates of stellar centroids with a sub-pixel accuracy.

As this algorithm is to be used to detect the attitude of spacecraft in real time, it should be able to detect the stars and detect their center-points as fast as possible, preferably as fast as the image sensor is capable of compiling new images. The speed of the algorithm is important as it allows for faster reaction to motion.

## 5. Detection of stellar sources

### 5.1. Pixel Thresholding

When detecting stars, it is necessary to set up a threshold that differentiates the star from the background. In an ideal situation, a star would be a point-spread function with a cross section similar in shape to a Gaussian function. The background of such images are not black voids, but rather include photons from unresolved astronomical objects, bias level, read noise, thermally generated dark current and other sources. These unwanted photons must be accounted for and removed from the image, in order to accurately calculate stellar positions from the image.[16]

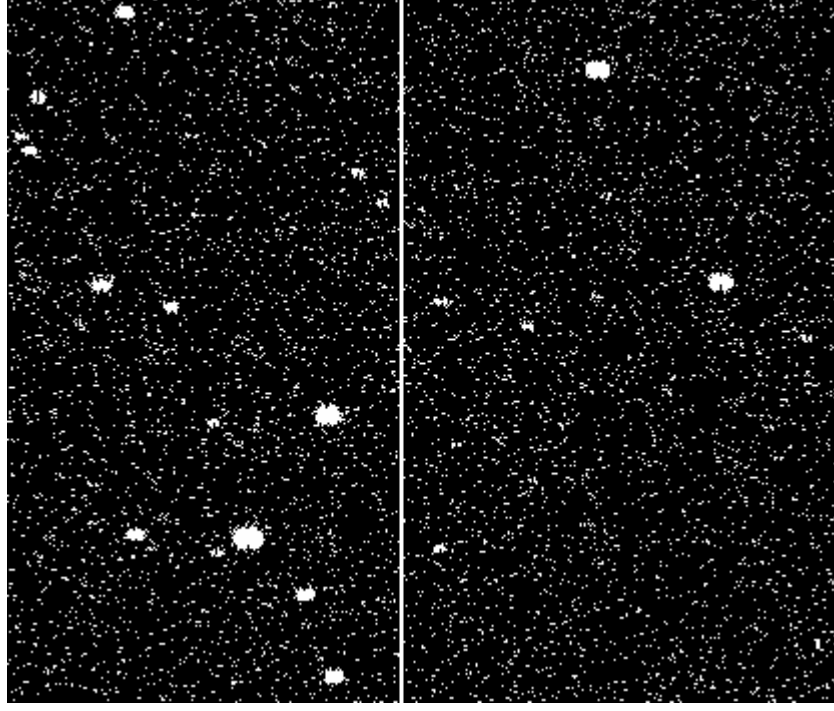
Readout noise, also called read noise, is usually quoted as the number of extra electrons introduced per pixel into the final readout of the device. Read noise consists of two parts. The first is the conversion from an analog signal to a digital number. The converter produces a statistical distribution of possible values centered around a mean. Thus, even if the same signal were to be quantized twice, the values obtained may be different. The second part comes from the introduction of spurious electrons from the sensor electronics. These two factors create additive noise in the output for each pixel. [16]

Dark Current, as a source of noise, comes from the electronics within the image sensor. Every material at a temperature above absolute zero experiences thermal noise. In an image sensor, this means that if the temperature fluctuations are high enough, electrons will be freed from the valence band of the semiconductor and be collected in pixels. For this reason, many astronomers and designers seek better cooling technology.[16]

Imperfections in the electronics are not the only sources of noise in an image. Bombardment from cosmic rays may fully saturate multiple pixels in an area. Over long periods of time, such bombardments can affect the work of an image sensor for a considerably long period of time or even cause permanent damage[16].

Figure 2 shows an example image. The white dots uniformly covering the image are as a result of dark current and readout noise. The clumps of white pixels are stars and the

white vertical streak is from an array of damaged pixels.



*Figure 2: Noisy input image with broken pixels.  
Image courtesy of the Tartu Observatory*

While the health of the image sensor is not within the scope of this thesis, it is important to bear in mind while developing an algorithm meant to process such images, especially as such bombardment can lead to detection of false objects.

One of the more easier ways of detecting the start of a bright star against a dark background is to detect a sharp rise in pixel intensities. In other words, simply comparing each pixel with the previous to find a large change.

$$\partial \text{pix}_i = |\text{pix}_i - \text{pix}_{i-1}| \quad (2)$$

While the act of simply calculating the difference and comparing it to a set threshold is fairly easy, it would not be able to adapt to a changing background and noise levels. If the differences between consecutive pixels in the background increased, the hard-coded threshold would not be useful. Also, if the background intensity level increased, it would make the difference between the pixels in a star and the surrounding noisy background decrease. As the profile of a star can be approximated by a Gaussian function[16], if the slopes are too shallow, a threshold based on calculating the difference between consecu-



tive pixels would filter such a star out. More importantly, this technique does nothing to remove the noise present in the image.

A more accurate technique, which is also the technique used in this thesis, is to calculate the arithmetic mean pixel value of the image and to use standard deviation of this calculated mean as the threshold, with variations of over five times the standard deviation considered as part of the stars. The factor of five was chosen based on the assumption that noise levels above the mean would follow normal distribution. Thus the chances of a noisy pixel rising above the average by more than 5 sigma are approx. 0.023%. While raising this threshold even further would further decrease the chances of a noisy pixel exceeding the threshold, it also reduces the number of pixels detected for each star.

The standard deviation is described by the equation:

$$S_N = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2} , \quad (3)$$

where  $S_N$  is the standard deviation,  $N$  the number of pixels,  $x_i$  the intensity value of the pixel of index value  $i$  and  $\bar{x}$  is the arithmetic mean of all values of  $x_i$ :

$$\bar{x} = \frac{\sum_{i=1}^N x_i}{N} \quad (4)$$

To a processor or FPGA, the summation part of the arithmetic mean is a trivial function. Division, on the other hand, is much more complicated, with most implementations requiring the function to be carried out in a loop[17]. As a suitable division function was not in the standard VHDL library, the author created a division function based on binary division[18]. Source code, in C, can be found in Appendix 2.

Calculating standard deviation requires a second summation, a second division for calculating the variance, the first being for calculating the mean, and a square root. Conventional square root functions use floating point numbers, which are very expensive, in terms of resources used, to implement on FPGAs.[19]

Thus, it was necessary to use functions that would allow for fast but accurate calculation of the mean, variance and standard deviation. The average and variance were calculated

using the following recurrent equation:

$$M_k = M_{k-1} + \frac{x_k - M_{k-1}}{k} , \quad (5)$$

$$S_k = S_{k-1} + (x - M_{k-1}) \times (x - M_k) , \quad (6)$$

where  $M_k$  is the arithmetic mean calculated after the inclusion of element  $x_k$ .  $x_k$  is the intensity of pixel  $k$ .  $S_k$  is the variance calculated after the inclusion of element  $x_k$ .

These were chosen as they required few multiplication and division functions. They also do not lead to a situation where  $S_k$  is negative, a problem that multiple other algorithms face.[17]

For the recurrent equation (6), the standard deviation is calculated using the equation: [17]

$$\sigma = \sqrt{\frac{S_k}{k-1}} , \quad (7)$$

where  $\sigma$  is the standard deviation and where  $k$  is greater or equal to 2.

It was necessary to find an algorithm for calculating the square root of a number. Of the most common square root functions there are few that can feasibly be implemented in FPGA architecture. This is because many use multiple division functions and floating point arithmetic to calculate the square root. For example, the Newton-Raphson method, described in [20], is a function that iteratively converges onto the square root of an input number. While it can be implemented to work with integers or fixed-point numbers, it uses a division function in each iteration. The author chose to use a digit-by-digit algorithm, commonly known as the binary method or the square-root by abacus algorithm[21]. A comparison of these two functions was carried out in order to determine how many more iterations were made in the Newton-Raphson method than the binary method. The results of this comparison are discussed later in this thesis. The source code for the test is available in Appendix 2.

## 5.2. Star Detection

The threshold process is meant to differentiate the background from the star pixels. This is done by calculating the background level and then subtracting this level from every pixel, forcing those that were below the threshold to zero. This leaves stars as the only values above zero. From here on, we shall refer to adjacent groups of bright pixels (the pixels that passed the threshold and are therefore not equal to zero) as *blobs*.

As the image from the sensor is streamed in, pixel-by-pixel, row-by-row, blobs are actually sequences of consecutive bright pixels, separated by longer sequences of dark pixels, as illustrated in Figure 3 below.

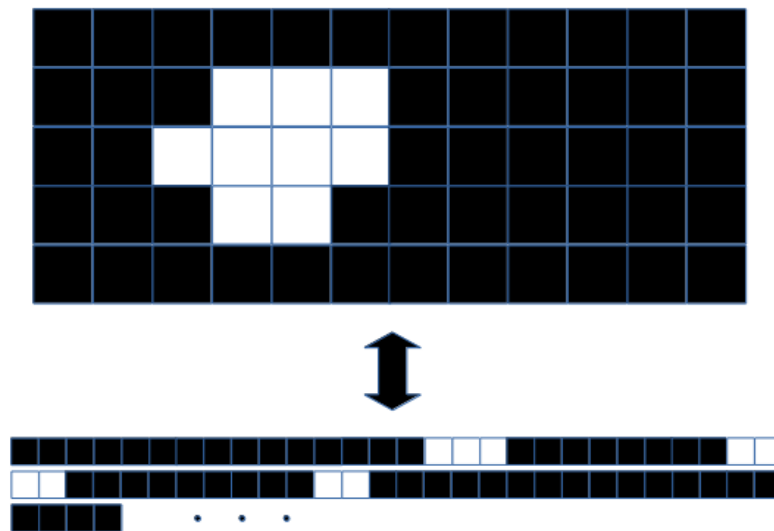


Figure 3: Image to Stream representation

The top part of Figure 3 is a representation of how a star might look in an image, with the blue lines dividing the pixels within the image. The bottom, a representation of how the pixels would be streamed out of the image sensor, starting from the top right.

Bearing this in mind, the algorithm goes as follows.

1. If the pixel is bright (it exceeded the threshold), go to 3.
2. If there are any blob-less pixels, then create a new blob with all the blob-less pixels (there should now be 0 blob-less pixels). Go to 8.
3. If the pixel above this (oldest in the pixel buffer) is not bright, go to 5.

4. Add this pixel, and any blob-less pixels to the blob containing the pixel above this one. Go to 8.
5. If the pixel on the left (newest in the pixel buffer) is not bright, go to 7.
6. If the pixel on the left is part of a pixel, add this pixel to the blob with the pixel on the left.
7. Add this pixel to the blob-less pixels.
8. Add this pixel to the buffer, in the same index as the x-coordinate of this pixel. (This overwrites the oldest value in most cases).

The block diagram of the algorithm above is in Figure 8, in Appendix 1.

Blob data in memory contains the sum of the values for every pixel included, the sum of weighted coordinates and a bit map of every pixel in the blob. This bit map is used when checking whether a given pixel is in the blob. After the entire image has been parsed, the coordinates for each blob are calculated with the equation:

$$COORD = \frac{\sum_{i=1}^{N_x} coord_i * intensity_i}{\sum_{i=1}^{N_x} intensity_i}, \quad (8)$$

where  $coord_i$  and  $intensity_i$  are the x or y-coordinates and intensity of pixel<sub>i</sub>.

### 5.3. Algorithm Assessment

The algorithm was developed in the Python programming language, with the source code available, in Appendix 3. The language was chosen due to its ease of use, and functional assessment was also carried out using Python. In order to test the algorithm, random positions for 80 stars were generated using the “*starlist*” command from the *art-data* package from the IRAF astronomical data analysis package[22]. These positions were constrained to be inside the 5 megapixel MT9P031 image sensor pixel array (i.e. 2592 x 1944 pixels). Using this star list, images were generated, adding noise based on vendor-published noise parameters of the sensor as well as Poisson noise, using the “*mkobjects*” command, from *artdata*. for accuracy testing. As the generated images

were in FITS format, using the the Python module “pyfits”[23] was needed. As the algorithm would later need to be converted to VHDL or Verilog, the module “myhdl”[24] was also used, to simulate the VHDL objects BIT\_VECTOR and STD\_LOGIC\_VECTOR whenever needed. Various noise levels were simulated by changing the exposure time for the simulated images.

After the algorithm had calculated the centroids of the stars in each image, the calculated coordinates were compared with the actual coordinates. The arithmetic mean of the differences was plotted against the signal-to-noise ratio (SNR) of each image. The SNR was calculated with the equation[16]:

$$\frac{S}{N} = \frac{N_T}{\sqrt{N_T + n_{pix} \times (N_S + N_R^2)}} \quad , \quad (9)$$

where  $N_T$  is the total number of photons detected in a star,  $n_{pix}$  is the number of pixels detected in a star,  $N_S$  is the background level and  $N_R$  the readout noise.

## 6. Results and Discussion

### 6.1. Square-Root Algorithm Comparison

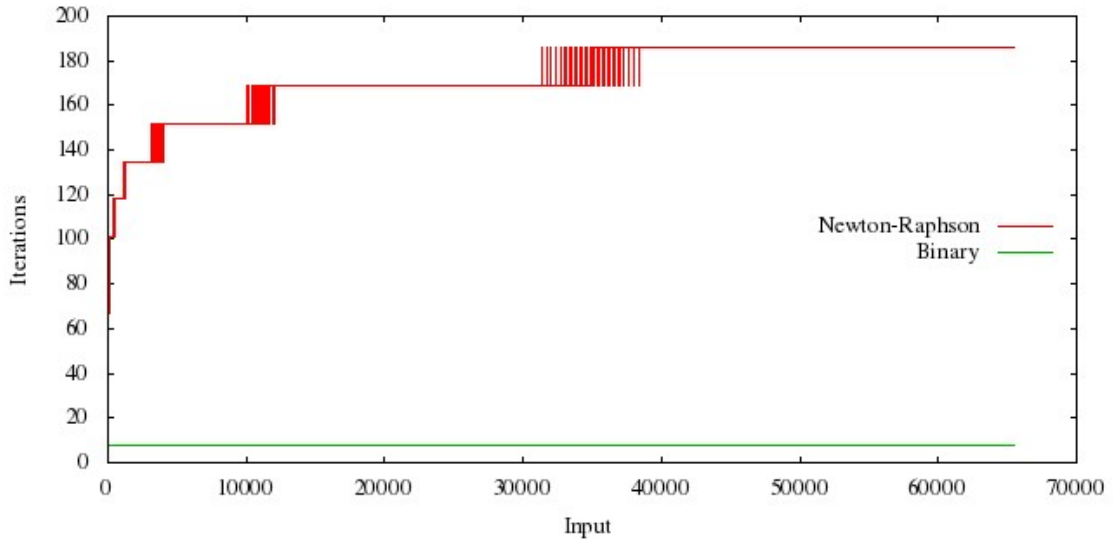


Figure 4: Comparison between Newton-Raphson method iterations to Binary method iterations

From Figure 4 above, it is evident that the number of iterations the Newton-Raphson method goes through is considerably higher than those in the binary method. Looking at the source code (Appendix 2), this is explained by the fact that each iteration in the algorithm contains a division, which is another iterative function. The number of iterations performed by the Newton-Raphson method is proportional to the input. The larger the number, the more iterations. The total number of iterations is therefore a product of the number of iterations carried out by the division function and the number of iterations carried out by the Newton-Raphson algorithm.

### 6.2. Algorithm Assessment

Based on the first assessment test described in the previous section, the graph in the upper panel of Figure 5 was compiled.

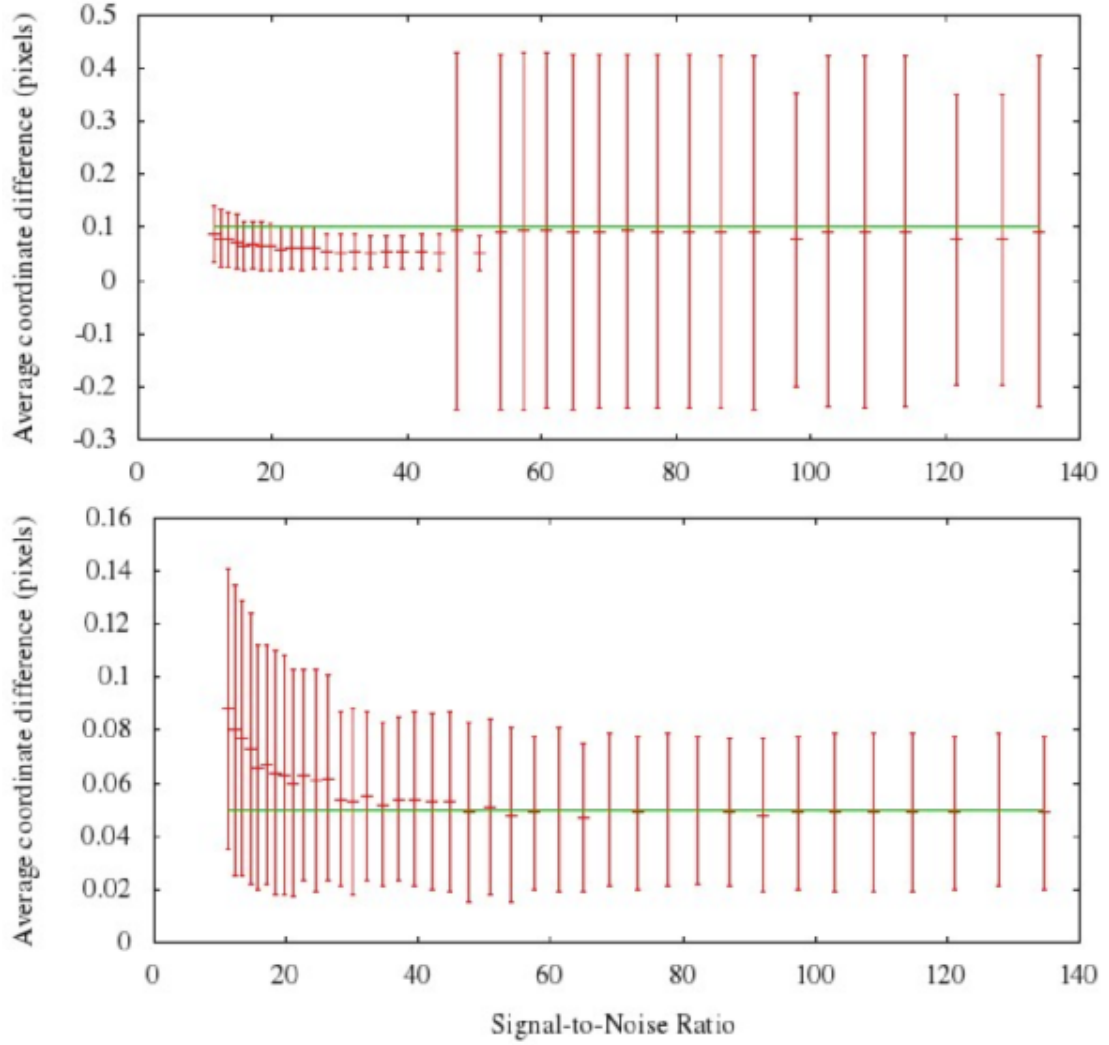
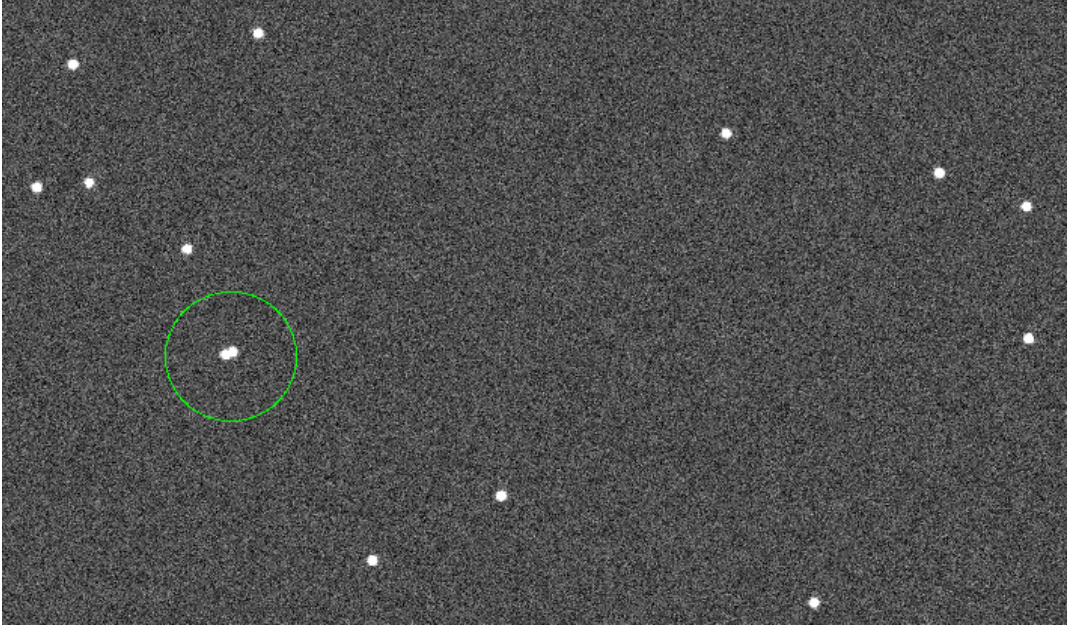


Figure 5: Determined stellar position accuracy, depending on signal-to-noise ratio of stellar sources. Error bars show one sigma standard deviation of the mean position differences. The upper panel shows the case of a pair of overlapping stellar sources, while the lower panel shows results with isolated stars.

The graph shows that the average accuracy of the algorithm was 0.1 (indicated by the green line) of a pixel, with more accuracy obtained from images with a smaller SNR. However, the sudden increase in the standard deviation slightly before the SNR value of 50 is not logical, rather a sequential change similar to the beginning of the graph would be more in order. This prompted the author to find where the increase in inaccuracy came from. Figure 6 shows the phenomenon that caused this.



*Figure 6: Overlapping stars that caused problems for centroid detecting algorithm. Separation between their stellar centers is 7 pixels.*

When randomly generating the positions of the stars used in the images, two stars were placed so close together, that they overlap, such that the algorithm interprets the two stars as one. This happened on stars with magnitudes of  $1^m.8$  and brighter. The difference in the coordinates calculated for the overlapped stars in this case was approximately 2 pixels. On removal of the two merged pixels, the second graph in Figure 5 (lower panel) was obtained.

The result is more logically acceptable, where with a rise in SNR leads to a better accuracy, up to a point. The given point in this case is close to 0.05 pixels, with a standard deviation of 0.03 pixels.

In the case of bright stars, this problem can be considered to be very rare. In this particular scenario, the overlapping stars were at a distance of approximately 7 arcminutes. According to the Yale Bright Star Catalog[13] there are 24 stars brighter than 4.5 magnitude that have a neighbour closer than 10 arcminutes (see Figure 7). Of these, only 11 stars would be detected over the threshold, and of these, only two are of magnitude  $2^m.0$  and brighter.



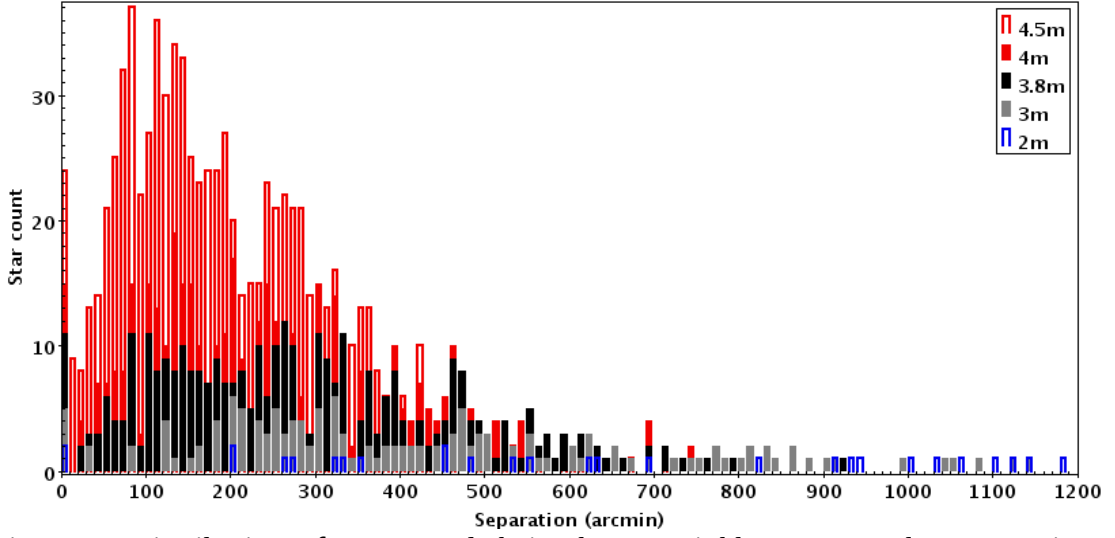


Figure 7: Distribution of stars and their closest neighbours' angular separations. Separations are given in arc minutes. Colours are used to code different stellar magnitudes, as given in the legend.

Should this rarity still be a problem, however, there are a few possible solutions, however none of them are addressed in the algorithm, and the development of such solutions is outside the scope of this thesis. As such, the development of this should be considered as a follow-up in two parts. The first of which is determine the probability that any given blob is made up of two stars. If a blob were made up of two stars, it would be plausible to consider an oval as a model of the blob. Thus, blobs more oval in shape are more likely to be made of two stars than round blobs. Depending on the distance between the stars in the image as well as their brightness, it is also possible to determine the probability by comparing the size of the blob, in pixels, and or brightness with a set threshold. If the probability is high enough, it would be necessary to develop an algorithm which calculated the possible coordinates of the two stars, based on the brightness of the pixels within the star.

The implementation of this algorithm successfully calculated the centroids of stars with an accuracy of 0.1 in the tests carried out, with an average accuracy of 0.05 pixels. However, it is important to note what happened to stars fainter than  $3^m.8$ . In all cases, the stars were not discernible from the background. As such, only singular pixels exceeded the threshold, with many of them actually part of the background rather than part of a star. Should this be an issue, the problem may be solved by either increasing the exposure time on the image sensor, using optics with a larger aperture or selecting a sensor

with a higher quantum efficiency. A combination of these could also be used. Alternatively, multiple images could be sampled and summed. Over the course of multiple images of the same area in the celestial sphere, the random noise will cancel out leaving a smoother background, making fainter stars easier to detect, by increasing the signal-to-noise ratio of the image. In the case of a rotating spacecraft, this solution will not work.

### **6.3. Synthesis for FPGA choice**

In order to determine the feasibility of using an FPGA for this application, the algorithm was rewritten in VHDL and then synthesized. As synthesis requires an FPGA to use as a model, the Xilinx XC7A35T[25] was used.

The two parts of the algorithm were synthesized separately. Synthesis of the threshold system (Appendix 4, Appendix 5 and Appendix 6) revealed that it would run with a minimal (approximately 1% of the total amount of CLBs in the FPGA) amount of resources. A simulation of this (source code in Appendix 8) indicated that the function would work correctly.

The blob detection system (Appendix 5 and Appendix 7) could not be synthesized, as it required too many resources. The main issue was in the last process, “blob\_data”, where the system was to send the centroids and intensities of each blob to the next phase. The system described was too large to synthesize using computer hardware available to the author. Possible solutions to this problem include the usage of larger computer systems or alternatively breaking up the hardware description to more manageable chunks.

## 7. Conclusion

The purpose of this thesis was to improve the Star Tracker algorithm developed at KTH, in the year 2014[2], and to determine which FPGA would be needed to do so.

The thresholding technique described in this thesis offers a more robust way of differentiating stars from the background. This is done using a running average and standard deviation needs to be calculated as each pixel is streamed in from the sensor. This is necessary because holding the entire image in memory while would require far too many resources. For this purpose, the author developed a system to calculate a new average and standard deviation for each pixel.

After thresholding, it is necessary to determine position and the brightness of each star detected. However, as the image is streamed in, the algorithm needs to be able to compile star data over multiple rows, bearing in mind that the rows of a star are divided by streams of pixels from the background. The author used a pixel map for each blob to determine which pixels should be included in specific blobs.

The algorithm developed succeeded in calculating the centroids of stellar objects with an increased accuracy (up to one twentieth of a pixel) and is capable of adapting to changing brightness levels in images. The author was, however, unable to develop a working hardware description for the algorithm. The hardware description developed required too many resources to be implemented. The key problem in the description was brought out and possible solutions suggested.

## 8. Kokkuvõte

Selle lõputöö “Tähetede tuvastamise algoritm ESTCube-2 sateliidi tähetuvastuskaam-  
erale” eesmärk oli edasi arendada tähetuvastussüsteemi, mida oli arendatud KTH Kun-  
inglikus Tehnoloogiainstituudis aastal 2014[2], ja hinnata, milliste parameetritega  
FPGA-d oleks otstarbekas kasutada ESTCube-2 missiooni puhul.

Käesolevas töös kirjeldatud taevafooni tuvastamise meetod pakub robustsema tähtede  
taustast eraldamise algoritmi. Selleks arvutatakse jooksvalt pikslite keskväärtust ja stan-  
dardhälvet. Tähtede tuvastamist pikslite voost kasutatakse seetõttu, et FPGA-s puudub  
piisav mälumaht, hoidmaks tervet kaamera pilti mälus ning ka arvutusi tuleb teha  
jooksvalt andmete sensorist välja lugemise käigus. Selle töö raames arendati algoritm,  
mis arvutab FPGA-s jooksvalt pikslite keskväärtust ja standardhälvet.

Peale tähtede tuvastamist tuleb määrata nende positsioonid ja heledused. Kuna pilti  
loetakse kaamerast välja rea kaupa, on ka iga tähe pikslid mitmetel piksliridadel.  
Käesolevas töös kasutatakse tähekujutise juurde kuuluvate pikslite kirjeldamiseks pik-  
slikaarti.

Valminud algoritm suudab, võrreldes varasemas töös [2] saavutatud ühepikslilise täp-  
susega, määrata tähtede asukohti sensoril suurema täpsusega. Käesolevas töös saavutati  
kõrge signaal-müra suhtega tähekujutiste asukoha määramise täpsuseks kuni üks ka-  
heksmnendik pikslit. Algoritm on suuteline töötama ka muutuva fooni nivoo ja müra  
taseme korral. Algoritmid kirjeldati (vt. Lisa 4 kuni 7) ka FPGA-de puhul kasutatavas  
VHDL-keeles. Probleemikohad VHDL koodis on antud töös välja toodud ning võima-  
likud lahendused ka pakutud.

## **9. Acknowledgements**

The author would like to thank his supervisors, Tõnis Eenmäe and Margus Rosin, as well as close friends and significant others, for their useful counsel and patience.

;)

## 10. References

- 1: Dr. Carl Christian Liebe, Star Trackers for Attitude Determination , 1995,  
<http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=387971>
- 2: Marcus Lindh, Development and Implementation of Star Tracker Electronics, 2014,  
<http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A737909&dswid=-7747>
- 3: Carl Christian Liebe, Accuracy performance of star trackers - a tutorial, 2002,  
<http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=1008988#article>
- 4: Natalie Clark, Intelligent star tracker, 2001,  
<http://proceedings.spiedigitallibrary.org/proceeding.aspx?articleid=903505>
- 5: Allan Read Eisenman, Carl Cristian Liebe, John L. Joergensen, The new generation of autonomous star trackers, 1997,  
<http://proceedings.spiedigitallibrary.org/proceeding.aspx?articleid=932611>
- 6: Attitude | Definition of Attitude by Merriam-Webster, Date: ; <http://www.merriam-webster.com/dictionary/attitude>
- 7: Steven A. Macintyre, Magnetic Field Measurement, 2000,  
<http://engineering.dartmouth.edu/dartmag/docs/macintyre.pdf>
- 8: Thomas Bak, Spacecraft Attitude Determination, 1999,  
<http://www.forskningsdatabasen.dk/en/catalog/2186106498>
- 9: Aaron Burg, Azeem Meruani, Bob Sandheinrich, Michael Wickmann, MEMS Gyroscopes and their Applications, ,  
<http://clifton.mech.northwestern.edu/~me381/project/done/Gyroscope.pdf>
- 10: Pierre Marchal, Field-Programmable Gate Arrays, 1999,  
<http://dl.acm.org/citation.cfm?id=299594>
- 11: J. Rose, A. El Gamal, A. Sangiovanni-Vincentelli, Architecture of field-programmable gate arrays, 1993, [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=231340](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=231340)
- 12: Bob Zeidman, Designing with FPGAs & CPLDs, 2002978-1-57820-112-9
- 13: Yale Bright Star Catalog, Date: 30/05/2016; <http://tdc-www.harvard.edu/catalogs/bsc5.html>
- 14: Datasheet: 1/2-Inch Megapixel CMOS Digital Image Sensor, Date: 30/05/2016;

[http://www.onsemi.com/pub\\_link/Collateral/MT9M001-D.PDF](http://www.onsemi.com/pub_link/Collateral/MT9M001-D.PDF)

15: Datasheet: 1/2.5-Inch 5 Mp CMOS Digital Image Sensor, Date: 30/05/2016;  
[http://www.onsemi.com/pub\\_link/Collateral/MT9P031-D.PDF](http://www.onsemi.com/pub_link/Collateral/MT9P031-D.PDF)

16: Steve B. Howell, Handbook of CCD Astronomy, 2006978-0-521-61762-8

17: Donald E. Knuth, The Art of Computer Programming; Volume 2: Seminumerical Algorithms, 19970-201-89684-2

18: Richard E. Haskell, Darrin M. Hanna, Digital Design Using Digilent FPGA Boards, 2010978-0-9801337-8-3

19: Donald G. Bailey, Michael J. Klaiber, Efficient Hardware Calculation of Running Statistics, 2013,  
[http://seat.massey.ac.nz/research/centres/SPRG/pdfs/2013\\_IVCNZ\\_214.pdf](http://seat.massey.ac.nz/research/centres/SPRG/pdfs/2013_IVCNZ_214.pdf)

20: Saba Akram, Qurrat al Ann, Newton Raphson Method, 2015,  
<http://www.ijser.org/researchpaper%5CNewton-Raphson-Method.pdf>

21: Square root by abacus algorithm, Date: 30/05/2016;  
<http://web.archive.org/web/20120306040058/http://medialab.freaknet.org/martin/src/sqrt/sqrt.c>

22: IRAF Project Home Page, Date: 30/05/2016; <http://iraf.noao.edu/>

23: PyFITS, Date: 30/05/2016; [http://www.stsci.edu/institute/software\\_hardware/pyfits](http://www.stsci.edu/institute/software_hardware/pyfits)

24: MyHDL, Date: 30/05/2016; <http://myhdl.org/>

25: 7 Series FPGAs Overview, Date: 30/05/2016;  
[http://www.xilinx.com/support/documentation/data\\_sheets/ds180\\_7Series\\_Overview.pdf](http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf)

## Appendix 1: Algorithm Block Diagram

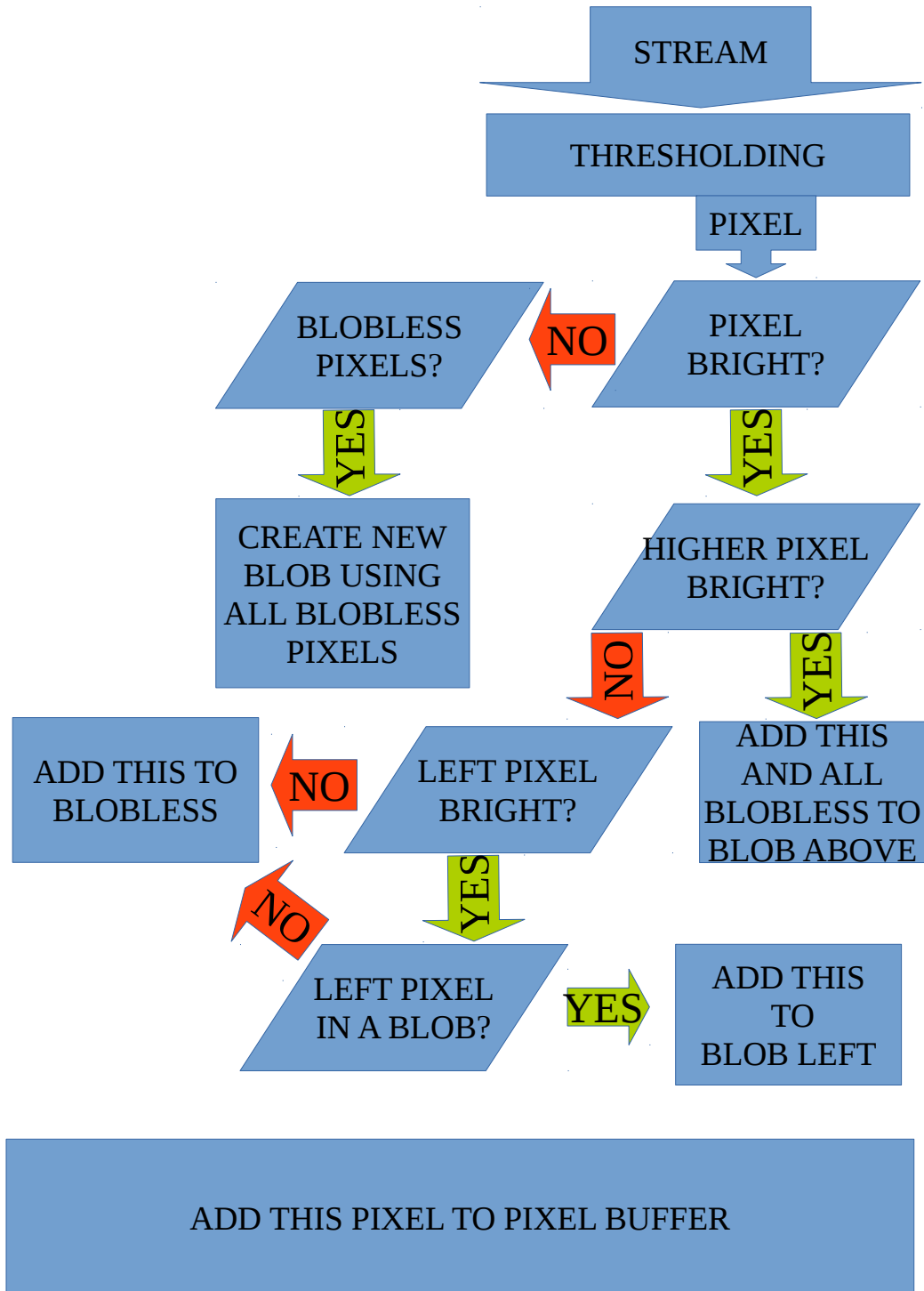


Figure 8: Algorithm Block Diagram



## Appendix 2: Source Code - Newton-Raphson and Binary Method Comparison

```
1  #include "stdio.h"
2  #include "stdlib.h"
3  #include <stdint.h>
4
5  int j = 0;
6
7  int32_t divide(int32_t n, int32_t d)
8  {
9      uint16_t N, D;
10     if (n < 0)
11         N = -n;
12     else
13         N = n;
14
15     if (d < 0)
16         D = -d;
17     else
18         D = d;
19
20     if (!N || !D || N < D)
21         return 0;
22
23     uint16_t Q = 0;
24     uint16_t R = 0;
```

```

25         int8_t i;
26         for (i = 15; i >= 0; i--)
27         {
28             uint16_t x = 1 << i;
29
30
31             Q <<= 1;
32             R <<= 1;
33
34             if (N & x)
35                 R = R + 1;
36
37             if (R >= D)
38             {
39                 R = R - D;
40                 Q = Q + 1;
41             }
42         j++;
43     }
44     if ((n < 0) != (d < 0)) //An XOR of the two is
true
45         return -Q;
46     return Q;
47 }
48
49
50 int newt_sqrt(int num) {

```

```

51     int k, x = 1;
52     while(1)
53     {
54         k = (x + divide(num, x)) >> 1;
55         if(x == k || x - k == 1 || k - x == 1)
56         {
57             return k;
58         }
59         x = k;
60         j++;
61     }
62 }
63
64 long isqrt(long num) {
65     long res = 0;
66     long bit = 1 << 14; // The second-to-top bit is set:
1 << 30 for 32 bits
67     // "bit" starts at the highest power of four <= the
argument.
68     while (bit > num)
69     {
70         bit >>= 2;
71         j++;
72     }
73
74     while (bit != 0) {
75         if (num >= res + bit) {

```

```

76         num -= res + bit;
77         res = (res >> 1) + bit;
78     }
79     else
80         res >>= 1;
81     bit >>= 2;
82     j++;
83 }
84 return res;
85 }
86
87
88 int main(int argc, char* argv[])
89 {
90     printf("%s, ", argv[1]);
91     printf("%d, ", newt_sqrt(atoi(argv[1])));
92     printf("%d, ", j);
93     j = 0;
94     printf("%lu, ", isqrt(atoi(argv[1])));
95     printf("%d\n", j);
96
97 }

```

## Appendix 3: Source Code - Star Tracker in Python

```
1  import pyfits
2  from sys import argv
3  from math import sqrt
4  from myhdl import intbv
5
6  class Blob:
7      def __init__(self, X, Y):
8          self.size = 0
9          self.sum = 0
10         self.x_sum = 0
11         self.y_sum = 0
12         self.parts = [intbv(0, min = 0, max=2**Y) for
j in range(X)]
13
14     def add_to_blob(self, x, y, pix):
15         if pix:
16             self.size += 1
17             self.sum += pix
18             self.x_sum += x * pix
19             self.y_sum += y * pix
20             self.parts[x][y] = True
21
22     def find_in_blob(self, x, y):
23         try:
```

```

24         return self.parts[x][y]
25     except:
26         return False
27     def add_sigma(self, sigma):
28         self.sigma=sigma
29
30     def get(self):
31         #if self.size > 4:
32             return (round(float(self.x_sum)/self.sum, 3),
round(float(self.y_sum)/self.sum, 3), self.sum, self.size)
33
34     M = 0
35     oM = 0
36     S = 0
37     k = 0
38     sigma = 0
39
40
41     def thresh(x, cutoff):
42         global M
43         global oM
44         global S
45         global k
46         global sigma
47
48         k += 1
49         oM = M

```

```

50     M = M - int((M - x) / k)
51     if M < 0:
52         M = 0
53     S += (x - oM) * (x - M)
54
55     if S > 0 and k > 1:
56         sigma = int(sqrt(S/(k - 1)))
57     else:
58         sigma = 0
59
60     if abs(x - M) < cutoff * sigma:
61         x = 0
62     else:
63         x = x - M
64     return x
65
66 print(argv[1])
67
68 img = 0
69
70 try:
71     img = pyfits.getdata(argv[1], ignore_missing_end =
True)
72
73 except:
74     exit(-1)
75

```

```

76
77 pixel_buffer = [None for i in range(len(img[0]))]
78
79 blobs = []
80
81 GLOBAL_X = 0
82 GLOBAL_Y = 0
83
84 Last_active_blob = None
85 size = 0
86
87 for rida in img:
88     for pixel in rida:
89         pix = int(thresh(pixel, 5))
90         if pix:
91             if not Last_active_blob:
92                 for blob in blobs:
93                     if blob.find_in_blob(GLOBAL_X,
GLOBAL_Y-1):
94                         Last_active_blob = blob
95                         break
96             if Last_active_blob:
97                 for i in range(size):
98
Last_active_blob.add_to_blob(GLOBAL_X - (i + 1), GLOBAL_Y,
pixel_buffer[GLOBAL_X - (i + 1)])
99                 size = 0
100                 Last_active_blob.add_to_blob(GLOBAL_X,

```



```

GLOBAL_Y, pix)
101
102             else: # We didn't find a blob this could
have been a part of.
103                 size += 1
104             else: # The pixel in question DID NOT pass
thresholding
105                 if size: # This pixel is preceeded by a
light pixel. We should see if we can add it to a blob.
106                     if not Last_active_blob:
107                         Last_active_blob =
Blob(len(img[0]), len(img))
108                         blobs.append(Last_active_blob)
109                         for i in range(size):
110
111                             Last_active_blob.add_to_blob(GLOBAL_X - (i + 1), GLOBAL_Y,
pixel_buffer[GLOBAL_X - (i + 1)])
112                             Last_active_blob.add_sigma(sigma)
113                             size = 0
114                     else:
115                         Last_active_blob = None
116
117                         pixel_buffer[GLOBAL_X] = pix
118                         GLOBAL_X += 1
119                         GLOBAL_Y += 1
120                         GLOBAL_X = 0
121
122 out_file = open(argv[1]+ ".csv", "w")

```

```
122 out_file.write(argv[1]+ "\n")
123 for i in blobs:
124     if i.get():
125         txt = "{0[0]:.3f}, {0[1]:.3f}, {0[2]},
126         {0[3]}\n".format(i.get())
127         out_file.write(txt)
```

## Appendix 4: Source Code – arith.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3  use IEEE.NUMERIC_STD.all;
4
5  package arith is
6      function bin_sqrt ( num : in UNSIGNED) return UNSIGNED;
7      function divide ( NUMERATOR,DENOMINATOR : in UNSIGNED) return UN-
SIGNED;
8  end arith;
9
10 package body arith is
11     function bin_sqrt (num : in UNSIGNED ) return UNSIGNED is
12         variable int_num : UNSIGNED (num'Length-1 downto 0);
13         variable res : UNSIGNED (num'Length-1 downto 0) := (others =>
'0');
14         variable bitt : UNSIGNED (num'Length-1 downto 0) := (0 => '1',
others => '0');
15     begin
16         bitt := shift_left(bitt, bitt'Length-2);
17         int_num := num;
18         if (num < 0) then
19             int_num := not int_num;
20             int_num := int_num - 1;
21         end if;
22         L1: for i in 0 to 31 loop
23             exit L1 when bitt <= num;
24             bitt := shift_right(bitt, 2);
25         end loop L1;
26
27         L2: for i in 0 to num'Length-1 loop
28             exit L2 when bitt = 0;
29             if (int_num >= (res + bitt)) then
```

```

30             int_num := int_num - (res + bitt);
31             res := shift_right(res, 1);
32             res := res + bitt;
33         else
34             res := shift_right(res, 1);
35         end if;
36
37         bitt := shift_right(bitt, 2);
38     end loop L2;
39     return res;
40 end bin_sqrt;
41
42 function divide ( NUMERATOR,DENOMINATOR : in UNSIGNED) return UNSIGNED
is
43     variable N : UNSIGNED (NUMERATOR'Length-1 downto 0) := (others =>
'0');
44     variable D : UNSIGNED (DENOMINATOR'Length-1 downto 0) := (others
=> '0');
45     variable Q : UNSIGNED (NUMERATOR'Length-1 downto 0) := (others =>
'0');
46     variable R : UNSIGNED (DENOMINATOR'Length-1 downto 0) := (others
=> '0');
47     begin
48         N := NUMERATOR;
49         D := DENOMINATOR;
50         if (N = 0 or D = 0 or N < D) then
51             return Q;
52         end if;
53         L: for i in 15 downto 0 loop
54             Q := shift_left(Q, 1);
55             R := shift_left(R, 1);
56             R(0) := N(i);
57             if (R >= D) then
58                 R := R - D;
59                 Q := Q + 1;

```

```
60             end if;
61         end loop L;
62         if (NUMERATOR < 0 xor DENOMINATOR < 0) then
63             Q := not Q;
64             Q := Q + 1;
65         end if;
66         return Q;
67     end divide;
68 end arith;
```

## Appendix 5: Source Code – sensor.vhd – VHD Package

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3  use IEEE.NUMERIC_STD.all;
4
5  package Sensor is
6      constant LENGTH_X : INTEGER := 2592;
7      constant LENGTH_Y : INTEGER := 1944;
8      constant N_X : INTEGER := 11; -- This is the number of bits in the num-
ber of pixels in the sensor: log(LENGTH_X)/(log(2)
9      constant N_Y : INTEGER := 10; -- This is the number of bits in the num-
ber of pixels in the sensor: log(LENGTH_Y)/(log(2)
10     constant N : INTEGER := 22; -- This is the number of bits in the number
of pixels in the sensor: log(LENGTH_X*LENGTH_Y)/(log(2)
11     constant MAX_BLOBS : INTEGER := 10;
12     type bit_map is array (0 to 9) of STD_LOGIC_VECTOR (19 downto 0);
13     type blob is record
14         first_x : UNSIGNED (n_x downto 0);
15         first_y : UNSIGNED (n_y downto 0);
16         b_map : bit_map;
17         x_sum : UNSIGNED (31 downto 0);
18         y_sum : UNSIGNED (31 downto 0);
19         sum : UNSIGNED (31 downto 0);
20         n : INTEGER range 0 to 127;
21     end record;
22     type blob_array is array (0 to MAX_BLOBS-1) of blob;
23     type pixel_array is array (0 to LENGTH_X-1) of UNSIGNED (15 downto 0);
24 end Sensor;
25 package body Sensor is
26 end Sensor;
```

## Appendix 6: Source Code – thresholding.vhd – VHD Source

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  use WORK.ARITH.ALL;
5  use WORK.SENSOR.ALL;
6
7  entity thresholding is
8      Port ( pixel_in : in UNSIGNED (15 downto 0);
9            pixel_out : out UNSIGNED (15 downto 0);
10           new_pixel : STD_LOGIC;
11           clk : in STD_LOGIC);
12  end thresholding;
13
14  architecture Behavioral of thresholding is
15      signal M : UNSIGNED (15 downto 0) := (others => '0'); -- the current
arithmetic mean
16      signal oM : UNSIGNED (15 downto 0) := (others => '0'); -- the old
arithmetic mean
17      signal k : UNSIGNED (N downto 0) := (others => '0'); -- the total num-
ber of pixels obtained
18      signal sigma : UNSIGNED (16 downto 0) := (others => '0');
19  begin
20      averaging : process(pixel_in)
21          -- this process calculates a new average with every change in
"pixel_in"
22          variable dif : UNSIGNED (15 downto 0) := (others => '0');
23      begin
24          k <= k + 1;
25          if M > pixel_in then
26              dif := M - pixel_in;
27              dif := divide(dif, k);
```

```

28         M <= M + dif;
29     else
30         dif := pixel_in - M;
31         dif := divide(dif, k);
32         M <= M - dif;
33     end if;
34 end process;
35
36 std_dev : process (M)
37 --this process calculates the standard deviation with every new aver-
age
38     variable dif : UNSIGNED (15 downto 0) := (others => '0');
39     variable odif : UNSIGNED (15 downto 0) := (others => '0');
40     variable active_pixel : UNSIGNED (15 downto 0) := (others => '0');
41     variable S : UNSIGNED (32 downto 0) := (others => '0');
42     variable n_S : UNSIGNED (32 downto 0) := (others => '0');
43 begin
44     active_pixel := pixel_in;
45     if M > active_pixel then
46         dif := M - active_pixel;
47     else
48         dif := active_pixel - M;
49     end if;
50     if oM > active_pixel then
51         odif := oM - active_pixel;
52     else
53         odif := active_pixel - oM;
54     end if;
55     S := S + divide (S + dif*odif, k-1);
56     sigma <= bin_sqrt(S)(16 downto 0);
57 end process;
58
59 threshy : process(sigma)

```



```

60      --this process calculates the new value of the pixel, based on the
threshold.
61      variable f_sigma : UNSIGNED (16 downto 0) := (others => '0');
62      begin
63          f_sigma := shift_left(sigma, 2) + sigma; -- multiply sigma by 5:
shift left twice, then add sigma.
64          if pixel_in < M + f_sigma then
65              pixel_out <= (others => '0');
66          else
67              pixel_out <= pixel_in - (M + f_sigma(15 downto 0));
68          end if;
69      end process;
70  end Behavioral;

```

## Appendix 7: Source Code – blobber.vhd – VHD Source

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4 use WORK.SENSOR.ALL;
5
6 entity blobber is
7     Port ( pixel_in : in UNSIGNED (15 downto 0);
8           clk : in STD_LOGIC;
9           blob_x : out UNSIGNED (31 downto 0);
10          blob_y : out UNSIGNED (31 downto 0);
11          blob_int : out UNSIGNED (31 downto 0));
12 end blobber;
13
14 architecture Behavioral of blobber is
15     component thresholding is
16         Port ( pixel_in : in UNSIGNED (15 downto 0);
17               pixel_out : out UNSIGNED (15 downto 0);
18               new_pixel : STD_LOGIC;
19               clk : in STD_LOGIC);
20     end component;
21     signal pixel_threshed : UNSIGNED (15 downto 0);
22     signal new_pixel : STD_LOGIC;
23     signal x_coord : integer range 0 to LENGTH_X - 1 := 0;
24     signal y_coord : integer range 0 to LENGTH_Y - 1 := 0;
25     signal b_array_top : integer range 0 to MAX_BLOBS - 1 := 0;
26     signal active_blob : integer range -1 to MAX_BLOBS - 1 := -1;
27     signal b_array : blob_array;
28     signal p_array : pixel_array;
29     signal blob_search_complete : STD_LOGIC := '0';
30     signal the_blobless : INTEGER range 0 to 20 := 0; -- We assume that
```

we'll not have stars larger than 20 pixels.

```
31 begin
32     thresh : thresholding PORT MAP(
33         pixel_in => pixel_in,
34         pixel_out => pixel_threshed,
35         new_pixel => new_pixel,
36         clk => clk);
37     coords : process (new_pixel)
38         -- This process keeps track of the current pixel coordinates.
39     begin
40         if x_coord = LENGTH_X - 1 then
41             x_coord <= 0;
42             if y_coord = LENGTH_Y - 1 then
43                 y_coord <= 0;
44             else
45                 y_coord <= y_coord + 1;
46             end if;
47         else
48             x_coord <= x_coord + 1;
49         end if;
50         p_array(x_coord) <= pixel_threshed;
51     end process;
52
53     search_for_blobs : process(x_coord)
54         -- This process searches the left and upper pixels to see if we have
55         any blobs.
56         variable x_crd : integer range 0 to LENGTH_X - 1 := 0;
57         variable y_crd : integer range 0 to LENGTH_Y - 1 := 0;
58     begin
59         blob_search_complete <= '0';
60         if b_array_top > 0 then
61             BLOB_SEARCH : for i in 0 to MAX_BLOBS - 1 loop
62                 if i >= b_array_top - 1 then
```

```

62             exit BLOB_SEARCH;
63         end if;
64         -- First, we need to map our global x and y coordinates
to the local bitmap of the blob.
65         if x_coord > b_array(i).first_x then
66             x_crd := 10 + x_coord -
to_integer(b_array(i).first_x);
67         else
68             x_crd := 10 - (to_integer(b_array(i).first_x) - x_co-
ord);
69         end if;
70         y_crd := y_coord - to_integer(b_array(i).first_y);
71
72         if b_array(i).b_map(y_crd)(x_crd - 1) = '1' or
b_array(i).b_map(y_crd - 1)(x_crd) = '1' then
73             active_blob <= i;
74             blob_search_complete <= '1';
75             exit BLOB_SEARCH;
76         end if;
77     end loop BLOB_SEARCH;
78 end if;
79
80     if blob_search_complete = '0' then
81         blob_search_complete <= '1';
82         active_blob <= -1;
83     end if;
84 end process;
85
86 add_to_blobs : process (blob_search_complete)
87     variable x_crd : integer range 0 to LENGTH_X - 1 := 0;
88     variable y_crd : integer range 0 to LENGTH_Y - 1 := 0;
89 begin
90     if rising_edge(blob_search_complete) then
91         if not pixel_threshed = X"0000" then

```

```

92             the_blobless <= the_blobless + 1;
93             if active_blob >= 0 then
94                 -- This pixel passed the threshold and we have a blob
to add it to.
95                 -- First, we need to map our global x and y coordi-
nates to the local bitmap of the blob.
96                 if x_coord > b_array(active_blob).first_x then
97                     x_crd := 10 + x_coord - to_integer(b_array(ac-
tive_blob).first_x);
98                 else
99                     x_crd := 10 -
(to_integer(b_array(active_blob).first_x) - x_coord);
100                end if;
101                y_crd := y_coord -
to_integer(b_array(active_blob).first_y);
102
103                ADD_BLOBLESS : for i in 0 to 20 loop
104                    if i >= the_blobless - 1 then
105                        exit ADD_BLOBLESS;
106                    end if;
107                    b_array(active_blob).b_map(y_crd)(x_crd - i) <=
'1';
108                    b_array(active_blob).x_sum <=
b_array(active_blob).x_sum + (p_array(x_crd - i) * (x_crd - i));
109                    b_array(active_blob).y_sum <=
b_array(active_blob).y_sum + (p_array(x_crd - i) * y_crd);
110                    b_array(active_blob).sum <=
b_array(active_blob).sum + p_array(x_crd - i);
111                    b_array(active_blob).n <= b_array(active_blob).n
+ 1;
112                end loop ADD_BLOBLESS;
113                the_blobless <= 0;
114            end if;
115        else
116            -- The current pixel did not pass the threshold.
117            if the_blobless > 0 then

```

```

118             -- We have blobless, so we should create a new blob.
119             CREATE_BLOB : for i in 0 to 20 loop
120                 if i >= the_blobless - 1 then
121                     exit CREATE_BLOB;
122                 end if;
123                 b_array(b_array_top).b_map(y_crd)(x_crd - i) <=
'1';
124                 b_array(b_array_top).x_sum <=
b_array(b_array_top).x_sum + (p_array(x_crd - i) * (x_crd - i));
125                 b_array(b_array_top).y_sum <=
b_array(b_array_top).y_sum + (p_array(x_crd - i) * y_crd);
126                 b_array(b_array_top).sum <=
b_array(b_array_top).sum + p_array(x_crd - i);
127                 b_array(b_array_top).n <= b_array(b_array_top).n
+ 1;
128             end loop CREATE_BLOB;
129             the_blobless <= 0;
130             b_array_top <= b_array_top + 1;
131         end if;
132     end if;
133 end if;
134 end process;
135
136 blob_data : process (y_coord)
137 -- This process will send blob data as soon as a blob is compiled.
138     variable sent_blob : integer range 0 to MAX_BLOBS - 1 := 0;
139     variable sum_x, sum_y : UNSIGNED (31 downto 0);
140     begin
141         if b_array_top > 0 then
142             parse_blobs : while sent_blob < b_array_top loop
143                 if y_coord - to_integer(b_array(sent_blob).first_y) >= 10
or y_coord = LENGTH_Y - 1 then
144                     sum_x := b_array(sent_blob).x_sum * 10;
145                     sum_y := b_array(sent_blob).y_sum * 10;
146                     blob_x <= sum_x / b_array(sent_blob).sum;

```

```

147             blob_y <= sum_y / b_array(sent_blob).sum;
148             blob_int <= b_array(sent_blob).sum /
b_array(sent_blob).n;
149             sent_blob := sent_blob + 1;
150         else
151             exit parse_blobs;
152         end if;
153     end loop parse_blobs;
154 end if;
155 end process;
156
157 end Behavioral;

```

## Appendix 8: Source Code – thresh\_sim.vhd – VHDL Simulation

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.ALL;
4
5  ENTITY Thresh_sim IS
6  END Thresh_sim;
7
8  ARCHITECTURE behavior OF Thresh_sim IS
9      COMPONENT thresholding
10     PORT(
11         pixel_in : IN  UNSIGNED (15 downto 0);
12         pixel_out : OUT  UNSIGNED(15 downto 0);
13         new_pixel : IN  std_logic;
14         clk : IN  std_logic
15     );
16     END COMPONENT;
17     signal pixel_in : UNSIGNED(15 downto 0) := (others =>
'0');
18     signal new_pixel : std_logic := '0';
19     signal clk : std_logic := '0';
20     signal pixel_out : UNSIGNED(15 downto 0);
21     constant clk_period : time := 10 ns;
22 BEGIN
23     uut: thresholding PORT MAP (
```



```

24         pixel_in => pixel_in,
25         pixel_out => pixel_out,
26         new_pixel => new_pixel,
27         clk => clk
28     clk_process :process
29     begin
30         clk <= '0';
31         wait for clk_period/2;
32         clk <= '1';
33         wait for clk_period/2;
34     end process;
35     stim_proc: process
36     begin
37         wait for 100 ns;
38         wait for clk_period*10;
39         pixel_in <= X"2010";
40         wait for clk_period;
41         pixel_in <= X"1010";
42         wait for clk_period;
43         pixel_in <= X"8010";
44         wait for clk_period;
45         pixel_in <= X"6010";
46         wait for clk_period;
47         pixel_in <= X"F010";
48         wait for clk_period;
49         pixel_in <= X"3010";
50         wait for clk_period;

```

```
51      pixel_in <= X"2010";
52      wait for clk_period;
53      pixel_in <= X"A010";
54      wait for clk_period;
55      pixel_in <= X"B010";
56      wait for clk_period;
57      wait;
58  end process;
59 END;
```

# **Non-exclusive license to reproduce and make this thesis public**

**I, ANDREAS RAGEN AYAL,**

1. herewith grant the University of Tartu a free permit (non-exclusive license) to:
  - 1.1. reproduce, for the purpose of preservation and making available to the public, including addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
  - 1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

**STAR DETECTION ALGORITHM FOR ESTCUBE-2 STAR TRACKER,**

supervised by **MARGUS ROSIN, MSc AND TÕNIS EENMÄE, Msc,**

2. I am aware of the fact that the author retains these rights.
3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 01.06.2016