LOST: An Open-Source Suite of Star Tracking Software

Mark Polyakov, Edward Zhang, Karen Haining
University of Washington
Guggenheim 211 University of Washington, Seattle WA 98195
xe@uw.edu

Faculty Advisor: Alvar Saenz-Otero
University of Washington
alvarso@uw.edu

ABSTRACT

We present LOST: Open-source Star Tracker (LOST), a suite of star tracking software particularly suitable for small satellite missions with limited computing resources and low-cost cameras. LOST contains implementations of a number of previously-proposed star tracking algorithms and a flexible framework for running and evaluating these algorithms. Our evaluation finds that LOST's algorithms are simultaneously able to maintain a strong combination of accuracy, runtime, and memory usage. In scenarios representative of a low-cost star tracker, LOST correctly identifies over 95% of images, and importantly, performs the entire star tracking pipeline in less than 35 milliseconds on a Raspberry Pi while using less than 1 MiB of memory, backed by a < 350 KiB database. These results indicate that LOST could be ported to an embedded or radiation-hardened CPU and still perform well enough to meet the accuracy requirements of many missions.

INTRODUCTION

Attitude determination is a crucial requirement for many satellite missions. There are two mainstream methods of attitude determination: the first method is to use a combination of a sun sensor and magnetic sensor, and the second is to use a star tracker, which is a combination of hardware and software that determines the satellite's attitude by identifying stars in a given photograph. Of the two methods, star tracking is generally the most accurate.[5]

However, even when desirable, star trackers are not necessarily practical for every mission. Notably, small satellite missions often have limited computing resources, as well as a limited budget. The limitation on computing resources may be in order to conserve electrical power, or because the satellite uses radiation-hardened components, which typically have only a few megabytes of RAM. We use the term "low-compute" to describe star tracking computers that are built on embedded microprocessors, do not use a traditional operating system, and have around 1 MiB of memory with 100 MHz CPU speed. (For example, the popular Vorago Cortex M4 CPUs have a clock rate of approximately 100 MHz.[9])

While commercial star trackers can handle these limitations, they can be prohibitively expensive,[5] as the most basic of commercial star trackers can cost tens of thousands of dollars. For example, Arcsec's Sagitta star tracker is listed as over $40,000.[1] As small satellites are deployed in increasingly large constellations and deployment becomes cheaper, the cost of star trackers becomes proportionally larger, rendering them unaffordable to many engineering teams looking to launch small satellites. Open-source star trackers, which make their software publicly available, are affordable alternatives to commercially-available ones. However, we find that previous open-source star tracking softwares exhibit good performance only in specific scenarios, or require relatively powerful computing hardware, making them unsuitable for low-compute missions.

To address the need for affordable star tracking software suitable for low-compute scenarios, we present LOST: Open-source Star Tracker. LOST is a free and open-source codebase for star tracking software. This paper first describes LOST, including (1) a high-level overview of the star tracking algorithms implemented for LOST, and (2) an overview of LOST's infrastructure, which includes a testing framework for evaluating algorithms on realistic generated star images and allows for the swapping out of different algorithms for different stages of the star tracking pipeline. Next, this paper evaluates all algorithms on various performance metrics such as speed, accuracy, and memory usage, under a number of conditions. Speed and memory tests are performed on both a desktop computer and a Raspberry

Pi, yielding results that suggest LOST could be run without significant modification on low-compute embedded systems with an accuracy comparable or better than that of other open-source star tracking softwares.

PREVIOUS WORK

Previous star tracking softwares can be classified into two groups: (1) “Plate solving” softwares originally designed for use in astronomy, and (2) star tracking software designed for use on actual satellites. Leading plate solvers such as Astrometry, ASTAP, and Match are excellent for their designed use case, which is identifying images taken through telescopes with high-quality cameras and long exposure times. However, these softwares require long compute times and large amounts of memory, making them unsuitable for star tracking done with cheap, lower-quality cameras. Plate solvers may also fail to identify images when only a small number of stars are visible, as they are primarily designed to identify photos taken with long exposure times, when many stars are visible. Therefore, we only discuss star trackers in this paper.

During our review of previous literature we found only three existing open-source star tracking softwares: OpenStarTracker, Tetra3, and SOST. A number of other publications claim development of open-source star trackers, but in fact do not make their source code available.

OpenStarTracker is the most well established of the three, and is the only one that is scheduled to be flown on a real-world mission (Oresat). However, OpenStarTracker relies on the heavyweight OpenCV computer vision library to perform centroiding, which makes it unsuitable for many embedded platforms. More importantly, its only implemented star identification algorithm is Star-ND. While it is fast and simple to understand, Star-ND is relatively sensitive to errors in observed star brightness as well as missing or false stars.

Tetra3 uses the Tetra algorithm for star identification. Like OpenStarTracker, Tetra3 also uses heavyweight third-party libraries such as numpy and scipy to perform all stages of star tracking. There is also a C implementation of the Tetra star identification algorithm, which we refer to as C-Tetra. While both Tetra3 and C-Tetra require relatively large databases, C-Tetra uses a database an order of magnitude larger at about 5 GiB. Additionally, C-Tetra only implements the star identification step and requires an external program for other stages of star tracking. While the C implementation could likely be adapted for use on a mission, doing so would require substantial effort.

SOST internally uses the Match plate-solving tool for star identification. Like other plate-solvers, Match is much slower and requires more memory than star tracking algorithms designed for use on space missions. The SOST authors report that the star identification step takes about 20 seconds on a Raspberry Pi 3.[6] If slow identification is acceptable, SOST may be suitable as a star tracking software.

We quantitatively compare LOST to C-Tetra in the Results section. We were unable to get OpenStarTracker to identify the set of images we used in our comprehensive test (see Methodology section), although with the right tuning of parameters it might have been possible.

SYSTEM DESIGN

Any star tracking software takes an image with visible stars as input and outputs the attitude of the star tracker’s camera as a 3x3 matrix or quaternion. LOST makes a strong distinction between the actual star tracking algorithms and the “infrastructure” used for evaluating and running the stages of the star tracking pipeline, as illustrated in Figure 1.

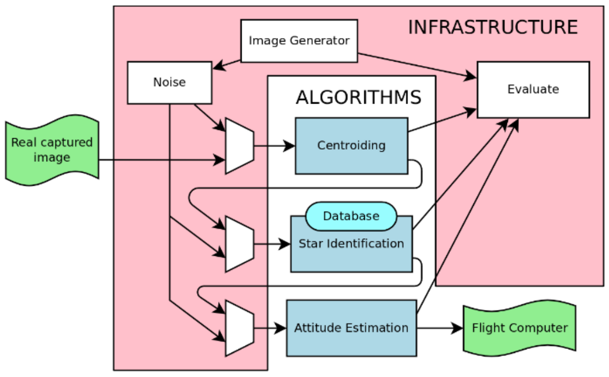

Figure 1: LOST System Diagram

Like most star trackers, LOST’s star tracking pipeline is composed of three main stages. First, in the centroid detection stage, the pixel coordinates of each star center (centroid) are computed to within a fraction of a pixel. Next, in the star identification stage, an algorithm determines the identity of each centroid with the help of a star catalog (e.g. the Yale Bright Star Catalog) and a pre-computed database specific to the star-id algorithm. Finally, in the attitude estimation stage, an algorithm computes the attitude that best agrees with the star centroids and identifications, which in a real mission is then sent to a flight computer.

LOST provides implementations of previously-proposed algorithms for each stage of the star tracking pipeline, each of which are described in the next section (see

Algorithms section). The user can easily swap out which algorithm is used for each stage of the pipeline.

LOST's evaluation infrastructure consists primarily of an image generator, which creates realistic star images based on true catalog star positions and a variety of configurable noise sources. The outputs from the image generator (see Infrastructure section) can be used to run and automatically evaluate the implemented algorithms on various performance metrics.

The source code for LOST is released under the MIT license, which permits free use and modification of the software. The source code is available on GitHub (https://github.com/UWCubeSat/lost/). LOST is implemented as a fully-documented C++ program that can compile and run without modification on the Linux operating system using a command-line interface. No third-party software libraries are required, although the Eigen linear algebra library is needed for certain optional algorithms.

ALGORITHMS

Centroiding

The centroiding stage determines which pixels in a given image belong to stars and locates the pixel coordinates (x_c, y_c) of each star center, called a centroid. Based on a review of literature, 5 centroiding algorithms were chosen and implemented for LOST.

Implementations for all centroiding algorithms resemble the following steps, given an image with mean pixel brightness μ and standard deviation σ:

1. Identify every cluster of pixels with brightness above the threshold $\mu + \alpha\sigma$, where α is tuned empirically ($\alpha = 5$ has been shown to work well[4]). Each cluster represents a possible star. Optimizations to this step ensure that multiple centroids will never be found for a single star.
2. Mitigate the effect of background noise by subtracting μ from each pixel in the image.
3. For each cluster of pixels, compute the pixel coordinates (x_c, y_c) of the centroid according to the specifics of the centroiding algorithm. This step is the most involved and is described below for each of the 5 centroiding methods.

Center of Gravity

The Center of Gravity algorithm (COG) computes the centroid by taking the weighted average of all pixels in a star, as shown in Equation 1.

Let B denote the set of pixels in a star cluster. Each pixel (x, y, w) has coordinates (x, y) with brightness w.

$$(x_c, y_c) = \frac{1}{\sum_{(x,y,w)\in B} w} \cdot \sum_{(x,y,w)\in B} [w \cdot (x, y)] \qquad (1)$$

Iterative Weighted Center of Gravity

An improvement over the Center of Gravity algorithm is the "iterative weighted COG" algorithm, which takes an estimate (x, y) and then computes the weighted average of the image pixel brightness multiplied by a 2D Gaussian function centered at (x, y). The standard deviation of the Gaussian function is computed as described in Delabie[4]. The iterative weighted algorithm performs the weighted center of gravity algorithm iteratively, improving its estimate at each step, until convergence.

1-dimensional and 2-dimensional Gaussian Least-Squares Fit:

The Gaussian least-squares fit algorithms operate under the assumption that the distribution of defocused light on a sensor can be modeled by a Gaussian function. Therefore, for each star cluster, both algorithms attempt to find the parameters of a Gaussian curve that accurately models the star. Parameters are determined by performing a nonlinear least-squares fit on pixel data within a window of specified size.

The 1-dimensional Gaussian least-squares fit considers the X and Y directions separately, reducing the number of parameters in the target Gaussian function. On the other hand, the 2-dimensional Gaussian fit considers all pixels in the window in one go. Since the 2-d Gaussian function (Equation 2) has more parameters (Equation 3) than that of the 1-d function,[4] the 2-d algorithm is slower, with its runtime scaling exponentially with increasing window size. Both methods are among the slowest of centroiding algorithms but are also the most accurate and resistant to image noise.

The equations for the 2-dimensional Gaussian fit are shown below. The 1-d Gaussian fit uses a similar approach.

$$argmin_\beta \sum_{i=-r}^{r} \sum_{j=-r}^{r} \left(A_{ij} - f(x_i, y_j, \beta)\right)^2 \qquad (2)$$

$$f(x_i, y_j, \beta) = a e^{\frac{-(x_i - x_b)^2}{2\sigma_x^2}} \cdot e^{\frac{-(y_j - y_b)^2}{2\sigma_y^2}} \qquad (3)$$

where r = radius (in pixels) of window, (x_0, y_0) = pixel coordinates of the window's center pixel, $\beta = (x_b, y_b, a, \sigma_x, \sigma_y)$ = parameters of the Gaussian curve to solve for. For our purposes, only the estimate of the centroid (x_b, y_b) is of interest.

Gaussian Grid

The Gaussian Grid algorithm, proposed in 2014 by Delabie et al., again aims to fit a Gaussian function to pixel data. Instead of using a nonlinear least-squares fit, Gaussian Grid shortens its runtime by using a set of closed-form expressions to approximate function parameters. LOST contains the first known open-source implementation and evaluation of this algorithm. Note that this method is only defined for a 5x5 window.

Star Identification

Given a list of centroids and a star catalog, the star identification stage attempts to identify each centroided star by matching it to a unique catalog star. Star identification is the most sophisticated part of the star tracking pipeline. We implement two pattern-matching algorithms suitable for lost-in-space mode, as well as a tracking mode algorithm that takes advantage of prior attitude information.

Tetra

Tetra is a fast O(1) star identification algorithm that achieves its low runtime by constructing and looking up 4-star patterns in a special hash table called the pattern catalog. However, this makes Tetra's database significantly larger than that of any other star-id algorithm, owing to the large number of possible star patterns and need for a reasonable hash table load factor.

LOST contains an implementation of Tetra based on Brown and Stubis's original paper.[2] Our implementation is optimized to improve practical performance; for instance, hyperparameter tuning gives a hash function that more uniformly distributes star patterns, which in turn greatly reduces the hash collision rate when performing lookup into the pattern catalog.

Pseudocode:

```
1. Choose the 4 brightest star centroids in the image
2. Each star pattern is constructed by calculating the $C(4,2) = 6$ pairwise inter-star distances, then dividing each value by the length of the longest distance
3. Sort and drop the last element since the longest distance divided by itself is always 1. We now have a sorted list of edge ratios $[e_1, e_2, e_3, e_4, e_5]$
4. A hash code $(a_1, a_2, a_3, a_4, a_5)$ is generated from $[e_1, e_2, e_3, e_4, e_5]$. To account for centroiding error, we consider all hash values such that $|a_i - e_i| < \varepsilon$, where $\varepsilon$ is a tuned constant.
5. Index and look up the hashed pattern in the pattern catalog. If there exists a matching pattern in the pattern catalog, then identify the stars in our 4-star pattern by pairing them to catalog stars. Otherwise, if there is no match, repeat steps 1-4 for some other combination of 4 centroid stars, in order of decreasing star brightness.
```

Pyramid

The Pyramid star identification algorithm is one of the most popular star-ID algorithms and has been used on many real-world missions. The Pyramid algorithm begins by choosing a pattern of 4 centroids in the image, then uses a range-query database to find all possible 4-tuples of catalog stars which match the observed inter-star distances. Mortari et. al· have determined analytically that, under reasonable assumptions of centroiding accuracy, matching 4-star patterns very rarely results in a false match, ensuring Pyramid has a low false positive rate.[8]

To our understanding, LOST contains the first open-source Pyramid implementation. While the algorithm is fairly simple, the original paper is light on implementation details, so LOST's codebase can help clear up the details for future researchers. Our implementation is optimized to improve speed, most importantly by building hash tables to quickly find the last two catalog starts that might match a pattern. Our implementation follows the pseudocode below.

Pseudocode:

```
1. Choose the four stars/centroids in the image with the shortest sum of inter-star distances, call $i, j, k, r$
2. Perform a database range query to find all pairs of catalog stars whose inter-star distance agrees with the $i - k$ distance.
    a. Build a hash table $M_{ik}$ from the returned pairs which maps each catalog star to another catalog star the same distance away as $i$ is from $k$, if it exists.
    b. I.e., $M_{ik}[m] = n$ if and only if the distance between stars $m$ and $n$ is almost the same as the distance from $i$ to $j$
3. Query and build a similar hash table for the $i - r$ distance, named $M_{ir}$
4. Query catalog pairs matching the $i - j$ distance, and for each matching pair $(c_i, c_j)$
    a. Let $c_k \coloneqq M_{ik}(c_i)$ and $c_r \coloneqq M_{ir}(c_i)$
    b. Check that $c_k$ and $c_r$ exist
    c. Check that all six inter-star distances between $i, j, k, r$ agree with the six inter-star distances between $c_i, c_j, c_k, c_r$. Also perform the spectrality check described by Mortari[8]
```

    d. If both checks pass, then (c_i, c_j, c_k, c_r) is a match for (i, j, k, r)
5. If there exists a potential match and it is unique, return it. Else select new (i, j, k, r)

Some details are omitted here. For example, M_{ik} and M_{ir} may have entries with duplicate keys. LOST implements the K-Vector database, described by Mortari,[8] to perform range queries in O(1) time.

Tracking Mode

The star-id algorithms listed above are “lost-in-space mode” algorithms, which assume no prior information about the star tracker’s attitude. “Tracking mode” algorithms, on the other hand, take advantage of a previous attitude estimate to identify images with high confidence even when few stars are visible. The attitude estimate comes from a previous image, which is reasonable assuming there is only a small change in the satellite’s attitude from the previous image to the current image.

LOST implements a simple tracking mode algorithm that begins by computing a list of candidate stars for each detected centroid. In the case of small rotations, there may be only one candidate star, so no further computation is necessary. Otherwise, tracking mode finds a unique combination of the candidate stars that agrees with the observed inter-star distances.

Attitude

LOST implements three attitude estimation algorithms: QUEST, the Davenport Q method, and TRIAD. Each of these algorithms tries to find a rotation that brings the star vectors in the reference frame as close as possible to the observed star vectors by minimizing Wahba’s loss function[3]. The TRIAD algorithm solves this analytically, using two pairs of measured and reference vectors alongside an intermediate reference frame, to determine attitude. TRIAD is usually not the best choice for star trackers because it only uses information from two stars instead of every identified star, so a large amount of useful information is discarded. Meanwhile, the QUEST and Davenport Q methods find an optimal solution to Wahba’s problem quickly.[5] Because these attitude estimation algorithms are well-documented and well-tested, we do not evaluate them in this paper.

INFRASTRUCTURE

LOST’s flexible infrastructure can be valuable for both engineers and researchers. As shown in the system diagram (see System Design section, Figure 1), LOST treats each stage of the star tracking pipeline individually, with each stage having a well-defined interface. Consequently, it is easy for a researcher to test the performance of a new star tracking algorithm quickly by swapping their algorithm into the pipeline, while letting the infrastructure handle the other stages of the pipeline and perform any desired evaluations. An engineer using LOST can quickly determine which algorithms are appropriate for their application and evaluate the performance of those algorithms on a battery of generated images representative of their specific imaging hardware. Additionally, because algorithms in each stage of the pipeline can be swapped in and out at will, an optimal combination of algorithms can be selected during runtime in response to various sources of image noise. We supply some insights on how to do this by comparing algorithms of each stage in the Results section.

In “real world” mode, LOST takes an image file as input. The image is sent to the centroiding stage, and the following two stages take input from the preceding stages to ultimately produce an attitude reading.

Figure 2: Sample Generated Image

In generated image mode, LOST generates a realistic star image. Stars are placed in the image according to their real positions in the Yale Bright Star Catalog. Additionally, the image generator is capable of simulating a variety of real-world noise sources, including read noise (Gaussian noise on every pixel), shot noise (Poisson noise due to the discreteness of photons received from low-brightness stars), motion blur, rolling shutter, and randomly occurring false stars. These noise sources are explained in more detail by Delabie.[4] For star-id testing specifically, the image generator keeps track of the “true” or “expected” centroid locations as it generates the image, which can be subsequently fed to a star-id algorithm to test it in isolation without an active centroiding algorithm. The centroids can also be randomly shifted to evaluate the performance of a star-id algorithm versus centroid error.

A number of comparators can be used to automatically evaluate the performance of each algorithm when running in generated image mode. The comparators compare the output of an algorithm with the "expected" output, which is known from the image generator, as generated images store information about the true centroid positions and star IDs. The comparators can then report how many centroids were detected, the mean centroid error, the number of stars correctly identified, etc. (see Evaluation section). The output of the algorithms can also be plotted to an annotated image file to visualize the results and help debug issues in new algorithms.

EVALUATION METHODOLOGY

The evaluation determines a number of key metrics, defined here:

- Availability: The fraction of all images that are reported as identified and the identification is correct (true positive rate). This corresponds to the probability that a run of LOST will yield correct attitude information, which may be a direct mission requirement.
- Error Rate: The fraction of all images that are reported as identified and the identification is incorrect (false positive rate)
    - Note that Availability and Error Rate need not sum to 1. A star-identification algorithm may be unable to identify an image at all, in which case the image is not counted towards either metric.
- Centroid Error: The mean distance between true star centers and detected star centers, in pixels. Only centroid errors from the 5 brightest stars in the image are included in the average, to avoid the confounding effect of noise on the number of total stars visible. This metric has no direct real-world consequences, but instead is useful to compare centroiding algorithms against each other and to determine which algorithm performs best under different expected noise conditions.
- Attitude Error: The minimum rotation angle to bring the detected attitude to the true attitude. In cases when attitude error is averaged over a number of measurements, only the cases when the attitude is "correct" (i.e., within half a degree of the expected attitude) contribute to the average. Consequently, star-id failures, which usually cause a blatantly incorrect attitude, only contribute to the error rate, leaving centroid error as the main contributor to attitude error.
- Desktop speed: The number of microseconds some algorithm takes to run on a desktop computer, with an AMD Ryzen 7900X CPU.
- Raspberry Pi speed: The number of microseconds some algorithm takes to run on a Raspberry Pi 4. Satellite missions increasingly use off-the-shelf hardware similar to Raspberry Pis, so we can expect LOST to have speed similar to these results.
- x86 CPU instructions: The number of CPU instructions executed on an x86 CPU architecture with instruction set extensions disabled. Note that this metric can only very roughly be used to predict speed, because instructions can take varying amounts of time to execute, and vary according to the instruction set used.
- Maximum memory usage: The maximum number of kilobytes of memory used (excluding infrastructure and permanently stored data), measured on a desktop.

The code to run our evaluations can be found at https://github.com/UWCubeSat/lost-evals.

Three types of tests are performed:

- Centroid tests, where centroid error (defined above) is plotted against read noise, number of photoelectrons received per star, and motion blur. As described above, centroid error is mainly useful for comparing between centroiding algorithms.

  Note that noise can also change the total number of centroids detected, but we do not plot the total centroids detected because it is independent of the choice of algorithm. Additionally, only the centroid errors for the 5 brightest stars in each image are included in the average.

  The centroiding tests vary the level of noise starting from a common set of parameters: 25 degree horizontal field-of-view (corresponding to a typical medium-FOV star tracker[7]), a 1024x1024 sensor resolution, 4,000 total photoelectrons received from a magnitude zero star (corresponding to a 200ms exposure with a 3cm lens aperture and typical quantum efficiency, see Liebe[7]), and Gaussian read noise applied to every pixel with a standard deviation of 2 photoelectrons per pixel.
- Star Identification tests, where availability and error rate are plotted against centroid error, number of false centroids, and the dimmest magnitude the camera can pick up.

  The Star-ID tests are run without a centroiding algorithm; centroids are fed in directly from the image generator, as described in the infrastructure section of the paper. The base settings for the star-id test are a 25-degree horizontal FOV, 0.3 pixels average centroid error, stars brighter than 5 magnitude being visible (corresponding to an average of 25 many stars per image), and an average of 1 false star per image.

- Comprehensive tests, where the entire software is run on a small number of scenarios. Availability, error rate, attitude error, and the speed metrics defined above are averaged over 100 trials. The comprehensive tests evaluate both LOST and other open-source star trackers. The goal of these evaluations is to measure the best performance each star tracker can achieve, so we pick algorithms that provide strong performance in each scenario. The four tested scenarios are:
    - 20-degree FOV, low noise: The same settings as in the isolated tests. For the LOST tests, we use the Tetra star-id algorithm and the COG centroid algorithm.
    - 20-degree FOV, high noise: The total photoelectrons received from a magnitude zero star is reduced from 4,000 to 2,000 (corresponding for example to a smaller lens), and motion blur corresponding to about 0.5 degrees of boresight motion and 4 degrees of roll per second of rotation at a 200ms exposure time. For the LOST tests, we use the Pyramid star-id algorithm and the COG centroid algorithm.
    - 45-degree FOV, low noise: The same settings as for 20-degrees, but with a wider 45 degree FOV. Like in the 20-degree FOV low-noise test, the LOST tests use the Tetra star-id algorithm and the COG centroid algorithm.
    - 45-degree FOV, high noise: Same as 20-degree high noise, but with a 45 degree FOV. Like in the 20-degree FOV high-noise test, the LOST tests use the Pyramid star-id algorithm and the COG centroid algorithm.

  In addition to LOST, we evaluate C-Tetra, the original Tetra's author's implementation of the algorithm. C-Tetra only implements the star-identification stage, so we use LOST's centroiding stage as input, and report C-Tetra's availability and error rate based on whether the output of its star-id stage is correct (the attitude error would be the same as for LOST, since star-id accuracy is determined mainly by centroid error). We were unable to get OpenStarTracker to identify the images in our comprehensive test cases, although it is possible there were configuration options we did not set correctly.

RESULTS

Centroiding

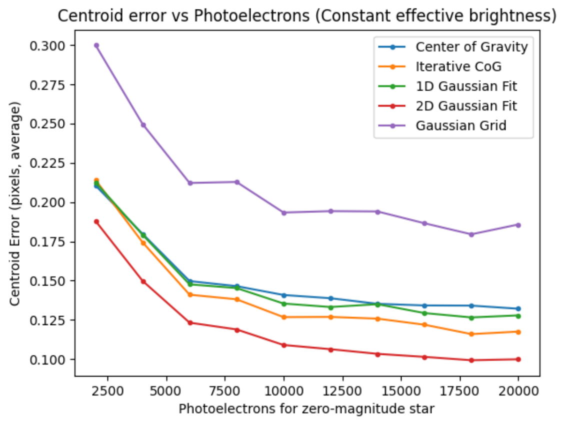

Figure 3: Centroid Error vs. Number of Photoelectrons

In Figure 3 we show how centroid error varies with the number of photoelectrons that the sensor receives from a magnitude zero star while keeping the effective brightness of each star constant by increasing the sensitivity. As photoelectrons per star increases, the effect of shot noise (Poisson noise due to the small number of photons received from each star[4]) decreases. The main factors controlling the number of photoelectrons received per star are lens aperture width, exposure time, and sensor quantum efficiency. The ordering between the centroid algorithms does not change as the number of photoelectrons per star increases, with 2D Gaussian Fit maintaining the best performance over the whole range. Thus, lens aperture and quantum efficiency alone should not play a major role in the selection of a centroiding algorithm.

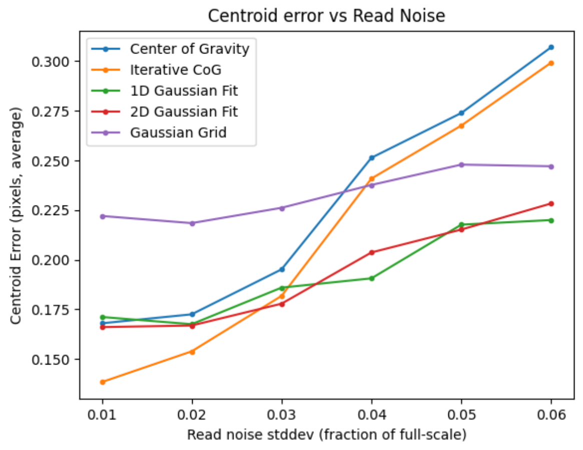

Figure 4: Centroid Error vs. Read Noise

In Figure 4 we compare centroid error with the level of Gaussian read noise (simple Gaussian noise applied to every pixel in the image). Read noise is mainly determined by the sensor being used, and may vary based on the type of sensor technology, the pixel clock rate, and many other factors. The results indicate that at high read noise the Gaussian fit methods are a better choice than the faster Center of Gravity methods, while if read noise is known to be small, the Gaussian fit methods have no benefit over the simpler and faster Center of Gravity algorithms. The Gaussian Grid method, which is much faster than the Gaussian fit methods but more accurate than the Center of Gravity methods at high levels of read noise, may also be appropriate to use in some cases where speed is of priority.

Star Identification

We evaluate star-id algorithms on availability and error rate, as defined in the Methodology section.

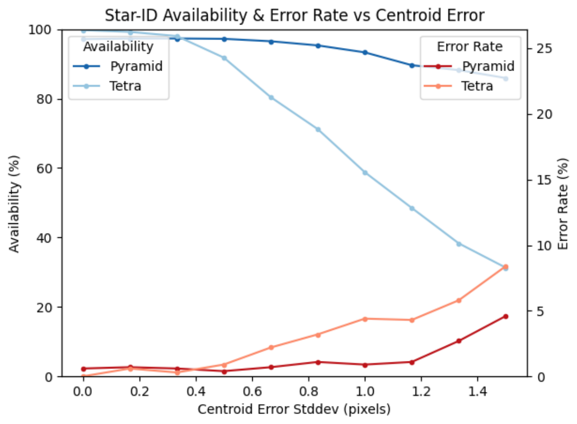

Figure 5: Star-ID Availability and Error Rate vs. Centroid Error

In Figure 5 we show how sky coverage and error rate vary with centroid perturbation (random positional error added to each centroid). Pyramid takes an estimate of the centroid error as a runtime parameter, which we update appropriately for each data point. The availability of both algorithms stays above 90% as long as centroid error is less than 0.5 pixels. (Recall the images used in this evaluation are 1024 x 1024 at a 25-degree FOV width, so 0.5 pixel is about 0.01 degrees). Centroid error can be estimated based on the graphs in the previous section, and is mainly reliant on the camera hardware. As shown in that section, centroid error is below 0.5 in many practical cases. If higher centroid error is expected, then the Pyramid algorithm shows a clear advantage, which is why we use Pyramid for the high noise cases in the comprehensive test (presented later).

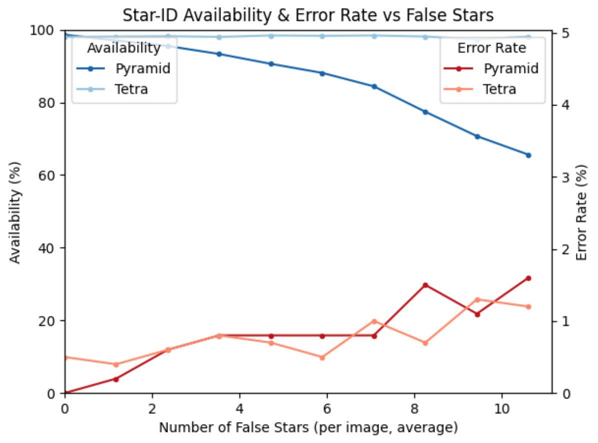

Figure 6: Star-ID Availability and Error Rate vs. Number of False Stars

In Figure 6, we plot availability and error rate against the number of false stars. The false stars are uniformly distributed, with uniformly random magnitudes between 3.0 and 7.0 (Tetra uses the star magnitudes to determine which 4-tuples of stars to attempt to match first). False stars can be caused by clusters of dead pixels, dust, planets, stars not included in the database, or a number of other sources. Tetra maintains higher availability as false stars are added while maintaining the same low error rate as Pyramid, so is preferable when there are many false stars.

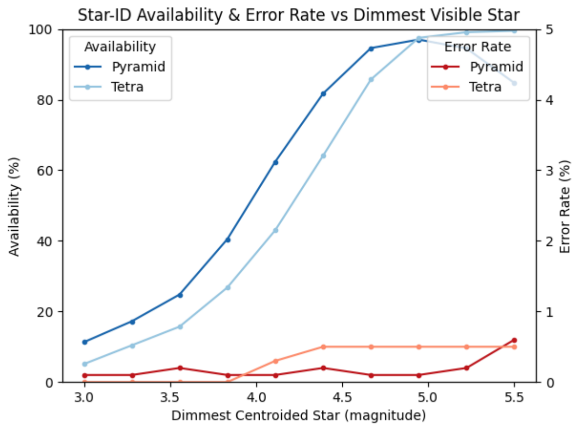

Figure 7: Star-ID Availability and Error Rate vs. Dimmest Visible Star Magnitude

In Figure 7 we plot availability and error rate against the magnitude of the dimmest star the camera system can detect. Note that we plot linearly against magnitude, but the number of stars increases nonlinearly as magnitude increases. At this field of view (25 degrees), the camera system must be able to image stars as dim as magnitude 5.0 in order to achieve close to 100% availability. Both Pyramid and Tetra have a similar dependence on the number of visible stars.

Attitude Estimation

Our QUEST implementation requires less than 1 KB of memory and finishes in less than 1 microsecond on our desktop hardware. It is well-established that QUEST typically solves Wahba's problem to within floating point error, so we do not evaluate attitude estimation further.[3]

Comprehensive

To evaluate the speed and memory requirements of LOST, and to compare it against other open-source star trackers, a number of "comprehensive" tests are performed. As explained in the methodology section, we chose to use different algorithms for the LOST-related rows in each scenario depending on what performed best. The details of the image generation settings used in each scenario are also explained in the methodology section.

Table 1: Comprehensive Testing Results

Scenario	20-deg FOV Low Noise	20-deg FOV High Noise	45-deg FOV Low Noise	45-deg FOV High Noise
LOST Desktop Speed (μs)	2512	4011	2482	3862
LOST Centroid Speed (μs)	2465	2396	2452	2383
LOST Star-ID Speed (μs)	39	1611	24	1475
LOST Raspi Speed (μs)	31,447	104,730	38,231	89,857
LOST CPU Instructions	25,153,935	76,315,835	25,323,548	51,877,918
LOST Centroid CPU Instructions	24,839,326	24,793,313	25,191,130	24,981,901
LOST Star-ID CPU Instructions	314,609	51,522,522	132,418	26,896,017
LOST Centroid Memory (KiB)	30	27	53	40
LOST Star-ID Memory (KiB)	20	676	7	337
LOST Availability (%)	100	65	100	91
LOST Error Rate (%)	0	0	0	8
LOST Attitude Error (degrees)	0.00944	0.06839	0.00696	0.02692
LOST Database Size (KiB)	41,088	336	10,162	303
C-Tetra Star-ID Desktop Speed (μs)	10	10	100	150
C-Tetra Availability (%)	74	8	96	88
C-Tetra Error Rate (%)	0	1	3	10
C-Tetra Database Size (KiB)	5,185,467	5,185,467	4,222,711	4,222,711

In both low-noise cases, LOST produced an availability of 100%. In the high-noise cases, LOST produced an availability of 65% and 91% for a 20-degree FOV and a 45-degree FOV, respectively. For both low-noise cases, LOST produces an error rate of 0%, and for high-noise cases LOST produces an error rate of 0% and 8%. These results indicate that LOST is consistently able to identify images correctly, while preserving a low error rate, so false positives rarely occur.

In both low-noise cases, LOST produces an attitude with error between 0.00944 and 0.00696 degrees. In both high-noise cases, LOST produces an attitude with error between 0.06839 and 0.02692 degrees. It is surprising to us that the attitude error in the 45-degree FOV scenarios is lower than the error in the 20-degree FOV scenarios. While a narrower FOV results in lower *angular* centroid error (the pixel error is the same), the greater number of stars visible in a wider FOV allows the attitude estimation algorithm to find a best-squares solution over a greater number of identified stars, helping to cancel out error. However, for other sets of parameters, especially in low noise conditions, the attitude error may be lower at narrow FOVs.

The speed and memory in all 4 scenarios indicate that LOST could be ported to run on low-compute hardware such as embedded systems. Many low-power embedded systems and relatively low-cost radiation embedded hardware, such as the Vorago Cortex M4 CPUs, have a clock rate of approximately 100 MHz. This is about 1/10th the clock rate of a Raspberry Pi, so it is reasonable to estimate that LOST will take one order of magnitude longer on such low-compute hardware. Specifically, LOST would likely be able to centroid and identify an image in less than 0.5 seconds and could be made faster by decreasing the image resolution. In the high-noise case, LOST might take 1 second or more, which is still acceptable for many missions. Further, the total number of (x86) CPU instructions LOST uses to identify an image is less than the number of CPU cycles most microcontrollers can perform each second, further supporting the idea that LOST could complete an identification in less than a second (although different architectures may require different numbers of CPU cycles). Additionally, LOST's memory usage stays below 1 MiB in all cases, and radiation-hardened memory of this size is widely available.

LOST maintains a higher availability and lower error rate than C-Tetra. Notably in the high-noise 20° FOV scenario, C-Tetra fails to identify the vast majority of the images with an availability of 8%, while LOST is able to keep availability above 65%.

CONCLUSION

We find that the star tracking algorithms implemented in LOST achieve availability of 100% in low-noise cases and at least 65% in cases with higher noise. Further, both centroiding and star identification take a total of less than 3 milliseconds on desktop and < 35 milliseconds on Raspberry Pi while consuming no more than 1 MiB of memory. These results suggest that LOST could be ported to low-compute hardware and run with performance acceptable for many satellite missions.

Our evaluation also includes an analysis of how various metrics vary with different noise sources, to aid in decision-making about which algorithms to use for different missions. In addition to performing the tests listed here, LOST leaves the reader with a framework that allows easily evaluating various algorithms under a custom set of noise sources.

Satellite development is no longer limited to multi-million dollar missions by well-funded organizations. As startups, undergraduate engineering teams, and even individuals increasingly develop their own satellites, the need is strong for affordable and open star tracking software. Our results demonstrate that LOST is a step in this direction

LIMITATIONS & FUTURE WORK

The algorithms currently implemented in LOST are not suitable for all missions. Pyramid requires accurate *a priori* knowledge of the camera system's parameters, such as focal length, in order to reliably identify images. Tetra requires a fairly large database (~70 MB). The centroiding algorithms do not work when there are large disruptions in the field of view (for example, if the Sun or Earth are in the FOV), even if some stars are visible in other parts of the FOV.

At the time of writing, LOST uses heap memory allocation, which is undesirable on embedded systems with limited memory. We plan to add variants of the algorithm implementations that use stack memory only, and have a maximum guaranteed memory footprint. For example, a statically-allocated Star-ID implementation would only be able to function up to some fixed maximum number of centroids, but would be able to identify images using only constant-size arrays and hash tables based on that maximum number of centroids.

In the future, we plan to implement extra auxiliary features in LOST. For example, it is possible to re-estimate the camera's focal length and lens distortion based on a successful star identification, which can improve attitude accuracy in case the lens was slightly deformed during launch.

LOST will be at the core of HuskySat-2's attitude determination and control system. HuskySat-2 (HS-2) is a CubeSat being developed by the University of Washington's Husky Satellite Lab. HS-2, expecting to launch in 2025, is intended as a technology demonstrator for cislunar and deep space attitude control featuring multiple open-source systems, including LOST for attitude determination and reaction wheels for attitude control.

ACKNOWLEDGEMENTS

We would like to thank our faculty advisor Prof. Alvar Saenz-Otero, as well as Prof. Kristi Morgansen, for their support and invaluable help. LOST was developed by members of the Husky Satellite Lab at the University of Washington. We would like to thank all contributors to LOST, especially Allen Aby and Andrew Smith, as well as the rest of the team.

REFERENCES

1. "Arcsec Sagitta Star Tracker," *CubeSatShop*, https://www.cubesatshop.com/product/sagitta-star-tracker/.
2. Brown, J. and K. Stubis. "TETRA: Star Identification with Hash Tables." *Small Satellite Conference*, Student Competition, August 2017.
3. Cheng, Y. and M.D. Shuster, "Robustness and Accuracy of the QUEST Algorithm," http://www.malcolmdshuster.com/Pub_2007a_C_cquest_MDS.pdf.

4. Delabie, T., J.D. Schutter, and B. Vandenbussche, “An Accurate and Efficient Gaussian Fit Centroiding Algorithm for Star Trackers,” The Journal of the Astronautical Sciences, vol. 61, issue 2, March 2014, https://doi.org/10.1007/s40295-015-0034-4.
5. Erlank, A.O., “Development of *CubeStar*: A CubeSat-Compatible Star Tracker.” *Stellenbosch University*, 2013. https://core.ac.uk/download/pdf/37420644.pdf
6. Gutiérrez, S.T., C.I. Fuentes, and M.A. Díaz, "Introducing SOST: An Ultra-Low-Cost Star Tracker Concept Based on a Raspberry Pi and Open-Source Astronomy Software," *IEEE Access*, vol. 8, pp. 166320-166334, 2020, doi: 10.1109/ACCESS.2020.3020048.
7. Liebe, C. C. “Accuracy Performance of Star Trackers - a Tutorial.” IEEE Transactions on Aerospace and Electronic Systems, vol. 38, no. 2, Apr. 2002, pp. 587–99. DOI.org (Crossref), https://doi.org/10.1109/TAES.2002.1008988.
8. Mortari, D. et al. “The Pyramid Star Identification Technique.” *Navigation*, vol. 51, no. 3, Sept. 2004, pp. 171–83. *DOI.org (Crossref)*, https://doi.org/10.1002/j.2161-4296.2004.tb00349.x.
9. “Radiation Hardened ARM® Cortex®-M4.” *Vorago Technologies*, https://www.voragotech.com/products/va41620.