New Algorithms for Automated Astrometry

By

Chris Harvey

A thesis submitted in conformity with the requirements

for the degree of Master of Science

Graduate Department of Computer Science

University of Toronto

Abstract

New Algorithms for Automated Astrometry

Chris Harvey
Master of Science
Graduate Department of Computer Science
University of Toronto
2004

In the 'lost-in-space' astronomy problem we are asked to identify the orientation and field-of-view of an image of the sky by matching stars in an image to stars in a catalog. In this work we present two solutions to the problem.

In the first approach we reduce an image to coarse positional information represented by a bit-string. We show that by making one assumption about the image we can efficiently solve the orientation problem using bitwise logical operations over an index.

In the second approach we search an image for constrained *n*-star shapes and match them to similar shapes in a pre-computed table. We show that by using constrained shapes we can reduce the time to match *n*-star shapes to the order of matching arbitrary 4-star shapes.

Finally, we perform experiments on synthetic and real catalogs and show that both approaches perform well.

Acknowledgements

I wish to thank my second reader, David Hogg, for taking the time to examine this work.

I wish to thank my parents for their constant support. Since this support has been unconditional my siblings and I have always taken it for granted. My parents should know it is nevertheless appreciated.

I wish to thank Cheryl for listening, and for proof-reading this thesis.

Finally, I wish to thank my advisor Sam Roweis. Without his involvement this thesis would certainly not be what it is.

Table of Contents

List of Figures

Chapter 1: The Problem

1.1 The Astrometry Problem

Estimating camera orientation based on star-field images has applications in two related areas: ground-based astronomy, and satellite attitude correction. Images from ground-based telescopes require re-estimation of their orientation for several reasons. First, due to slight mechanical miscalculations the automatic determination of the orientation as the image is taken may not be accurate. Second, if manual telescope pointing is used, orientation measurement may not be reliable. Third, orientation information may not be recorded (as in the case of very old images), or may simply be lost. Errors in satellite attitude estimation, on the other hand, are much more severe. Attitude changes for satellites are often inexactly measured and therefore require periodic re-estimation. Furthermore, for various reasons a satellite may not trust its current estimated frame of reference, and so will require a complete re-calculation of its

orientation. This general problem of estimating of an image's orientation without a starting reference is commonly called the "lost-in-space" problem.

1.2 Available Tools

Although satellite attitude can be partially determined using either the Earth's horizon or the Sun as a guide, both telescope and satellite orientation estimation often requires the use of observed star positions. To this end, a variety of star catalogs have been constructed. These catalogs can vary in size from a few thousand up to hundreds of millions of stars, and include information captured by ground based telescopes. These catalogs typically contain positional estimates for stars across the entire sky (often with tolerances on their errors), as well as estimates of the magnitude of every star in a series of light bands. In practice these catalogs tend to capture the brightest stars in the sky but become less complete as they reach a certain magnitude. With this in mind, the creators of a catalog will typically provide an estimate the catalog's completeness up to particular brightness (*e.g.* 99% complete up to magnitude 11.5).

While we may initially be optimistic about solving the orientation problem by using these catalogs, in practice numerous complications plague the process. In the easiest scenario we are given a large field-of-view and some knowledge of the camera parameters. In this case we can use a few stars from the catalog and employ brightness and geometric relationships to solve the problem. However, in the most general case we cannot rely on significant foreknowledge about the parameters of the images that we will be required to solve. Thus, the fewer restrictions an algorithm employs the better. By far the most common complications occur when brightness information is unknown or highly unreliable. Without accurate brightness information it is difficult for an algorithm

that employs specific brightness relationships to decide which stars to use when the number of stars in the image is large (i.e. > 100). In addition, without foreknowledge of the camera's field-of-view, space and processing demands increase greatly. Not only must many geometric relationships across different scales be stored, but as the image scale increases nearby stars can become difficult to resolve from one another. Finally, images may be missing catalog stars or may contain unknown stars, and the positions of the stars that are known stars may be noisy. Taken together, all these elements conspire to make identifying specific stars very uncertain.

1.3 Existing Solutions

As described above, there are a number of scenarios when star identification is necessary. Consequently, different approaches have been developed that rely on a specific set of assumptions based on the circumstances under which the image was captured. These assumptions include things like: a good initial orientation estimate, the ability to capture a sequence of pictures over time, knowledge that all visible stars will be listed in the search catalog, and foreknowledge of the test image's field-of-view. Below we consider a series of different scenarios and some of the approaches used to solve them.

Since inaccurate orientation information for ground-based telescopes is typically corrected by someone who has knowledge about the camera and image properties, there already exist some methods for solving this scenario. The simplest process involves manually identifying several stars in the image by hand and having a program provide a best-fit estimate of the image orientation. A slightly more complex process involves the user providing a 'good' initial estimate of the image orientation and camera properties. A

program then estimates the image direction by finding stars expected to be visible in the image and using their geometric relationships to determine the exact orientation. Clearly neither of these approaches requires a significant search on the part of the solving program; rather they act to 'clean up' the estimated direction.

Work on satellite attitude correction, on the other hand, tends to be more directly focused on the pure "lost-in-space" problem. Some existing techniques exploit the ability of a satellite to take multiple images in succession while moving in a controlled direction. Work in [17] demonstrates the viability of an iterative process that uses the features of a single central star in the image. During each iteration of the algorithm catalog stars are assigned a probability of being the central star, given the probable orientation of the previous image and the intervening camera motion. The process stops when subsequent images converge to a single highly probable star being the central star. The authors of [13] propose two related methods, both of which use specific and accurately measured star relationships and apply to a spinning satellite. In the first, candidate orientations for the current image are determined based on the angle between pairs of stars. Based on the known motion of the camera, regions of the sky that should never become visible in the future are eliminated. Since the camera continues to rotate in the same direction, subsequent re-estimations of the image orientation determine a single plausible path of motion for the camera. The resulting path can then be used to determine where the camera is pointing at any given moment. In the second approach an index is built linking neighbouring stars to one another. Assuming a slow and known camera motion, candidate orientations are checked against properties of the stars that appear as the camera moves, thereby progressively eliminating implausible orientations.

The previous methods employ weak orientation estimators, but make use of the necessary coherence of subsequent observations to converge on a final solution. These approaches are less useful, however, when the camera has a limited ability to move or when a single image is presented without any other metadata. Therefore, significant work has gone into orientation estimation based on a single image.

If assumptions are made about the angular orientation of the image (*i.e.* that it lines up in a particular way with the catalog), then a brute force method can be employed. The authors of [16] describe a method under this assumption whereby candidate orientations are tested at coarse intervals. When the orientation chosen in the catalog approaches the true orientation, the image stars will have matching catalog stars that are all displaced by a similar offset. The catalog can then be precisely matched to the image.

Naturally, image orientation cannot be constrained in general. The implementation described in [6] employs a wide-field-of-view and accepts only the brightest stars. The algorithm uses the angles between the two nearest stars to any bright star to check a small database. The goal is to identify stars that, based on their position, angular relation, and magnitude, reside in the same constellation. The algorithm then uses the identified constellations as directional guides.

While the method used in [6] is useful, a wide field-of-view limits the accuracy of the orientation estimation. Furthermore, this method cannot translate to images taken with a small field-of-view. There are, however, a number of other approaches that utilize precise geometric relationships between image stars to determine attitude at a smaller scale. [2] presents early work in this area using triangle lists. The algorithm generates triangles from the brightest image stars and compares them against triangles generated

from the star catalog. By comparing the angles and distances between stars in the image and the catalog, votes are assigned to catalog stars whose relationships match image stars. Using these votes an image-to-catalog star matching is determined and a final orientation is calculated. Unfortunately, this approach does not scale well in a large star catalog because it requires significant index space and uses a potentially time-consuming voting process. In addition, this method relies on an ordered ranking of the brightest stars in the image since incorrectly chosen image stars can confound the voting process.

In [7], the authors describe a genetic algorithm that uses distances and angles between stars relative to the center of the image to identify the image's orientation. In this approach a cost function that compares candidate orientations against the image is defined. The algorithm then tries to converge towards a most probable solution, employing randomization to avoid local minima. In [8] the same authors reapply the genetic algorithm approach with some minor alterations. In addition, they apply a neural network and try to classify images according to a known set of positions using the angular and distance relationships between the brightest image star and the nine next brightest. Although both approaches are interesting, they rely heavily on accurate brightness ranking. Additionally, adding or removing bright stars from an image is likely to have a detrimental effect on these approaches.

The approach in [12] also uses angle and distance information, but addresses the problem of brightness ranking and deals with spurious stars created by faulty imaging devices or incomplete catalogs. The approach ignores brightness altogether and simply tries star quadruples in the image, ordering the angular relationships and using them to hash into an index. Furthermore, the authors demonstrate that using a catalog of 5000

stars and reasonable constraints on the positional error virtually guarantees accurate direction estimate once an appropriate quadruple has been found. In practice, and particularly in situations where constraints on the field-of-view are known, this method is effective and very fast. As the size of the star catalog grows and positional error increases, however, matching problems may begin to arise. Furthermore, while more complex star shapes can be used to search in large catalogs, higher order shapes require checking significantly more combinations of image stars.

Although using direct angle measurement is an attractive approach for identifying shapes, [3] instead uses the SVD of the direction vectors of the brightest stars in the image to create a hash code that is more robust to positional noise. While this approach, unlike other shape-based methods, does not require direct star pairing, it does require relative brightness information.

In general, it is clear that specific star angles and distances have been used often and to good effect, despite some lingering problems when brightness information is not present or star density is high. There is, however, an alternative approach: instead of matching stars directly, one can examine the coarse pattern of star positions.

The authors in [1] place a grid over an image whose scale is known, and center and align a grid according to two bright stars. Each grid bin is then marked as either containing a star or not containing a star, thereby creating a binary grid. The binary grid is then tested against a table of grids. Due to the possibility of increased or decreased density of stars in the image as compared to the catalog, Bayesian decision theory is used to adapt the matching function according to the density of stars in the image. The authors report good results in the face of location and brightness noise. A similar approach is

attempted in [5], however instead of using a binary grid the number of stars in each bin is used in a more straightforward comparison. As a result, [5] appears sensitive to large increases or decreases in the number of stars observed.

1.4 Elements of Success

Based on the problems in the existing approaches to orientation estimation, we can define three general goals for any ideal orientation estimating system. First, during matching the system should minimize the number of features that need to be calculated, or the number of times features need to be calculated using different reference points. This means that combinatorial features like arbitrary *n*-star shapes should be avoided if possible, since as the number of image stars climbs the number of permutations of the stars in the image explodes. Second, the process should not require information about the absolute magnitude of stars in order to function, and should preferably not even require a relative ordering of star brightness. Although it is still reasonable to require brightness information, it is not preferable when dealing with all possible identification scenarios. Third, the process should be insensitive to scale and orientation, and permit complete inference of an image's field-of-view. This is the most important property since it eliminates the need to tune the algorithm to the parameters of a specific imaging device.

The preceding three elements are by far the most difficult to attain, particularly in conjunction. Yet even if a process is capable of attaining one or more of the preceding goals, there is another set of easier but more vital goals that should be attained. First, the process needs to be robust to interloping (*i.e*. un-cataloged) stars and to missing stars. While interloping stars are more likely than missing stars in practice, both situations can occur and it is therefore important to deal with either. Second, the process should produce

a relatively small index—i.e. on the order of tens of megabytes if possible. In this vein, the process should permit an optional trade-off between index size and the probability of failure. Third, the features used in the index should be easily searchable so that the number of elements examined at solve time can be minimized. This becomes a significant issue as the size of the index grows or if the number of features that can be identified in an image is large. And last, the process should be robust to jitter of the positions of the stars. Since it is inevitable that some error will occur in the positions of stars identified in the test image, it is important that the process not be dependent on exact values.

Finally, there is one optional but highly desirable property for an ideal orientation solving system: the process should permit the use of additional image information to speed up or improve the accuracy of the matching process, such as foreknowledge about the image's scale, the stars' brightness, or an estimate of the image's direction.

1.5 Assumptions

With a set of goals thoroughly defined in Section 1.4, we can proceed to investigate methods of solving the orientation estimation problem. In the remainder of this work we present two approaches to the problem, both of which have distinct advantages. In these methods the following assumptions have been made:

i) The catalog used to solve images contains star positions that are noise-free.

ii) The transformation from a raw image of the sky into x/y-coordinates has already been performed.

iii) The pre-computation time for any indexes we produce is not important as long as the lookup time is fast.

iv) Interloping stars appear in random places.

v) Positional jitter is random.

Chapter 2: A Grid-Based Approach

2.1 Motivation

As previously mentioned, there are a number of significant drawbacks to using well-measured geometric shapes for orientation estimation. The first issue involves the number of stars per shape. The more stars used per shape the more unique the shapes, and accordingly the less likely we are to find false matches. Unfortunately, since the number of shapes in an image is *n* choose *k* (where *n* is the number of stars in the image and *k* is the number of stars per shape), as *k* approaches $\frac{n}{2}$ there is an explosion in the number of shapes that can be found. The corresponding cost in search time and storage means that we must avoid mid-range values of *k*. As a result, the number of stars used per shape is determined by a trading uniqueness for space and time, and unfortunately *k* often ends up being set quite low. The second issue involves the dependence of shapes on every star being present and correctly positioned within the shape. If positional jitter is added, or

worse yet if some stars in the catalog do not appear in the image, shapes are much harder to find

One solution to these issues is to examine course-grained geometric features based on the entire image instead of fine-grained features based on a few precise positions. Ideally, as few stars in the image as possible should be used to orient the features, reducing the time required to pre-process the image. In addition, by using only a few stars to orient the features, the number of keys required in the index can be decreased. Under this scheme, space is saved, the time required for the key comparison process is diminished, and the probability of false matches is reduced.

In this chapter propose a grid-based approach as an implementation of such a coarse-grained scheme. First, this approach has the advantage of using only two stars to orient the features, thus reducing the search space to *n* choose two, rather than *n* choose three, four or more. Second, it employs a coarse-grained set of features, whose fineness can be increased without a combinatorial explosion in space and time. Finally, with foreknowledge of the expected number of stars per image, the feature fineness can be adjusted to optimize the tradeoff between index space and the probability of false matches.

Specifically, given a field-of-view at which all the test images will be generated, we propose to lay a grid over the images at a scale significantly greater than the positional accuracy of the stars. Our coarse-grained feature is a record of the resulting bins that contain less than a preset number of stars, which allows the grid to be compared against an index. The positional accuracy of the bins can be controlled by adjusting the size of the grid and the number of stars used to determine an ‘empty’ bin.

The major difficulty in this process is figuring out how to compare the grid placed over a test image to grids in the index. In [1], Bayesian decision theory is used to decide on the most likely match. Instead, we propose taking advantage of the standard reason image grids and indexed grids fail to exactly match—camera exposure time. In practice, without foreknowledge of the camera's parameters, we cannot expect an image to contain all (and only) the corresponding catalog stars. Rather, we can expect that the exposure of the test image is either longer or shorter than the exposure of the images used to build the catalog. Therefore, we operate under the assumption that either catalog stars will be missing in the image *or* unknown stars will be added to the image. We further assume that both situations will not occur simultaneously.

2.2 Step 1: Indexing

The indexing phase requires us to build grids over pairs of stars from a catalog, record the resulting bin information, and save the data in an index file for later use. The input to the indexer consists of several parameters: a star catalog, the field-of-view of the images to be solved (θ), a fineness parameter which specifies the scale of the grid (f), and a threshold parameter which specifies the number of stars at or below which a bin is considered 'empty' (t).

The first step is to determine which pairs of stars to index, since indexing every pair of stars is unnecessary and therefore wastes time and space. Instead, we pre-filter the catalog to determine the minimal set of stars required to guarantee any image will contain at least one pair. However, because of the possibility of star drop-out it is important that the stars we use should maximize the probability that any image in which the indexed stars *could* appear they *will* appear (*i.e.* we should prefer pairs of bright stars over pairs of

dim stars). To find these pairs we can use the following rule: a pair of stars should be indexed if there is some region of the sky (where a region is determined via the field-of-view parameter θ) such that it is the brightest pair in that region. The algorithm presented in pseudo-code in Listing 1 below can be used to accomplish this task.

```
Input
   Catalog:   A list of stars consisting of position and brightness.

Output
   KeyPairs:  A list containing the ID's of pairs of stars from the catalog
              that should be used for indexing.

Variables
   Map:       A representation of the sky, broken into pieces that are
              either 'covered' or 'uncovered'.
   C:         A copy of the catalog stars, sorted by decreasing brightness
   S:         A set of 'useful' stars--stars considered to be good candidates
              for use in forming pairs of stars to index.
   s:         A single star.
   CurStar:   The star from C we are currently examining.
   StarArea:  The area of the sky over which any image will contain a star.
   InterArea: The intersection of the areas over which any image will
              contain  two stars
   I:         A counting variable.

Functions
   RegionNear(star): A function that takes a star and returns the region of
                     the sky over which any image will contain that star.

Algorithm
   1  Map{} ← {uncovered}
   2  KeyPairs ← {}
   3  C ← sortDecreasing(Catalog)
   4  S ← C[1]
   5  I ← 2
   6  While (Map{} ≠ {covered}) And (I ≤ |C|)
   7    CurStar ← C[I]
   8    StarArea ← RegionNear(CurStar)
   9    If Map{StarArea} ≠ {covered}
   10     For All s ∈ S
   11       InterArea ← StarArea ∩ RegionNear(s)
   12       If Map{InterArea} ≠ {covered}
   13         KeyPairs ← KeyPairs ∪ {CurStar,s}
   14         Map{InterArea} ← {covered}
   15       End If
   16     End For
   17     S ← S ∪ CurStar
   18   End If
   19   I ← I + 1
   20 End While
```

Listing 1: Pseudo-code for an algorithm to choose pairs of stars for later indexing.

In lines 1 through 5 of Listing 1, the algorithm is initialized. A map of the sky [`Map`] is built with all sections set to 'uncovered'. The set of pairs of stars to index [`KeyPairs`] is set to empty, the set of catalog stars [`Catalog`] is sorted by descending brightness, and the set of 'useful' stars [`S`], (that is the set of stars likely to produce index-able pairs in the future), is set to contain the brightest catalog star. In line 6 we check to see if either the map is completely covered, or we have run out of catalog stars to investigate, and in either case the algorithm ends. In lines 7 and 8 we determine the next available catalog star [`CurStar`] and calculate the region of the sky in which it is visible [`StarArea`]. That is, based on the field-of-view θ we determine the maximum angular distance from which the star is guaranteed to be visible in any image. In lines 9 through 18, first we determine if the current star's visible region is considered completely 'covered'. If it is not, we check the set of 'useful' stars, in order from first to last entered, against the current star and see if the region of the sky in which an image in guaranteed to contain both the current star and the 'useful' stars is not yet 'covered'. If it too is 'uncovered', we add the current star and the appropriate 'useful' star to the list of key pairs, and we mark the appropriate region of the sky map as 'covered'. Once all the stars in the set of useful stars have been checked against the current star, the current star is added to the set of useful stars.

This algorithm accomplishes the goals previously laid out: first, it performs the general task of ensuring any image taken of the sky at the appropriate field-of-view will contain a pair of indexed stars. Second, it guarantees that the two brightest stars in any image will be an indexed pair of stars. This property is guaranteed through the combination of several elements:

i) Stars are examined from brightest to dimmest.

ii) Any star that can conceivably cover a part of the uncovered sky in the future is added to the useful star list for later use, thus ensuring that brighter stars that *could* be used in pairs in the future are kept around for this purpose.

iii) Since the useful stars are checked against the current star in order from first to last entered, key pairs containing brighter stars are always created rather than pairs containing dimmer ones.

Unfortunately, by ensuring that the brightest pair of stars in any image is indexed we cannot guarantee a truly minimal number of stars in the catalog are used as key pairs. However, the need to ensure that indexed pairs are actually visible in test images certainly trumps the need for minimal coverage.

With a complete set of star pairs that covers the sky, we can begin the second step of indexing—the process of constructing grids over the pairs of stars and calculating the resulting bin information.

For each pair, we first determine all the stars in the catalog that could possibly be visible in an image in which the current pair of stars is visible. Next, we construct a two dimensional image with the stars rotated and translated so that one of the paired stars sits at the center of a graph, and the other sits directly to its right (see Figure 1).

The image is then broken into a grid according to the fineness parameter f and the resulting bins are checked to determine which contain less than the threshold number of stars t, and which contain more than the threshold number. Those that

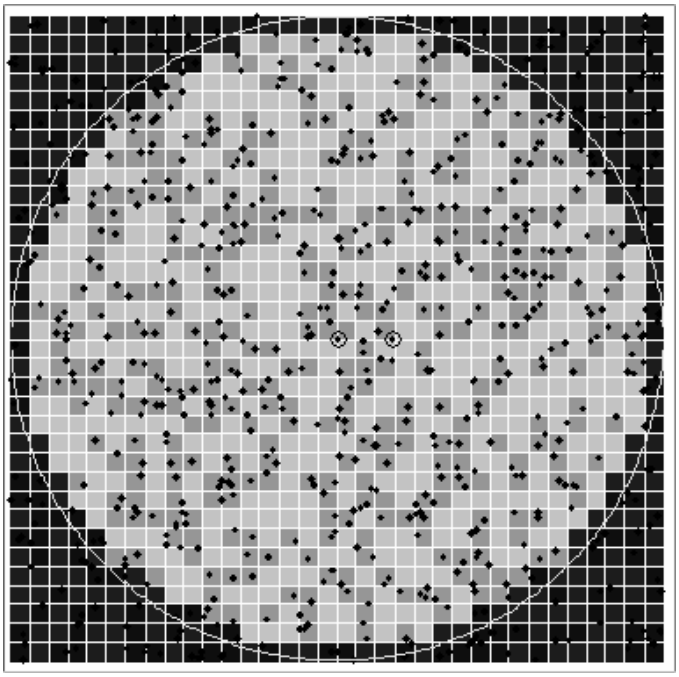

Figure 1: An image of a 'binned' pair of stars. Black dots inside the circular region indicate stars. Circled at the center sits one paired star, and to the right of it sits the other. Black squares outside the circle indicate regions of the catalog not fully visible in images containing the pair. Inside the circle lighter blocks indicate regions containing zero catalog stars and darker blocks indicate regions containing at least one catalog star.

contain less than the threshold we consider 'empty', and those above the threshold, 'full'. It is important to note that depending on the fineness of the grid a certain number of the bins will only fall partially within the previously calculated visible region (*e.g.* those indicated by the black squares partially inside the circle in Figure 1). In order to avoid problems using these partial bins we simply ignore them.

After all the star pairs have been 'binned' we build an index file containing the paired stars and their grid information. The representation of the grid within the index file is dependent on how we intend to match the image grid and therefore these details will be more fully discussed in Section 2.4 (Optimizations).

2.3 Step 2: Matching

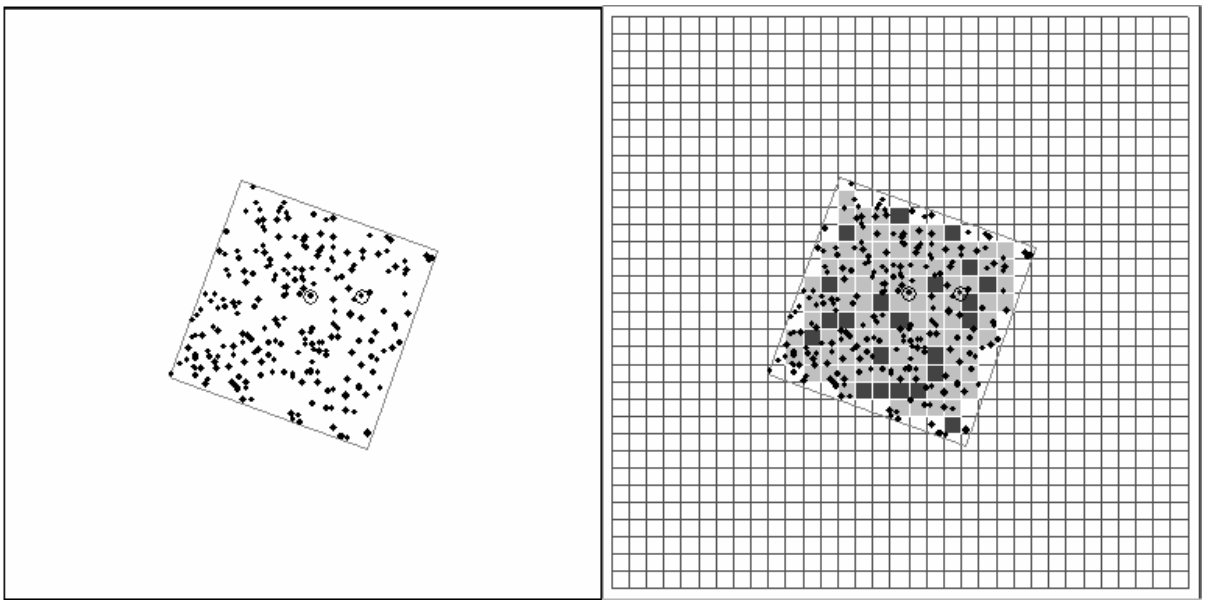

Figure 2: On the left, a sample image after rotation and translation. The star pair used for the transformation is circled. On the right the resulting 'binned' image.

The process of matching an image to the index mimics the indexing process. Pairs of stars in the image are used to create a grid which is then examined to determine which bins are 'empty' and 'full' (see Figure 2). The resulting bin information is then used to find a match in the index.

The first step is to decide which pair of stars in the image to build the grid around. Recall that not every pair of stars in the catalog was indexed, and thus an arbitrary pair of stars in the image is unlikely to be an indexed pair. Unfortunately, in the absence of brightness information we must proceed with the brute force approach and try every combination of two stars. Where brightness information *is* available, we test brighter pairs before dimmer pairs. Since the set of star pairs used during the indexing phase was chosen to ensure the brightest pair of stars in any image is indexed, we construct a subset

of image stars that we are confident contains the image's two brightest. Then, we only need to examine all pairs of stars within this subset rather than all pairs in the image.

Regardless, for each star pair we mimic the indexing process. First, we rotate and translate the image to place the current star pair at the center of a graph. Next, we place a grid over the image and find out which bins are completely within the image boundaries. We determine which bins are 'empty' and which are 'full', and finally we match the bins against the index to produce a list of candidates for the true identity of the pair.

One immediate concern is that our list of candidates may consist of many possible solutions. Since each pair of candidates will require a thorough comparison of the stars in the image against the catalog in order to verify the solution's correctness, we would like to minimize the number of candidates tested. To improve the accuracy of the candidate list we perform the following additional step. When we examine a pair of image stars we perform the binning process twice: once with the first star at the center of the grid, and again with the second star at the center of the grid. Before running any candidate pair through the final correctness check we compare the two lists—the pair of correct solution stars should be present in both candidate lists with the order of the pair reversed (see Figure 3). If this check fails, the pair cannot be a true solution and it is disregarded. This extra process has the advantage of increasing the total number of bins checked for any pair: because the grid alignment is different depending on which star is at the center, we don't examine identically placed bins in the image each time. Additionally, because of this alternate placement of the grid, different bins are visible along the edges of the image, again increasing the variety of the bins checked.

Figure 3: An image of the double matching process. The same image and the same pair produce two different solution lists depending on which star centers the grid (the double-circled star) and which lies along the x-axis (the single-circled star). A good solution should appear in both lists with the order of the pair reversed.

The real question remains as to how we compare the image bins against the index. The obvious approach is to match the image bins directly with the index, and to employ some distance function or correction scheme based on how positional jitter in the image may have affected the bins. The major cause of matching error, however, is when interloping stars appear in the image or when catalog stars drop out of the image. Fortunately, we know the reason for interlopers and drop-outs—differences in the exposure time of test images relative to the exposure of the images that compose the catalog. Therefore, we know that we will tend to only see interlopers when the test image's exposure is longer than the catalog's, and only see drop-outs when the exposure is shorter than the catalog's. This fact means we can tune the matching process to address each of these two problems separately rather than at the same time.

As a result, the matching process is a logically deductive one. Assuming that we see interlopers in the image (and only interlopers) we know that if we choose the correct pair of image stars with which to determine the grid, the only conceivable difference between the bins in the image and those for the corresponding pair in the index should be that some bins that are 'empty' in the indexed grid are now 'full' in the image grid. That is, bins that are full in the index must also be full in the image. As a consequence we can eliminate a pair of stars in the index from candidacy if they contain a full bin where the image contains an empty bin. Likewise, the opposite relationship holds true when there are drop-outs (and only drop-outs)—we can eliminate a pair of stars in the index from candidacy if their grid contain an empty bin where the image contains a full bin.

If we represent a bin as a binary value, we can compare bins using a logical-or operation. More specifically, if we view the bins in the index and image as forming two bit-strings we can construct codes that can be compared using only a single long logical operation. In the case where we only expect interlopers we do the following: set the index bins that are full to 0's and bins that are empty to 1's, and set the image bins that are full to 1's and bins that are empty to 0's. Then by performing a logical-or on the bits of these strings, zeros only occur when index bins are full and image bins are empty, and so a result of all 1's indicates a possible match (see Figure 4). The same operations can be used when drop-outs are expected by reversing the values applied to each condition in the index and image.

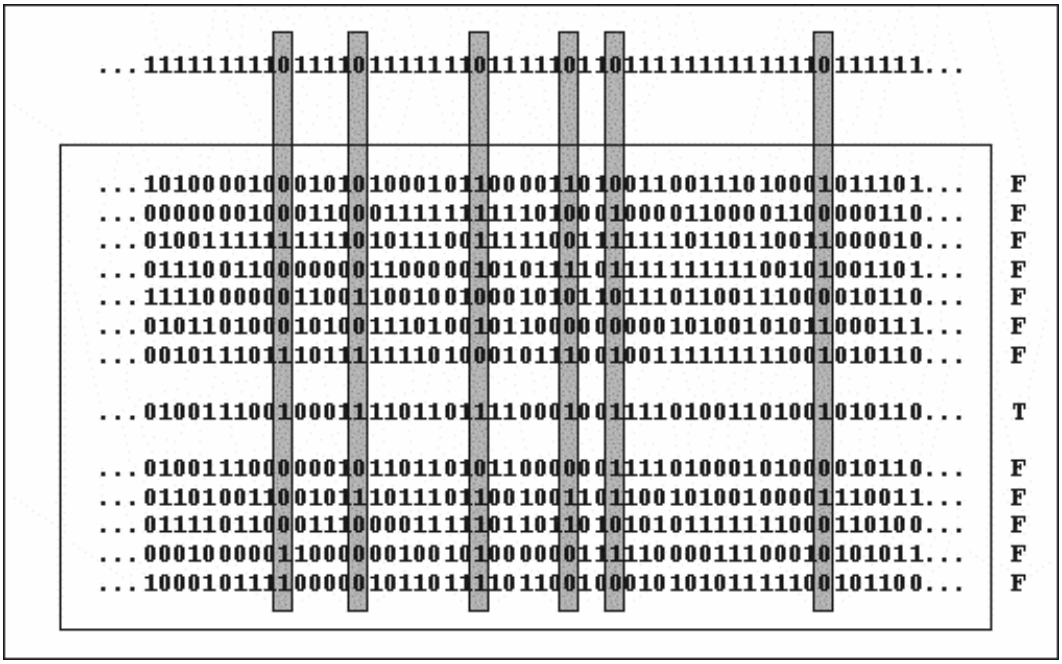

Figure 4: An image representing the matching process of an image code (at top) against indexed codes using a logical-or operation. Codes that, when or'd with the image code, result in no zeros are solution candidates and result in a T (or true) result. Codes that result in one or more zeros fail.

2.4 Optimization Issues

Now that we a clear idea of how to match codes we can optimize the index to control the space and time requirements of matching. Here, there are two major issues: first, how the data should be represented in the index file, and second, how fine a grid should be used.

2.4.1 Data Representation

Since the code-matching procedure is the equivalent of a single logical-or (see Figure 4), we are only interested in checking the bins in the index where the image code contains a zero.

To this end we construct a table where each row corresponds to a particular bin, and each column corresponds to a particular pair of indexed stars. Each entry in the table indicates whether the indexed pair's bin is 'full' or 'empty' at the corresponding bin

position. When we match we maintain an indicator for the candidacy status of every pair—the indicators all start off set to true, indicating that all pairs are possible solutions. To find the set of candidate solutions we examine the rows in the index that correspond to bins set to zero in the image. As we examine each row we set the status indicators for a star pair to false if the bin status for the pair violates the matching rule. In practice the bin information and status indicators can be can be represented as bit-strings and can be quickly manipulated using logical operations if we save the data in the index by row. So, for example, if we construct the index table so that full bins contain a 1 and empty bins contain a 0, we can perform a logical-and over every discriminating row to solve for the interlopers-only case. After performing a logical-and over all the useful rows, bits that still contain a 1 correspond to plausible solutions. In addition we only require a single index to perform either of the matching rules—using the previous arrangement we can simply perform a logical-not before the logical-and to solve for the drop-out case as well.

2.4.2 Grid Size

The more bins we use to index with, the more space we require and the more calculations we must perform when matching. At the same time we would like to maximize the number of different image codes available to us, since any indexed pairs that have the same code will be indistinguishable and will require direct comparison of star positions to determine which solution is correct. Our goal then, if we know in advance the expected number of stars per image, is to choose a grid fineness that gives us the largest expected number of different image codes.

Since the bins are only binary in value, if we are given a particular number of bins per image, n, and a particular number of empty bins in an image, k, the number of

possible codes is *n* choose *k*, which is maximized when $k = \frac{n}{2}$. Of course for any particular image we don't know in advance *exactly* how many bins will be empty or full, but we do know that the probability of exactly *k* bins being empty is distributed according to the binomial distribution $\binom{n}{k} p^k (1-p)^{n-k}$, where *p* is the prior probability that any particular bin is empty. As a result our goal is to manipulate *p* so that the expected number of codes is maximized, which always occurs when $p = 0.5$. As a consequence, when we bin the images we'd like to ensure that for each bin $P(\text{stars in bin} \le \text{star threshold}) = 0.5$ (recall that the star threshold refers the number of stars at or below which a bin is considered 'empty').

The distribution of stars in the bins can be approximated by a Poisson distribution where:

$$\lambda = \frac{\text{expected stars per image}}{\text{number of bins per image}}$$

We can't control the expected number of stars per image, but we can control the number of bins per image. As a result we'd like to choose the number of bins per image such that λ results in:

$$Poisson(\text{stars in bin} \le \text{star threshold}, \lambda) = 0.5$$

If we set the star threshold to zero, solving this problem is trivial. We get:

$$\begin{aligned} & Poisson(\text{stars in bin} \le 0, \lambda) &=& \frac{\lambda^0}{0!} e^{-\lambda} \\ \Rightarrow \quad & \lambda &=& -\ln(0.5) \end{aligned}$$

So that given the expected number of stars per image, and a threshold of zero stars, we should use a grid size that gives us a number of bins per image according to:

$$\text{number of bins per image} = \frac{\text{expected stars per image}}{-\ln(0.5)}$$

If we use a non-zero threshold the problem requires more work however. In general, where the threshold used is *t*, we get:

$$\left(\frac{\lambda^0}{0!} + \frac{\lambda^1}{1!} + \ldots + \frac{\lambda^t}{t!}\right)e^{-\lambda} = 0.5$$

which can be solved using numerical methods on a case by case basis.

This raises the question: which star threshold is preferable? The answer is related to the number of stars we expect to see in the test images, and the positional accuracy of the star positions. In theory the specific value does not matter so long as it permits us to use a grid with fineness significantly larger than the positional accuracy of the stars in the image. In practice a threshold of zero or one stars is preferable, if possible, since the optimal number of bins per image can be calculated most easily using these thresholds.

2.5 Running Time

With the indexing and matching algorithms explained, the question remains as to how efficient these algorithms are. In this section we break down the running time for the indexing and matching steps.

2.5.1 Indexing Time

In the first phase of indexing we calculate a set of star pairs. In the worst case, we compare every star in the catalog against every other star, giving us a running time of $O(n^2)$. For each possible pair we check the region in which both stars can be seen in images together. By using a sectioned map of the sky we can find this intersection and

check for 'uncovered' regions of the map in time $O(k)$ for each pair, where k is the number of sections in the map. This makes the pair generation time $O(n^2 k)$, where n is the number of stars in the catalog, and k is the number of regions used to partition the sky.

In the second phase, we loop through the pairs of stars from the first phase, which requires time $O(p)$, where p is the number of pairs of stars. For each pair, we construct an image using the catalog. For each image, we construct a grid and check for 'full' and 'empty' bins, which requires we place every star in the image into a bin, and declare each bin either 'full' or 'empty'. This image generation and bin check can be done in time $Max(O(n), O(b))$, where n is the number of stars in the catalog, and b is the number of bins used in indexing. Therefore, assuming that the number of stars in the catalog is greater than the number of bins used for indexing, (which is a safe assumption), the second phase can be done in time $O(pn)$, where p is the number of star pairs, and n is the number of stars in the catalog.

Thus, in the first phase the star pairs can be found in time $O(n^2 k)$, and in the second phase the pairs can be indexed in time $O(pn)$, where n is the number of stars in the catalog, k is the number of regions used to partition the sky, and p is the number of star pairs generated.

2.5.2 Matching Time

In the matching phase, at worst we have to examine every pair of stars in the test image, requiring time $O(m^2)$, where m is the number of stars in the image. For each pair, we rotate, translate, and place every star in the image into a bin, and then determine a list

of discriminating bins. We then perform logical operations over the discriminating rows of bins in the index to eliminate star pairs, requiring time $O(bp)$, where b is the number of bins used for indexing, and p is the number of star pairs in the index. However, using a bit-string structure for the index lets us perform the logical operations in a bitwise fashion using word-sized sections of memory, meaning the running time can be reduced to $O(b \log_w(p))$, where w is the word size of the CPU in bits. The matching process therefore requires time $O(m^2 b \log_w(p))$, where m is the number of stars in the image, b is number of bins used for indexing, p is the number of pairs in the index, and w is the word size of the CPU in bits.

2.6 Experimental Results

In this section we explore the effectiveness of the grid-based approach at solving star images. To this end, an image solver was written and tested against a uniformly distributed catalog and synthetically generated images. The catalog was based on the Tycho2 catalog which contains approximately 2.5 million stars. This catalog is not uniformly distributed however, so a subset containing 300,000 of the brighter stars was sub-sampled to create a roughly uniform sub-catalog. Figure 5 shows the density of the initial catalog and the sub-sampled catalog.

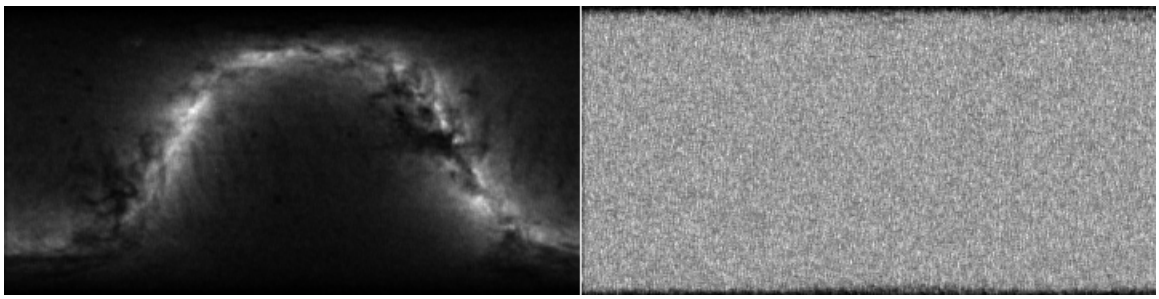

Figure 5: On the left, an image representing the density of the original Tycho2 catalog. On the right, an image representing the density of the sub-sampled Tycho2 catalog used in the grid-based experiments. The images vary in declination vertically and in right-ascension horizontally.

When images were generated, un-cataloged distractor stars were drawn from randomly generated distractor catalogs with uniform positional distribution and a Gaussian brightness distribution with the same brightness mean as the sub-sampled Tycho2 catalog. This means that the synthetic images contained distractor stars that were brighter than at least some of the catalog stars. In practice since catalogs contain the brightest stars in the sky, we would expect any distractor stars to be dimmer than the catalog stars. So, by using a slightly incorrect distractor brightness distribution we also test the robustness of the solver to extreme starting conditions.

Notice that in Figure 5 the sub-sampled catalog contains very few stars in the top and bottom, corresponding to extreme values in declination. This affectation is a result of the Tycho2 catalog, which tends to be sparse at the poles. Unfortunately this means that images generated near these regions have fewer stars than would normally be expected. Figure 6 shows the distribution of catalog stars in the synthetic test images. Notice that the histogram is largely Gaussian, but that a number of images with low star counts were also generated as a result of the density at the poles.

A set of 1000 synthetic images was produced containing 110 catalog stars each, on average. The set of images was then augmented with distractor stars from several distractor catalogs. Each distractor catalog contained an increasing number of stars, ranging from half the number of catalog stars up to five times the number of catalog stars. Several indexes were built, and each was tested against the test images with various numbers of distractors. The results of the experiments were clear: without jitter, the solution to every image was found, regardless of the number of distractors. This is not

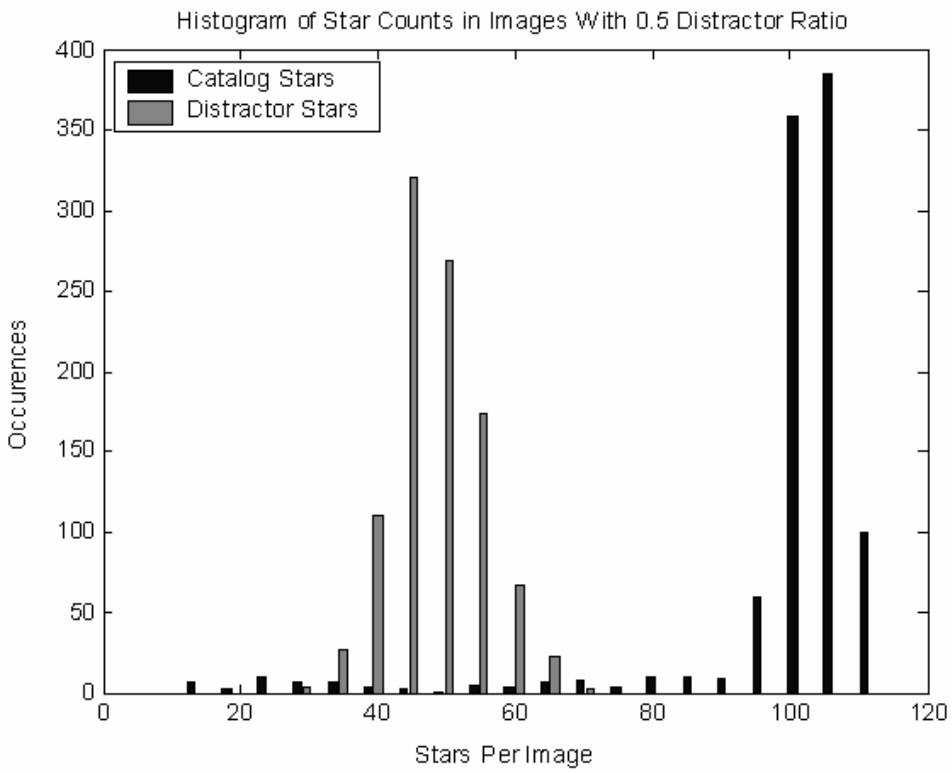

Figure 6: A histogram showing the distribution of the number of catalog and distractor stars present in the test images when a distractor ratio of 0.5 distractor stars for every catalog star was used.

surprising—if the algorithm for choosing stars for indexing works correctly there must always be an indexed pair of stars in every image. Once found, this pair must eventually allow the algorithm to determine the correct solution. The real issue however, is how much space was required to attain these results, and how many incorrect solutions were found before the correct solution was determined.

2.6.1 Index Space Requirements

To examine the effect of the binning size on the matching process, a variety of binning sizes were used, specifically $3 \times 3, 5 \times 5, 7 \times 7, ..., 29 \times 29$ bins per image. It is important to note that by identifying the indexes this way we are referring to the number of bins that can fit into a test image. This size does not directly indicate the number of

bins that are used around each pair in the index, since a larger area around the star pair is required to ensure that all conceivable test images containing the pair can be properly matched (recall Figure 1 in Section 2.2). The calculation for the number of bins can be broken down as follows: if the test images have the aspect ratio $X \times Y$ then we must use a circular region with radius equal to the corner-to-corner distance across the image. So, if the images have an aspect ratio of 1×1, the radius of the circle is proportional to $\sqrt{2}$, and the area proportional to 2π. As a consequence, for example, where a 1×1 section of the sky corresponds to 13×13 bins, the index will generate approximately 1062 bins. This means that for each indexed pair of stars 1062 bits are required.

The experiments used images containing on average 110 stars, and to facilitate full catalog coverage approximately 1 million star pairs were required. In the case of 13×13 bins per image, this leads to a predicted index size of 132 Megabytes. Figure 7 summarizes the space requirements of all the indexes used in the experiments, and verifies the accuracy of our predictions.

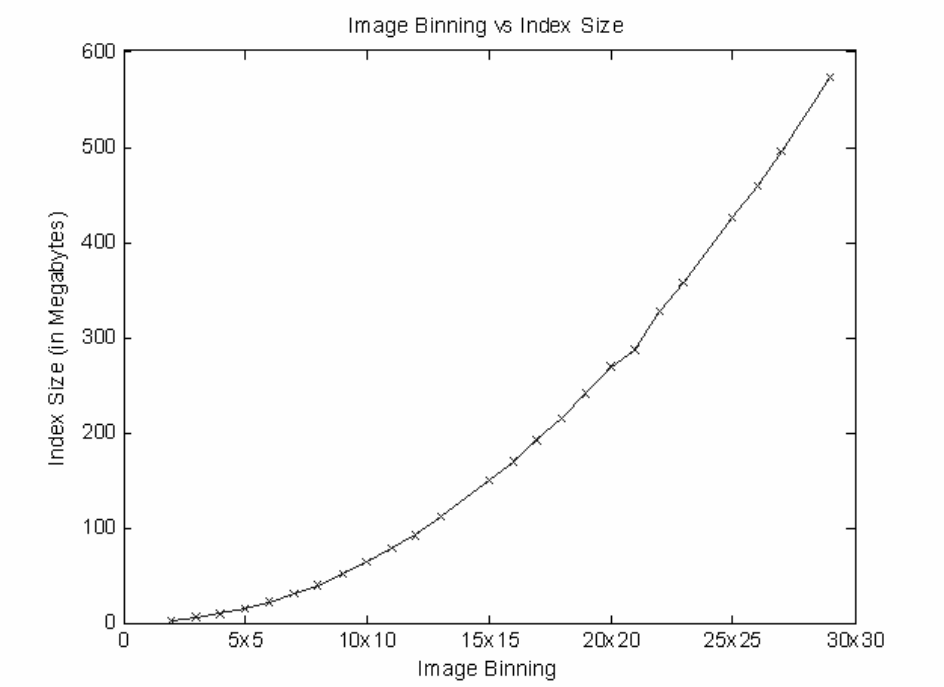

Figure 7: The size of the indexes, in bytes, versus the number of bins contained in each test image.

2.6.2 The Effects of Grid Size

As discussed in Section 2.4.2, if we know in advance the number of stars we expect to find in an image, we can tune the grid size to maximize the number of available codes and to minimize the number of required bins. Using the calculations derived in that section, where the star threshold used to determine full and empty bins is set to zero, 110 stars per image requires about 160 bins. For grids built using an equal number of bins along the x and y-axes, we would therefore expect a grid size of 13×13 bins per image to be the closest to having 50% of its bins empty. Figure 8 shows the experimental measurement of this property across the test images, and proves our hypothesis correct. Figure 8 also shows how the 50% rate changes as distractor stars are added to the test images.

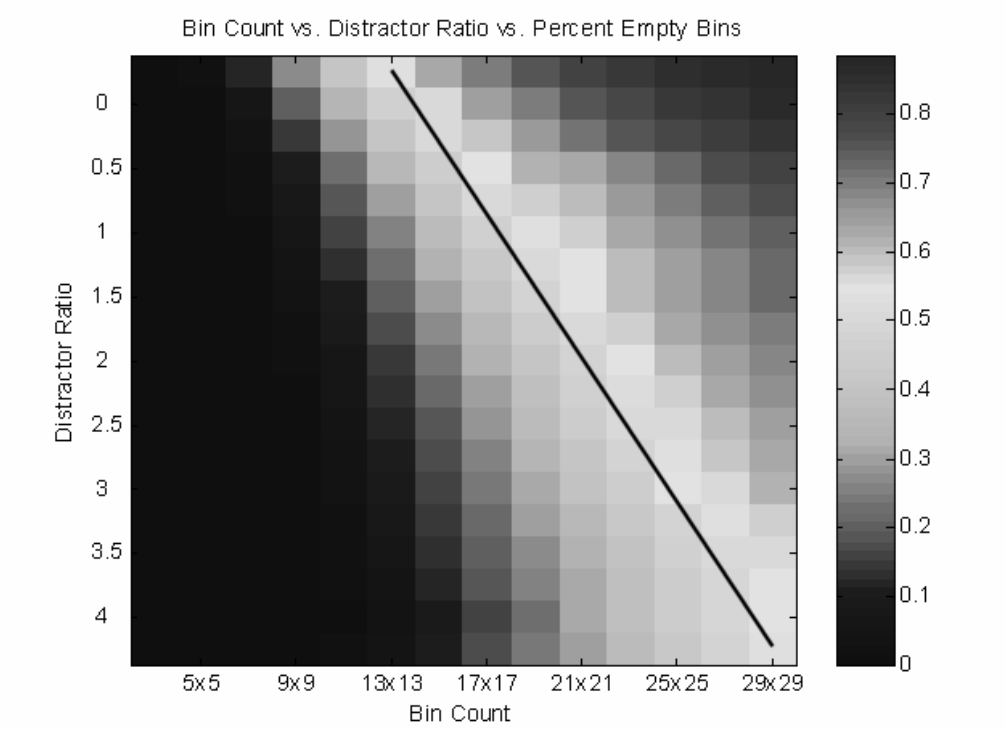

Figure 8: An image of the average proportion of empty bins found in the test images according to each binning size. The dark line roughly marks where the bins are 50% empty. The distractor ratio refers to the average proportion of distractor stars in an image relative to catalog stars. So, given an image with 100 catalog stars, where the distractor ratio is 4.0 we would expect an extra 400 distractor stars to be present.

2.6.3 The Final List Length

After the matching algorithm is run against an image, a list of candidate solutions is identified which will hopefully contain the correct identities of the stars used to orient the grid. Naturally, a final check of these candidate solutions should be performed using the complete catalog before deciding on a final answer. Since this check requires a thorough examination of the image however, it should be avoided as often as possible. Therefore, one area of concern is the number of candidates returned by the algorithm. Figure 9 provides a summary of the number of candidates returned per image at various grid sizes, when no distractor stars were added. To summarize the results: when indexes used at least an 8×8 grid only a single candidate solution was almost ever returned. On

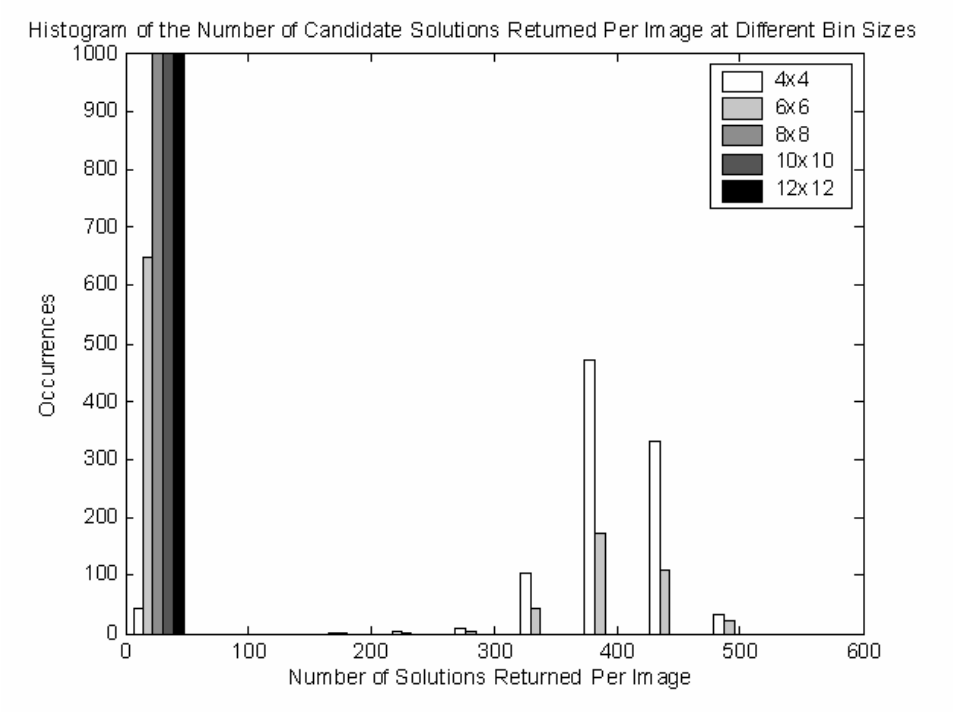

Figure 9: The distribution of the number of solution candidates returned per image at various bin sizes. No distractors were present in the test images. Note that when the bin size was at or above 8x8 seldom more than one solution candidate was returned by the algorithm.

the other hand, when an image grid below 8×8 was used many solutions could be returned.

Due to the scale of the histogram in Figure 9, there is no clear breakdown of the number of candidates returned when fewer than a hundred were found. Figure 10 provides a clearer representation of the results for a single binning size of15×15. In this case the length of the candidate solution list is compared at various distractor ratios, and demonstrates that a manageable number of candidates is usually provided by the algorithm, even when many distractor stars are present in the image.

These results demonstrate the effectiveness of the matching algorithm at keeping the number of candidate solutions small, however the results only refer to the candidate lists generated when an indexed pair of stars was chosen for solving. In practice we cannot guarantee that the correct pair will always be chosen, especially if the pairs in the

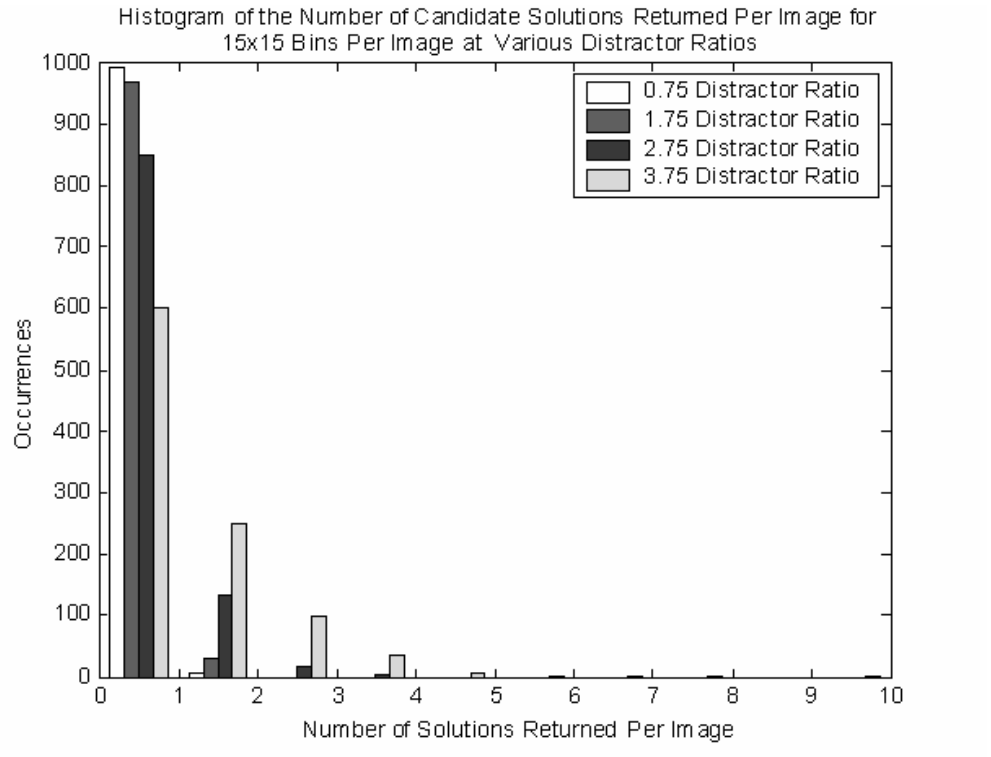

Figure 10: The distribution of the number of solution candidates returned per image for a 15x15 bin index as the number of distractor stars was varied.

image cannot be ranked by brightness. The next question therefore, is how often candidate solutions are returned when an inappropriate pair of stars is chosen for solving. Figure 11 demonstrates how this value varied for a 12×12 bin sized index. To summarize, when the number of distractors was less than one-and-a-half times the number of stars in the image, the algorithm seldom returned a false solution. Above a distractor ratio of one-and-a-half, the matching algorithm began to return false solutions a significant percentage of the time. It is important to remember however that false candidates can be reduced by increasing the number of bins used for indexing. A 12×12 bin size was clearly inappropriate as the number of stars in the image increased. At the extreme in Figure 11, each image contained 400 stars, but at most 144 bins were used in the solving phase. It is no surprise therefore that false solutions were encountered.

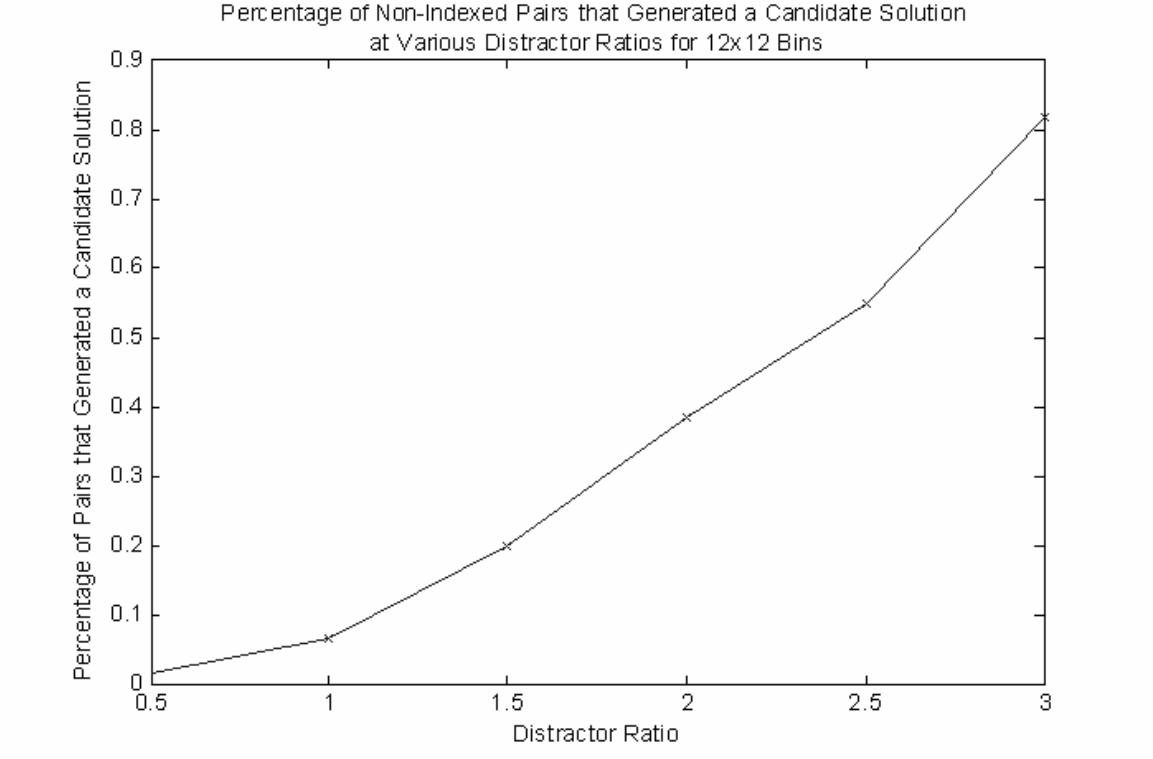

Figure 11: A plot of the percentage of the time that a non-key pair generated at least one candidate solution, for an index using a 12x12 binning size.

2.6.4 Effects of Jitter

As a final step, we tested the effects of jitter on the grid-based method. When a maximum estimate on positional error is available we can expand and reshape the grid to compensate for this problem. Here, we tested how sensitive the method was to jitter when no explicit compensation was performed. The results (see Figure 12) show that the approach is very sensitive to jitter when compensations are not made. The success rate dropped to almost twenty percent when the jitter was only two percent of the size of the bins used for matching. This suggests that we must be very aware of properly estimating positional error when using the grid-based approach.

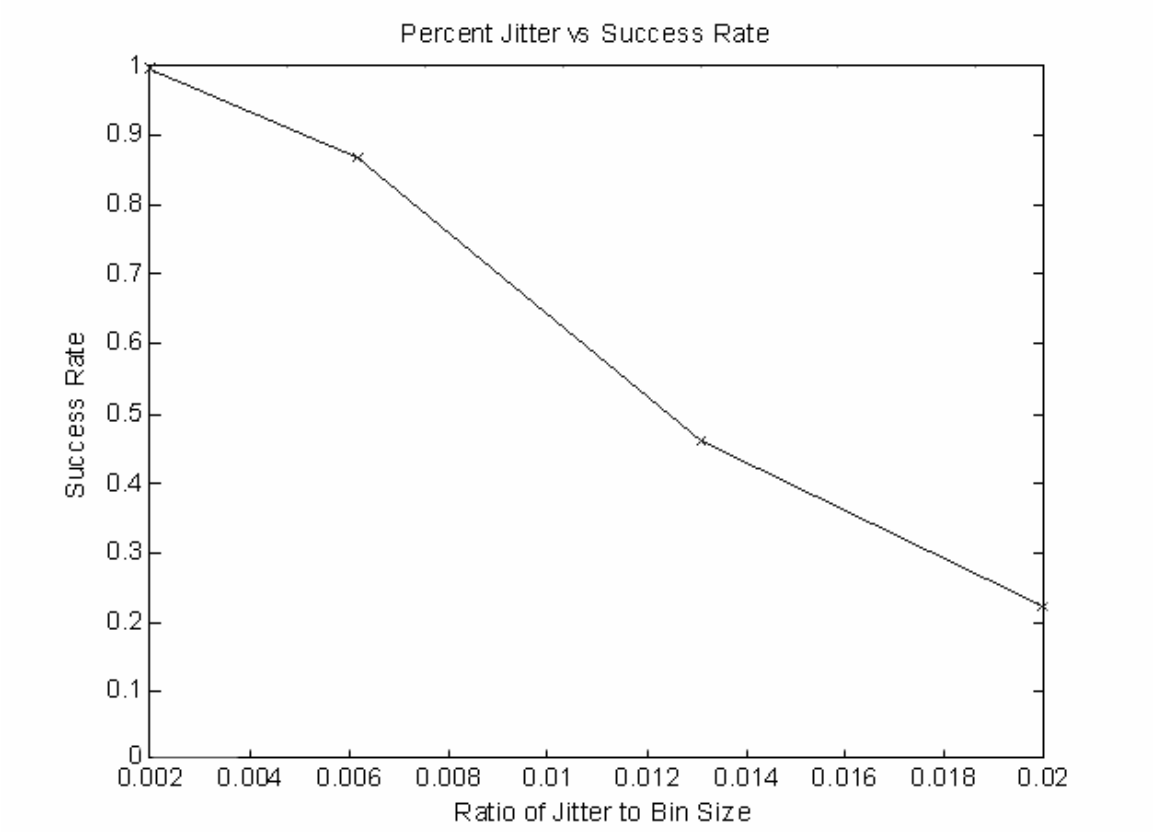

Figure 12: The success rate of the grid-based approach when jitter was added to the star positions, and the bins were not expanded to compensate. The ratio of jitter to bin size refers to the jitter added to the image relative to the size of the bins used to match the image.

2.7 Conclusions

The experimental results summarized above are very encouraging. Not only were all the images solved correctly, but the number of candidate solutions proposed by the algorithm was generally small. Furthermore, when a sufficiently large number of bins were used the results demonstrated that the grid-based approach was very reliable.

In summary, the advantages of this grid-based approach are several:

i) Interlopers and drop-outs are dealt with directly and symmetrically as part of the matching algorithm.

ii) The matching is performed using coarse features which only require two stars for orientation.

iii) We can optimize the index for space and accuracy and we can improve the solving time when information is available concerning the number of stars per image and the star brightness.

iv) By assuming that only distractors or drop-outs occur, the matching process can be done using simple bitwise operations, resulting in a fast matching process.

There are, however, two major drawbacks to this approach. First, the algorithm requires information about the field-of-view of the test images in order to build the index. If the field-of-view is not the same in the test images', the grid will be misaligned and any solutions will be incorrect. Second, even when applied to a relatively small set of stars like the 300,000 star Tycho2 sub-catalog, the resulting indexes are on the order of several hundred megabytes in size. When applied to a much larger catalog, the index files will also be much larger, and so if large disk resources are not available this approach may be prohibitively demanding. Of these two problems, the first is by far the more problematic, and so should certainly be the focus of any future work in this area.

2.8 Future Work

Based on the problems identified above, the direction for future work in this area is clear. There are several immediate areas for exploration related to image sizing. First, if sufficient information about the character of the stars in the image is available, it might be possible to estimate the scale of the image based on the distribution of similar stars in the catalog. The grid placed over the image could then be scaled so that it covers the appropriate regions of the image.

Another approach might be to index the catalog with a large number of bins and star pairs, and then generate lower resolution samples of this index. When matching, we could try using the various resolutions, and when only small discrepancies occur try to make minor adjustments to the scale of the image grid in order to find a perfect solution. In a slight modification to this scheme, we could change our binning from uniform to dynamic, where the grid fineness is based on the distance of the key stars from one another. So, for example, we might set a rule that rather than using 12×12 bins across the entire image, the grid is scaled so that twelve bins separate the pair of key stars, regardless of their actual angular distance. Both of these approaches are, however, susceptible to significant increases in space requirements.

In terms of general space requirements, it will be difficult to decrease the amount of space the index uses. One immediate option is to compress the index. The index will still eventually require decompressing however, at which point the space problem will again become relevant. Thus, there is no obvious direction for improvement in the area of space although further examination may still be warranted.

Chapter 3: A Shape-Based Approach

3.1 Motivation

As discussed in the motivation for the grid-based approach, the primary problem with using geometric shapes is the inherent explosion in search time as the number of stars in an image and the number of stars per shape increases. The grid-based method addresses this explosion by controlling the number of stars per shape: by using only two stars to define a complex shape the search space is reduced to *n* choose 2. However, the processing time required to construct the grid and test the resulting bit-strings for all the pairs can still be costly in the worst-case scenario. Another way to deal with the search explosion is to choose shapes that are easier to search for—i.e. some type of shape whose complexity does not significantly affect the time required to find it. We can accomplish this task if we accept the notion that rather than searching for arbitrary *n*-star shapes, we should constrain the shapes to a more easily found subset.

This idea of shape constraint is the basis of the next approach. In this method, we use pairs of stars to determine a narrow angular wedge. We then examine the wedge for a minimum number of stars, and record the relative distances and angles of each star from the star at the apex of the wedge. While difficulty with this approach arises when stars are missing, it nevertheless has several significant advantages. First, many stars per shape can be employed, allowing the method to be applied quickly to images with numerous stars while at the same time creating highly unique identifiers. Second, by using information about relative distances rather than absolute distances, this approach is scale invariant, and so can infer an image's field-of-view. Finally, this method employs a series of parameters that can be tuned to control the size of the index and the probability of spurious image matches.

3.2 Step 1: Indexing

We begin the indexing process with three parameters: an image angle (θ), a wedge angle (ϕ), and a star count (k). The image angle refers to the minimum field-of-view we hope to be able to solve. The wedge angle refers to the size of the region in which we shall look for shapes. Finally, the star count refers to the number of stars we will use in the shapes.

These parameters will vary depending on the number of stars we expect to find in test images, the size of the index we would like to build, and the size of the catalog we will be using. Tuning the image angle is straightforward—the smaller the image angle, the smaller the regions we examine, and so only shapes containing a small number of stars will be available for indexing. Conversely, the use of a large image angle means that many more shapes will be found for indexing and this can balloon the size of the final

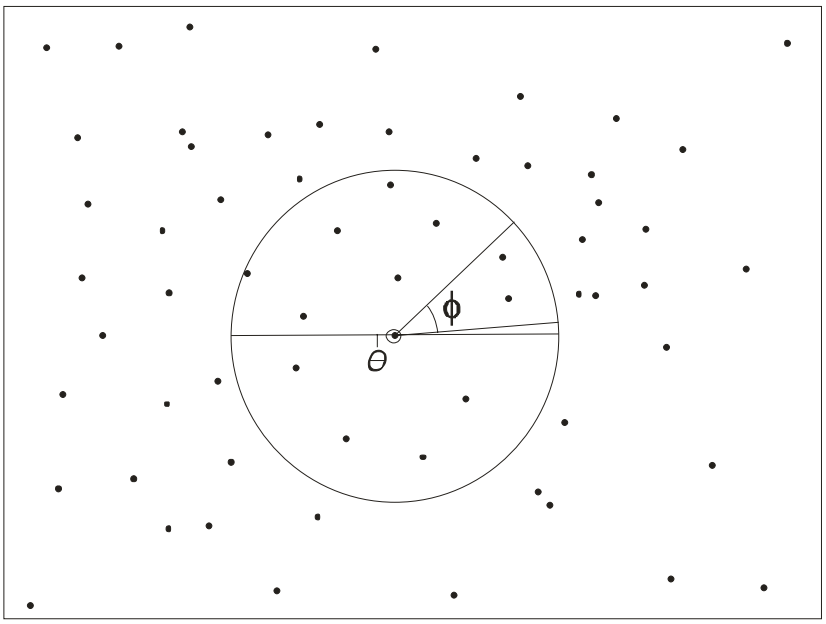

Figure 13: A circular region with diameter equal to the image angle parameter θ is used to determine a set of 'nearby' stars. A wedge is constructing according to wedge angle ϕ.

index. Tuning the wedge angle further controls how many shapes will be available for indexing. At the extremes, a 360 degree wedge is like looking for arbitrary *n*-star shapes, and a near-zero degree wedge is like looking for stars that form lines. In practice, it is preferable to use an intermediate wedge angle—too large an angle results in an explosion of the number of index-able shapes, but too small an angle makes the angular relationships almost useless. Finally, the star count impacts the number of shapes that will be found for indexing, but it also increases the differentiability of the shapes (since it is harder to produce two similar 10-star shapes than two 3-star shapes).

After setting the parameters, we begin the process of indexing the catalog. For each star, we construct a two-dimensional image centered on the star, containing all the catalog stars that fall within the diameter of the image angle parameter (see Figure 13). The image is constructed using a perspective camera model; however in practice the images are nearly orthographic due to the small image angles typically employed.

A wedge radiating from the star of interest and sized according to the wedge angle parameter is swept around the image, and is oriented using a second star (see Figure 14). That is, the second star is centered in the middle of the resulting wedge, and defines a pie piece of the two dimensional image.

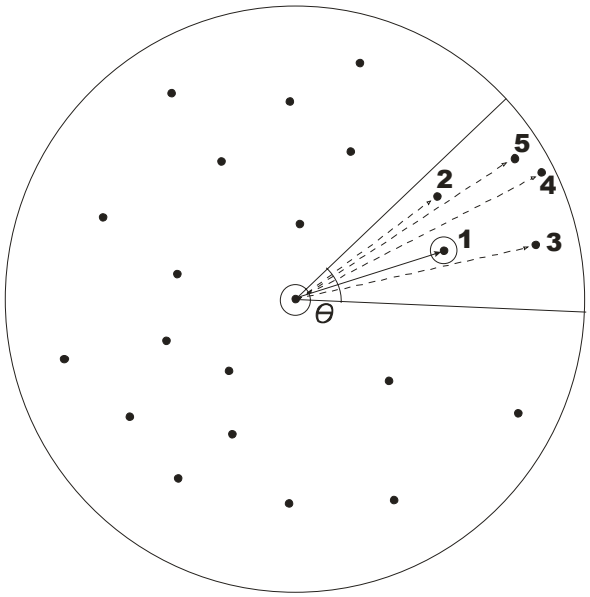

Figure 14: An image of the initial indexing step. A wedge of angle θ is rotated around a star, oriented using a second star (in this case the star labeled '1'). Note that the orienting star need not be the nearest star to the central star of interest.

The wedge is then examined to see which stars fall into it. If at least the threshold number of stars are found (*i.e*. the number specified by the star count parameter), the nearest stars that fall under the threshold count are then indexed as a single shape. It is important to note that this policy can also be varied to choose the *k*-brightest stars.

Since our final goal is to use the shapes to solve not only the test images' positions, but also their field-of-view, it is important that we use scale invariant measures to describe them. To this end, two pieces of information are recorded that completely describe the shape: the *relative* distances of all the stars in the wedge from the central star, and the angles the stars in the wedge make relative to the star used to orient the

wedge (see Figure 15). To create a relative distance for each star, and to simplify using this information during the matching process, distances are recorded relative to the distance of the farthest star to the star of interest. This means that the distance to the last star, d_k, is considered to be equal to 1.0, and the remaining distances are recorded as a ratio of lengths. Using Figure 15 as a guide, the remaining distances would be recorded as $\frac{d_1}{d_4}, \frac{d_2}{d_4}$, and $\frac{d_3}{d_4}$. This method not only provides a scale invariant measurement, but also ensures that all the distance measures fall in [0,1], with the distances approaching one as they get farther from the central star.

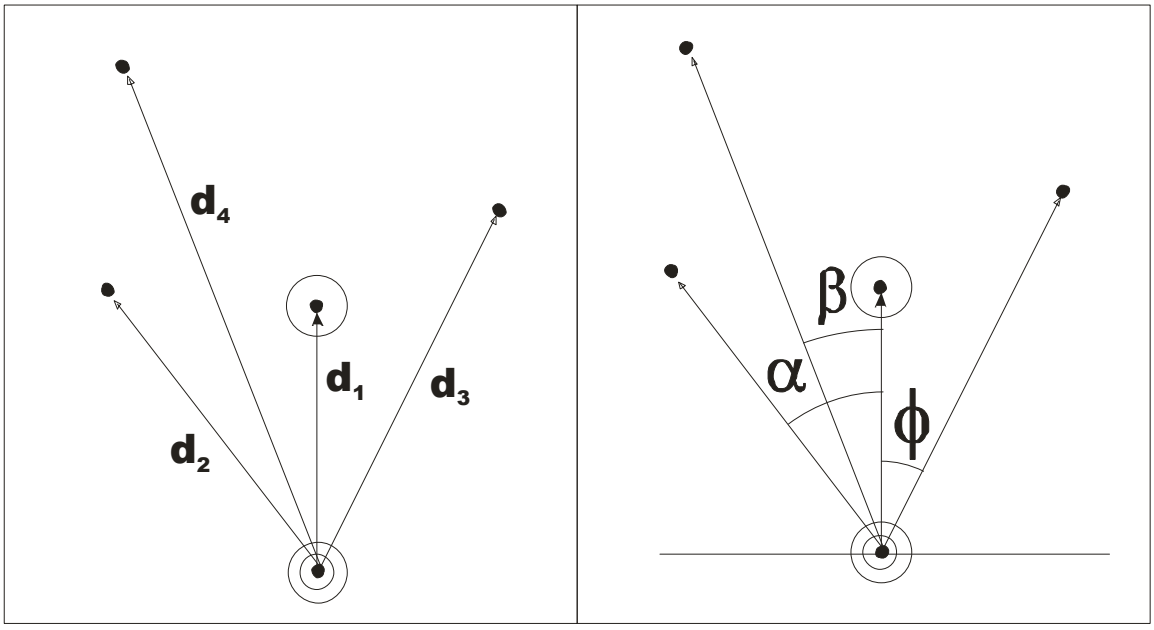

Figure 15: The two pieces of indexed information are represented. On the left, the distance of stars from the central star. On the right, the angles of the stars relative to the orienting star. In each image, the star of interested is double-circled, and the orienting star is single-circled.

Now that we have a complete list of shapes generated from the catalog, we can construct the final index. To facilitate fast lookup we use the first distance ratio, $\frac{d_1}{d_4}$, as a hash key, and organize the shapes according to their key. Thus a table is constructed

where each entry contains a list of shapes whose first distance ratio falls within a tolerance. The tolerance is set as an indexing parameter, and influences the speed of lookup and the size of the index. The smaller the tolerance, the more bins in the table and hence the larger the index, but the fewer shapes we need to look through in any particular bin. The larger the tolerance, the more shapes we may have to look through during the matching phase, but the smaller the index overhead. At lookup time, of course, we may still end up searching through several bins in the table that are close together in order to improve our changes of finding the true identify of a shape in the image. The table's lists are composed of shape identifiers that can be used to look up the complete list of distances and angles in separate tables (see Figure 16). With this information recorded to a file, the index construction is complete.

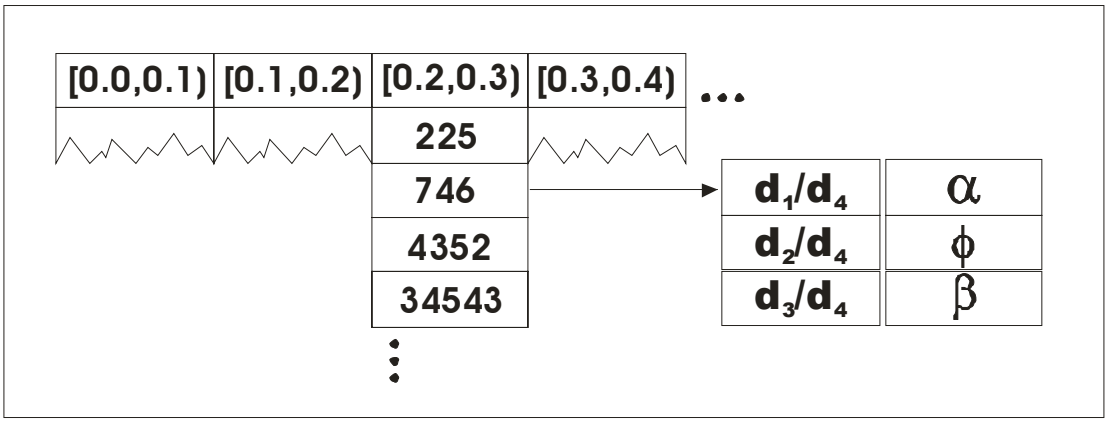

Figure 16: A diagram of the index layout. A lookup table contains shape ID's, which point to a list of distance ratios and angles, sorted from nearest to farthest from the central star of interest.

3.3 Matching

With a completed index we can begin the task of solving images. The process is simple in that it mimics the index construction process. The matcher is provided with two parameters, (besides the index, the original catalog which is used for the final orientation estimation, and the image to solve): a range of angle tolerances, and a range of distance

tolerances. During solving, we expect to see images with fields-of-view that vary greatly from the initial indexing scale. There is a high probability that at a large image scale (relative to the initial indexing scale) significant positional accuracy will be lost. To combat this, we set a range of tolerances that responds to the proximity of the stars in a wedge to the central star of interest. The closer the stars are to the central star, the larger the tolerance used. This is because when stars are closer their distance ratios and angles will be more adversely affected by small errors in position. In this work the tolerance was varied linearly as stars got farther from the central star until a cutoff point was reached (see Figure 17).

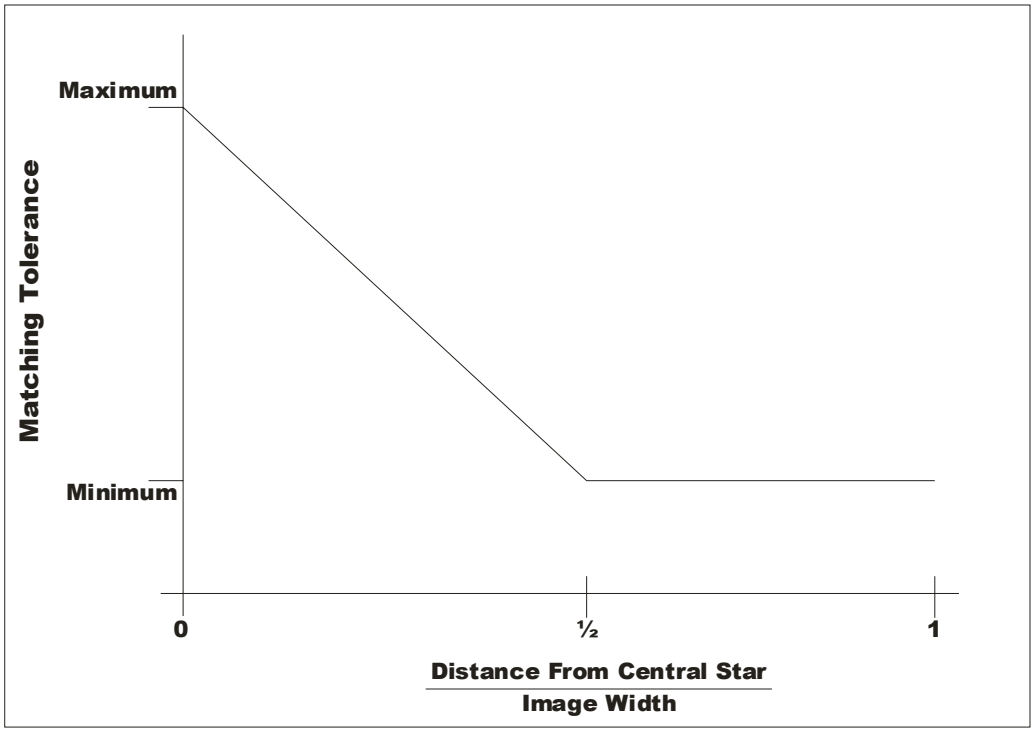

Figure 17: The change in matching tolerance as the distance to the central star increases (distance being measured relative to the width of the image).

The complete matching process proceeds as follows: each star in the image is considered one at a time. The current star is used as the central pivot for a wedge, just like in the indexing phase, and any wedges around it containing at least the threshold

number of stars are matched against the shapes in the index. When a shape match is found, the catalog is used to determine the identity of the shape's stars, and the view angle and camera orientation are solved (see Section 3.4 for a complete explanation of this final step).

There are two elements that differentiate the matching process from the indexing process however. The first element, as mentioned above, is that tolerances are used for matching the angles and distances. These tolerances are also used to expand the wedge when looking for shapes. The second element concerns interlopers. It's entirely likely that test images will contain numerous interlopers, so we cannot guarantee that we will find indexed shapes in an image simply by using the nearest stars to the central star. Instead, when we find a wedge that contains at least the threshold number of stars we need to test all the shapes in the wedge against the index. At first it may appear that we are headed for a combinatorial matching process. Fortunately, because the shapes are found using wedges, there is a strict ordering of the stars which means we don't need to try all the combinations. Instead, we loop through the stars in the wedge to choose the 'farthest' star, which becomes the basis for the relative distance ratios, and then try to match the resulting ratios and angles using the remaining stars. It may turn out that when using a particular star as the 'farthest' we still have more stars in the image shape than the index requires (as a result of interlopers, or simply because the image is taken using a large field-of-view). Again, because of the strict ordering of the stars in the wedge, when we find a star that doesn't match an indexed shape we can simply skip it and try the next image star in its place. If the indexed shape really does appear in the wedge, we will eventually find the correct solution through this process of skipping stars that do not

match. Throughout this procedure we have one more restriction that helps eliminates candidates: the 'farthest' star cannot be any star nearer to the wedge apex than the orienting star. That is, since the orienting star directs the wedge it *must* be included in the shape, and so no star nearer than the orienting star can possibly be the 'farthest' star. Because of this property, we only need to test different 'farthest' stars until we reach the orienting star, at which point we can stop.

This concludes the process of direct image solving for a particular index; however, there are still two more problems that arise with practical image matching. First, images taken using a large field-of-view (and hence having many stars) will contain a huge number of shapes with small star counts. Under these conditions, indexes that use only a few stars per shape will be slowed considerably. Second, as the field-of-view increases the absolute positional accuracy of stars in the image decreases. To counteract this we can increase the matching tolerances, but only at the risk of increasing the probability of mismatches. One solution to this second problem is to simply use more stars per shape and a larger field-of-view when indexing, which would counteract the lower accuracy per star with a higher number of values that must be matched. Unfortunately, this method means we will fail to solve an image taken with a small field-of-view. Instead, our solution is to create several indexes using a progressively larger field-of-view and more stars per shape. We can then solve images using a waterfall approach: we begin using the index with the highest order shapes and largest field-of-view. If and when that index fails to find a match, we drop down to the index with the next largest order shapes and field-of-view. This process continues until we either find a solution or run out of indexes. Naturally, this means that in the worst case solving can

take a long time. However, the advantage of this method is that it acts as a guard against the problems inherent in solving an image with a completely unknown and unestimated field of view.

3.4 Solving the Field-of-View and Orientation

Once the algorithm has matched at least three image stars with corresponding stars in the catalog, there remains the problem of determining the view angle and the orientation of the image. In the grid-based method, where we know the field-of-view right from the beginning, solving the orientation is relatively straight-forward. Here, we have no idea of the scale of the image, and so a more complex approach must be used. At this point we have several pieces of information for use in the solving process:

i) A known correspondence between at least three 2D image stars and 3D catalog stars.

ii) The angles between the 3D catalog stars.

iii) The assumption that the camera projection is in the center of the image.

iv) The assumption that the pixel aspect ratio is 1.0.

Points iii) and iv) serve to simplify the details of solving, while i) and ii) provide us with all the information we require for our calculations. Our first task is to determine the 3D positions of the image stars relative to the camera, which can then be used to determine the camera's orientation and field of view. Our problem is that the 2D positions in the original image, when taken using a standard camera, do not scale directly to the corresponding *x/y*-coordinates in camera space. In practice, as mentioned before, when dealing with a small field-of-view (like in star images) the image can be approximated as an orthographic projection (meaning that the 2D positions *do* scale to

the *x/y*-camera coordinates); however, for technical correctness we will still solve the complete perspective problem.

In order to find the positions of the stars in camera space, we want to determine the direction vector of the image stars using pixel units, and then convert these values into camera-coordinates. The important value to determine, since we have the direction vectors' *x/y*-coordinates in pixel units, is the corresponding *z*-coordinates in pixel units. Fortunately, the *z*-coordinates are the same for each vector, as can be seen in Figure 18. This is the piece of information that allows us to leverage the angle, θ, between the corresponding catalog stars and subsequently solve for the camera coordinates.

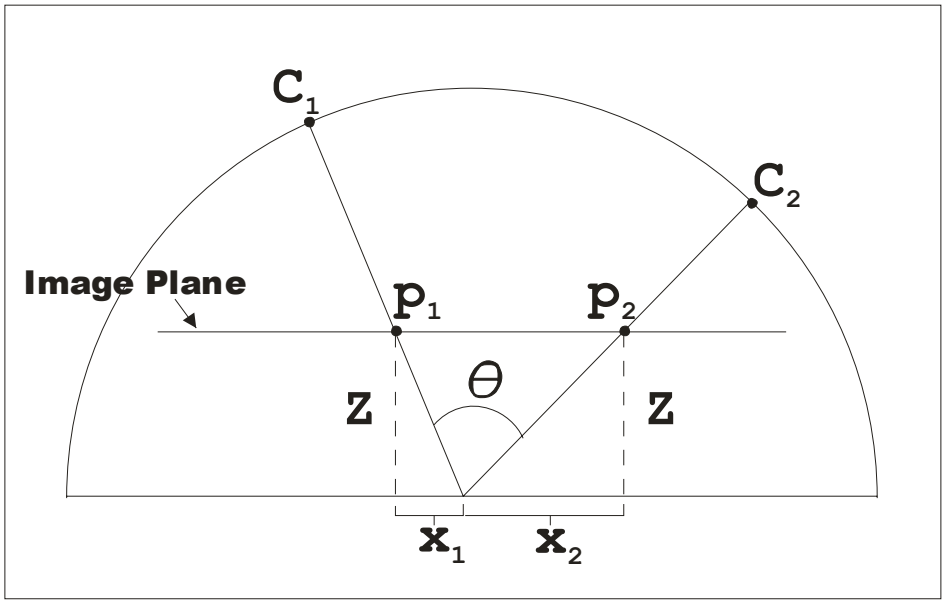

Figure 18: A two dimensional representation of the geometric relationship between the camera image and the actual positions of the stars in camera space. Points c_1 and c_2 are the stars in camera space, and p_1 and p_2 are the stars as they appear in the image plane. Values x_1 and x_2 are the distances of the points p_1 and p_2 from the center of the image. We can use the angle θ (calculated from the coordinates of the catalog stars corresponding to p_1 and p_2) to find z. The important point to note is that z is the same for both p_1 and p_2. The three dimensional version of this image is analogous: it simply requires using the additional values y_1 and y_2.

Since the angle between the stars in world coordinates is the same in camera coordinates, we can use the cosine rule to determine the *z*-coordinate according to:

$$\begin{aligned} Cos^2\theta &= \frac{(p_1 \bullet p_2)^2}{\|p_1\|^2\|p_2\|^2} \\ &= \frac{(x_1x_2 + y_1y_2 + z^2)^2}{(x_1^2 + y_1^2 + z^2)(x_2^2 + y_2^2 + z^2)} \end{aligned}$$

Solving for z we get:

$$z = \frac{1}{\sqrt{2}}\sqrt{\frac{-1}{(Cos^2\theta - 1)}\left(-2a + (b+c)Cos^2\theta + \sqrt{-4(Cos^2\theta - 1)(-a^2 + bcCos^2\theta) + (-2a + (b+c)Cos^2\theta)^2}\right)}$$

Where:

$$\begin{aligned} a &= x_1x_2 + y_1y_2 \\ b &= x_1^2 + y_1^2 \\ c &= x_2^2 + y_2^2 \end{aligned}$$

Substituting *z* into each of the points as their respective *z*-coordinate we have constructed each star's position vector—*i.e*. the vector along which the light from the star travels into the camera, passing through the image plane. The true positions of the stars relative to the camera are merely the normalized unit vector versions of the pixel unit vectors.

Now that we have the stars' positions in camera coordinates and world coordinates, it is a simple matter to reconstruct the camera basis from which the image was generated. Where we have three points w_1, w_2, and w_3 in world coordinates, and the corresponding three points c_1, c_2, and c_3 in camera coordinates, each basis vector can be determined by solving a linear system of the form $Ax = b$. So for example, for the camera basis$[\vec{i}\ \vec{j}\ \vec{k}]$, basis vector $\vec{i}$ can be found by solving:

$$\begin{bmatrix} w_{X1} & w_{Y1} & w_{Z1} \\ w_{X2} & w_{Y2} & w_{Z2} \\ w_{X3} & w_{Y3} & w_{Z3} \end{bmatrix} \begin{bmatrix} i_X \\ i_Y \\ i_Z \end{bmatrix} = \begin{bmatrix} c_{X1} \\ c_{X2} \\ c_{X3} \end{bmatrix}$$

The remaining two vectors can be determined in an analogous fashion.

In the final step, the viewing angle can be calculated using the distance to the camera in pixel units and the width of the viewing plane in pixel units. So, if the width in pixels is V, and the viewing plane distance in pixels is z as before, the viewing angle is:

$$ViewAngle = 2\,ArcTan\left(\frac{0.5V}{z}\right)$$

It is important to note that in the procedure outlined above only three stars have been used to solve the orientation and view angle. It is a requirement that three stars be identified in order to solve the orientation problem; however, *more* than three identified stars might available. In this case a similar but more robust approach can be taken. First, the distance to the image plane can be calculated for each pair of stars and an average taken among the results. Second, the camera basis can be found using SVD, thus providing a least-squares solution. The camera basis is found as follows: first matrices A and B are formed with rows containing the world coordinates and corresponding camera coordinates respectively. Next, the matrix $C = B^T A$ is calculated, and the matrices U and V determined via $SVD(C) = [U, S, V]$. The rotation matrix, with its rows consisting of the basis vectors $\vec{i}$, $\vec{j}$, and $\vec{k}$ in order, is then $R = UV^T$. A complete reference to this result, referred to as the orthogonal Procrustes problem, can be found in [14].

As a final step, the resulting camera basis also has properties that allow us to avoid false matches. The resulting matrix, since it is a rotation matrix, should contain normalized rows. If these rows are not properly normalized then the rotation matrix is invalid and indicates that the solution can be disregarded.

3.4 Predicting the Index Size

One more area of interest is predicting how many shape we should expect to index for a given set of parameters. Using this information we can tune the indexing phase to construct a single powerful index, or a construct a complimentary set of indexes to use in conjunction. Here, we describe the calculations required for the prediction. See Section 3.6 for further information on how these values vary as the parameters are changed.

Operating under the assumption of a uniformly distributed star catalog, we first need to know the surface area of the sphere that will be examined around each star. Using spherical coordinates, where r is one half of the maximum field-of-view, the surface area of the sphere is:

$$\begin{aligned} A &= \int_0^{2\pi} \int_{\pi-r}^{\pi} Sin(\phi)\, \partial\phi\, \partial\theta \\ &= 2\pi\left(-Cos(\pi) + Cos(\pi - r)\right) \end{aligned}$$

The area marked out by a wedge is proportional to the angle of the wedge, so where the angle of the wedge is w, the surface area of the wedge, W, is:

$$W = \frac{w}{2\pi} A$$

From this information we determine the proportion of the sphere's surface marked out by a single wedge, P:

$$P = \frac{W}{4\pi}$$

Next we need to know how many wedges we expect to examine over the entire indexing process. For any individual star the number of stars we expect to see around it,

E, (and hence the number of wedges examined), where N is the number of stars in the catalog, is:

$$E = \frac{A}{4\pi}(N-1)$$

Thus the total number of wedges checked over the entire catalog, C, is $C = NE$.

The probability of the number of stars in a wedge is a Poisson random variable with parameter $\lambda = NP$. As a result the probability of finding a shape in any particular wedge, L, where M is the number of points required per shape is:

$$\begin{aligned} L &= Poisson(\text{stars in wedge} \geq M-1, \lambda) \\ &= \sum_{x=M-1}^{\infty} Poisson(x, \lambda) \\ &= 1 - \sum_{x=0}^{M-2} Poisson(x, \lambda) \end{aligned}$$

Note that we use the threshold *M-1* rather than *M*. This is because in order to examine a wedge it is a given that there is already one star in the wedge, so rather than guaranteeing that *M* stars are in the wedge, we need only guarantee *M-1 more* stars are in the wedge.

From these equations we find the expected number of shapes produced from the catalog, X, is calculated as $X = LC$. The final size of the index can be calculated using the expected number of shapes multiplied by the required space to store each shape. See section 3.6.3 for results regarding the accuracy of these predications.

3.5 Running Time

With the indexing and matching algorithms explained, the question remains as to how efficient these processes are. With that in mind, we break down the running time for the indexing and matching steps.

3.5.1 Indexing

During indexing, for each star we construct an image containing all the stars within a certain angular distance. Once the image is generated, each star in the image is used to orient a wedge, and we check to see which stars fall within the wedge in order to look for shapes. If enough stars fall into the wedge the distances and angles are sorted and stored. Wedge generation requires time $O(n)$, and building shapes from each wedge requires time $O(n\log(n))$, meaning that the shape finding process requires time $O(n^2\log(n))$. Since this shape finding process is repeated for every star, the complete indexing process therefore requires time $O(n^3\log(n))$, where n is the number of stars in the catalog.

3.5.2 Matching

During matching we use every pair of stars in the image to construct wedges, requiring time $O(m^2)$, where m is the number of stars in the image. Once a wedge with sufficiently many stars has been found, we have to use various orderings of the stars within the wedge to look for shapes in the index, which in the worst case scenario requires time $O(m)$. Each time we try a different star as the 'farthest' star, we compare the new shape in the wedge against shapes in the index, which requires linear time for each shape checked and in the absolute worst case requires checking every single shape in the index, resulting in time $O(ms)$, where s is the number of shapes in the index. The complete matching process therefore requires time $O(m^4 s)$, where m is the number of stars in the image and s is the number of shapes in the index. It is interesting to compare this result with the time required to match arbitrary 4-star shapes. In the 4-star case we

have to find every quadruple, requiring time $O(m^4)$. For each quadruple we try to match 4-star shapes in the index, which at worst requires time $O(s)$. The total match time therefore is $O(m^4 s)$, where m is the number of stars in the image, and s is the number of shapes in the index. So, the shape-based approach described in this work requires time proportional to matching 4-star shapes, but allows us to employ any-star shapes instead.

3.6 Experimental Results

In this section we explore the effectiveness of the shaped-based approach. To do this an image indexer and solver were written and tested against a set of synthetic images generated from an artificial catalog. The catalog was composed of 1 million stars generated using a uniform distribution. Random images were generated using a field-of-view ranging from 0.5 to 2.5 degrees. Random distracting stars were also added to ensure a mean of 100 stars across all the images. Figure 19 contains a histogram of the number of catalog stars contained in the test images.

Six indexes were built using an increasing field-of-view and an increasing number of stars per shape. The field-of-view in each index varied so that 6, 10, 20, 40, 60, and 80 stars were examined (on average) around each star during the indexing. Each index used shapes containing 4, 4, 6, 9, 13, and 16 stars respectively. In addition, the following wedge sizes were used: 190°, 40°, 28.5°, 24°, 28°, and 28°. These parameters were chosen to ensure that on average 30 shapes would be at least partially visible in any image containing 6, 10, 20, 40, 60, and 80 stars. The size of the resulting indexes was 282, 171, 134, 105, 106, and 96 megabytes respectively. The motivation for producing

this particular set of indexes was to ensure that there was a very good chance a completely indexed and discernable shape would appear in every test image.

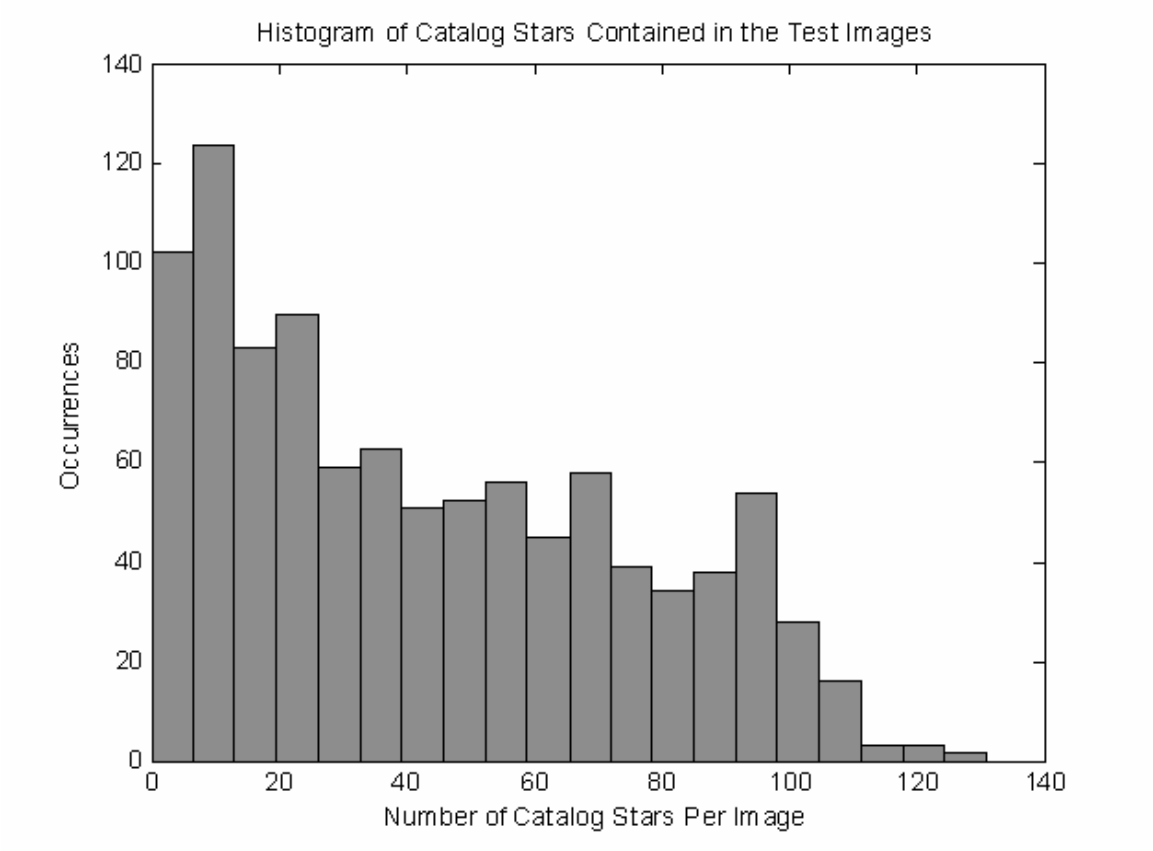

Figure 19: The distribution of the number of catalog stars in each test image. Notice that many of the images contained fewer than 10 catalog stars, which makes solving potentially quite difficult.

3.6.1 Experimental Success

In summary, the experiments were very successful. Using the combined waterfall approach described in Section 3.3, ninety-two percent of the images were solved correctly (see Figure 20 below), and the images that were not solved contained very few catalog stars (*i.e.* <18). It is important to understand that even though two of the indexes were built using 6 and 10 stars (on average) to find shapes, we could not have expected these indexes to consistently solve images with less than 18 stars. First, the indexes were built with averages in mind, and so it was still entirely possible to generate images where

no indexed shapes were visible (this issue is discussed further in Section 3.7). Second, when the indexes were created, shapes were found around a star sitting in the exact center of an image. Actual images cannot be relied upon to contain a star in the center, and this decreases the likelihood of complete shapes being visible in a small field-of-view. For good results we have found empirically that an image should contain at least twice the number of stars that the index used to find shapes, and three times the number of stars to perform perfectly. This idea is borne out by Figure 20, showing that each index usually

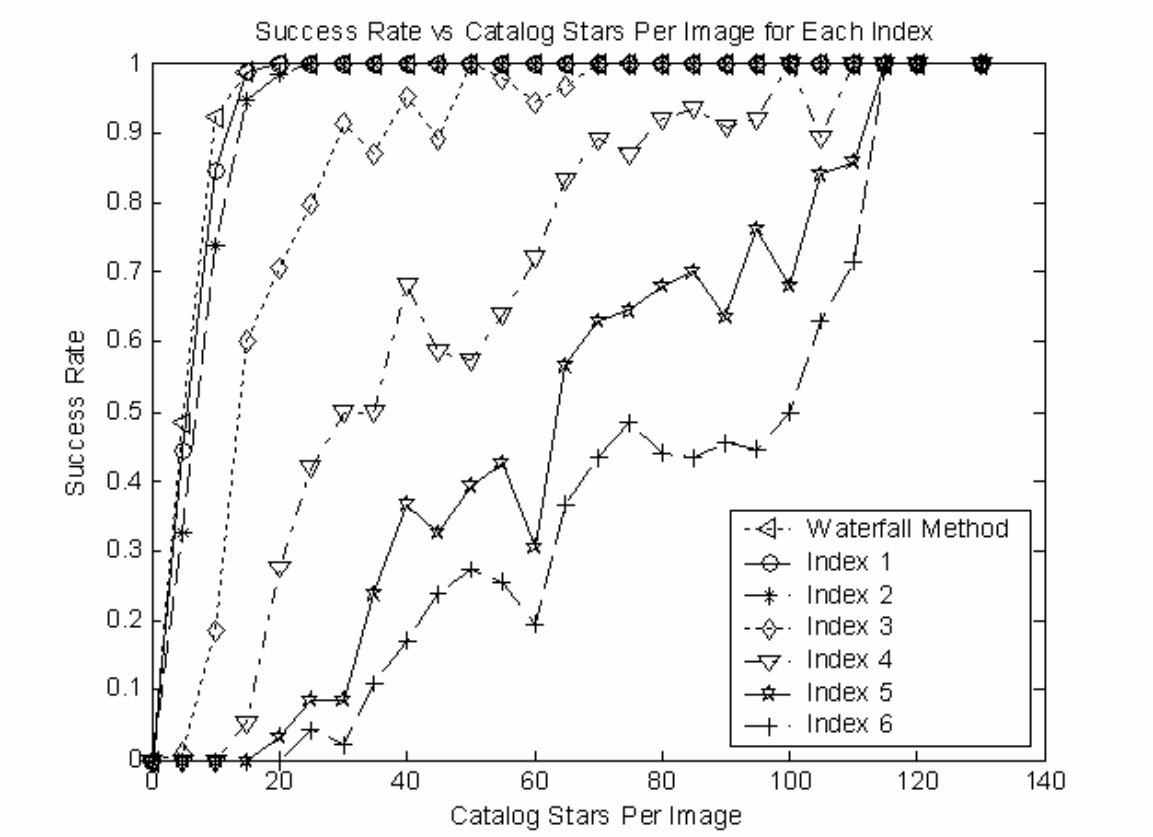

Figure 20: A plot of the change in success rate for different numbers of catalog stars per image. The numbered indexes refer to the indexes described to at the beginning of Section 3.6. The waterfall approach refers to the process of using all the indexes together.

started to solve images perfectly when those images contained between two to three times the number used during indexing. As a side note, the complete waterfall approach resulted in a success rate nearly identical to the index built using the smallest field-of-

view. This is not surprising since the synthetic images contained very accurate star positions, and so the loss of positional accuracy in larger field-of–view images, for which the waterfall approach is designed, was not significant enough to have a large effect.

3.6.2 Experimental Success with Positional Jitter

Now that we know the algorithm can solve accurately measured images, we can look at images with positional noise. Experiments were run is the same fashion as before, with Gaussian positional noise added to the star positions. Figure 21 (next page) shows the results of these experiments. In summary, even with positional noise the combined waterfall approach had a success rate of 85%. A comparison of Figure 20 and Figure 21 shows that every index performed somewhat worse when noise was added, but this is to be expected. Notice that indexes 1 and 2, which were created using a small field-of-view, became less successful as the number of catalog stars per image, (and accordingly the field-of-view) got larger. This fact confirms the usefulness of the waterfall approach at solving images over a variety of view angles—with a large field-of-view small positional noise will have an increasing effect on indexes built using a small field-of-view, thus justifying the use of a set of indexes at various sizes.

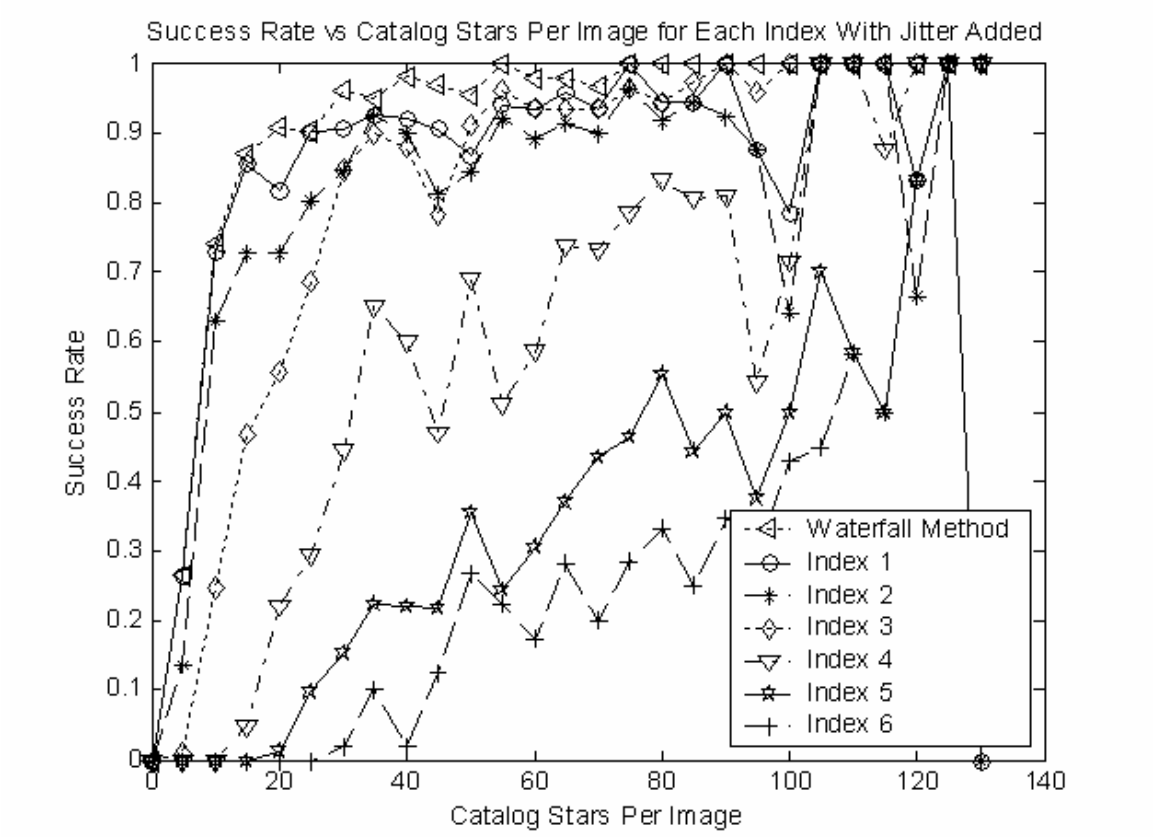

Figure 21: A plot of the change in success rate for different numbers of catalog stars per image, when positional jitter was added. The numbered indexes refer to the indexes described to at the beginning of Section 3.6. The waterfall approach refers to the process of using all the indexes together.

3.6.3 The Effect of Drop-Outs

The experimental results above all used distracting stars, but never dropped stars out. Naturally, because the algorithm employs strict shapes it is not specifically designed to solve the situation where drop-outs occur. It is still interesting to know however, how effective the algorithm will be in those situations. To this end an index was built on a catalog containing 10,000 random stars. The catalog was constructed so that in the subsequent tests 30 shapes would be at least partially visible in the test images. Experiments were then run to test the success rate when progressively larger numbers of catalog stars were randomly removed. Figure 22 summarizes the results, and shows that

the solver was still almost 50% successful when 40% of the stars in the image were removed.

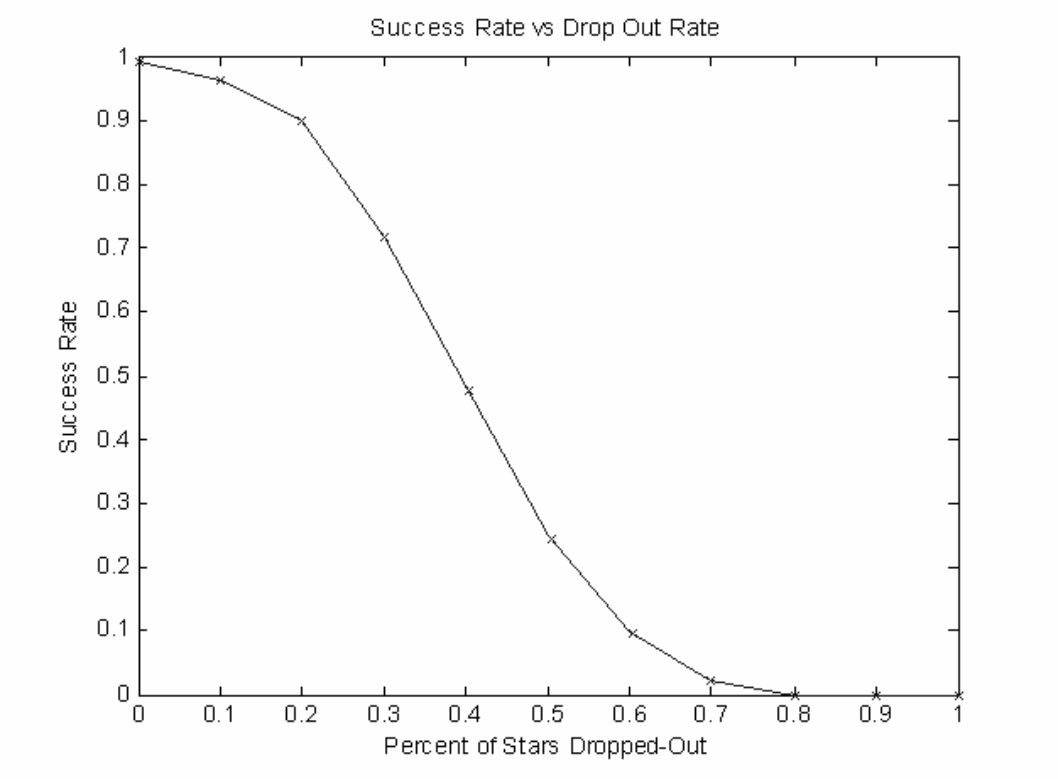

Figure 22: A plot of the success rate of the solver as more and more catalog stars were dropped out of the test images.

3.6.4 Parameter Effects

The qualitative effects of varying the search radius, wedge angle, and stars per shape during the indexing process were described in Section 3.2. Here we directly demonstrate the quantitative effects.

As discussed previously, increasing the number of stars per shape will decrease the number of shapes discovered during indexing. Figure 23 summarizes the result of varying this parameter while maintaining a constant field-of-view and wedge angle. The index was built using a large enough field-of-view to ensure that fifty stars were examined around each star, on average. The first important feature of the plot is that the

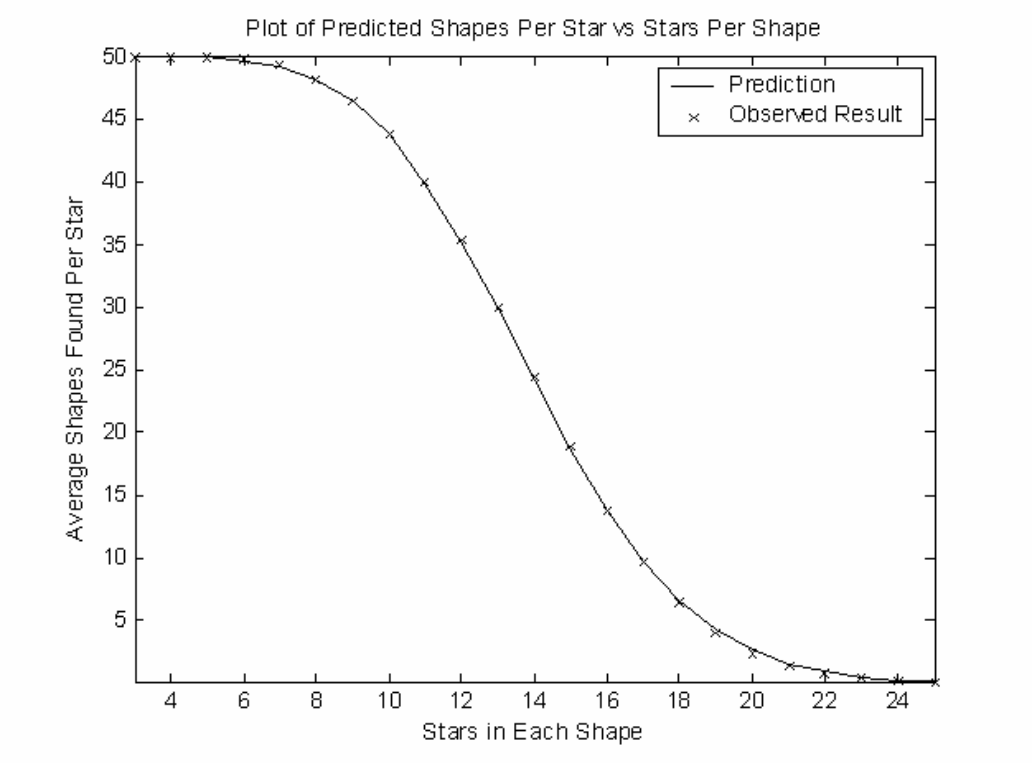

Figure 23: A plot showing how the number of shapes found per star varies with the number of stars required in a shape.

predicted number of shapes, (calculating using the equations in Section 3.4), was very accurate. We can also see a critical section where the number of shapes found drops sharply as the number of stars per shape increases. It is important to keep in mind, however, that even when the number of shapes per star is close to zero, in a large catalog this can still amount to a large number of indexed shapes. Not surprisingly, a similar but inverted relationship occurs when the size of the wedge angle used to look for stars is varied (see Figure 24). Here too, the equations in Section 3.4 prove to be accurate predictors of the number of shapes found per star.

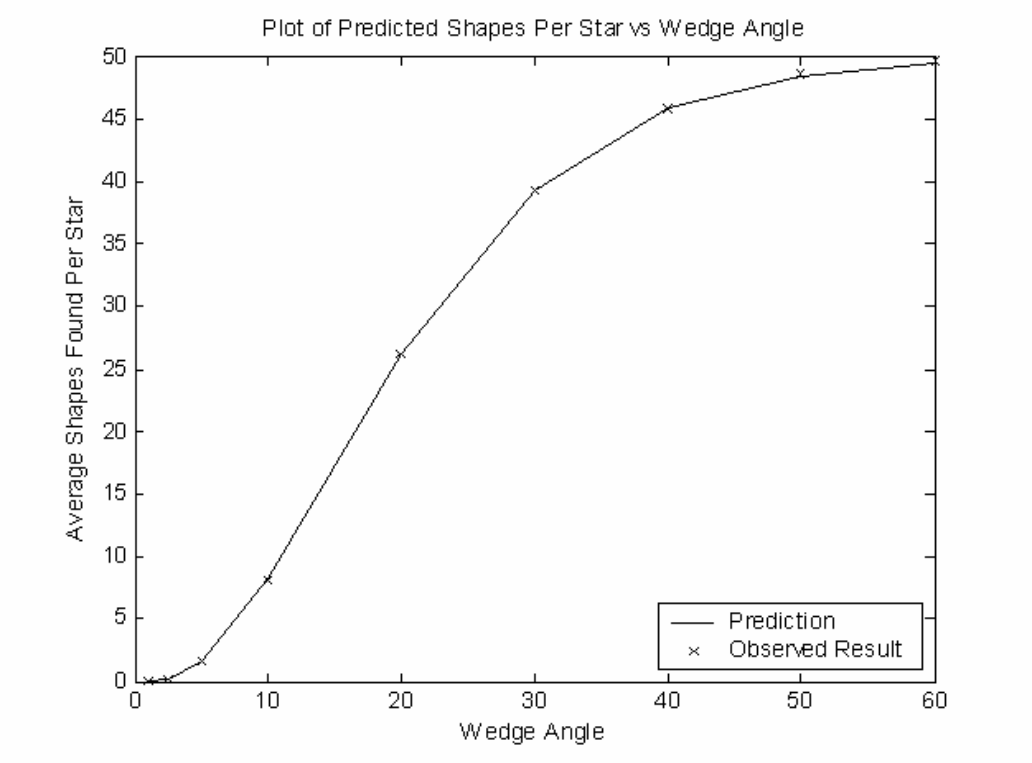

Figure 24: A plot showing how the number of shapes found per star varies with the size of the wedge angle used to look for shapes.

A related question is how well the lookup table for distance ratios works. Ideally, we would like a uniform distribution of shapes associated with each entry in the table. Figures 25 through 28 show the reality: the distribution of the ratio used for the table lookup is peaked, and shifts towards zero as the number of stars per shape increases. This suggests that a different method of hashing the shapes is warranted.

One solution that comes to mind is to dynamically resize the bins in the table to create a uniform distribution. Unfortunately, the matching tolerances invalidate this approach since the tolerances will still force us to check the same set of shapes regardless of how the bins are sized. Another solution to the lookup problem might be to use the angles rather than the distance ratios. Figure 29 shows the distribution of these angles. Notice the strong spike near zero, which is a result of the star that directs the center of the

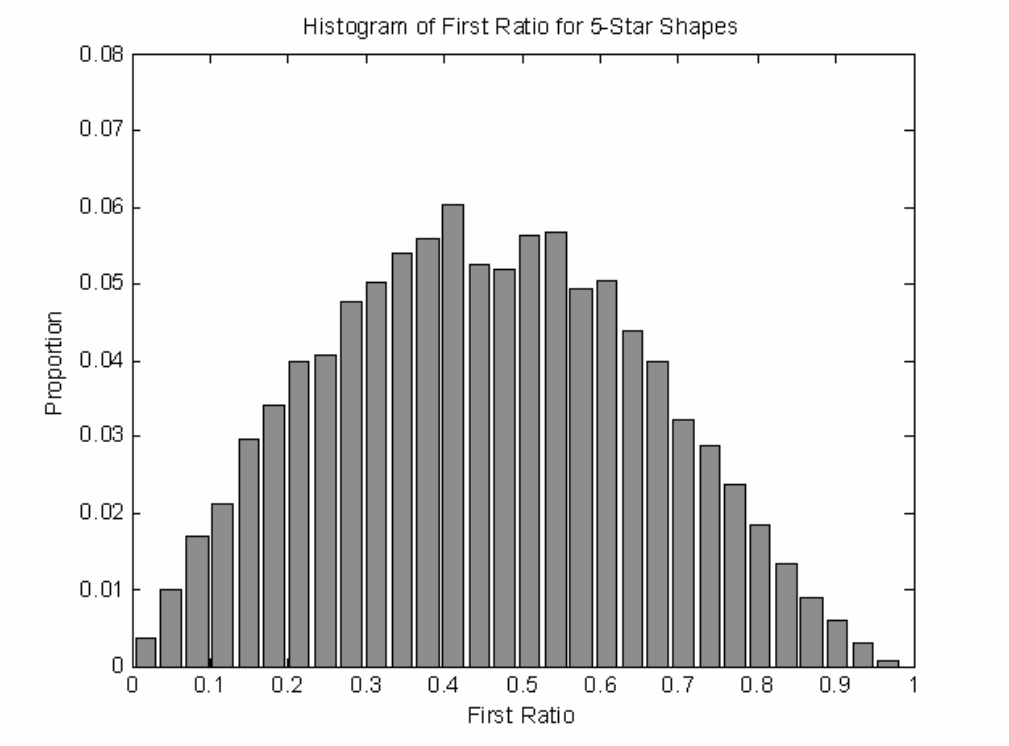

Figure 25: The distribution of the first ratio (the ratio used in table lookup) when 5-stars per shape were used.

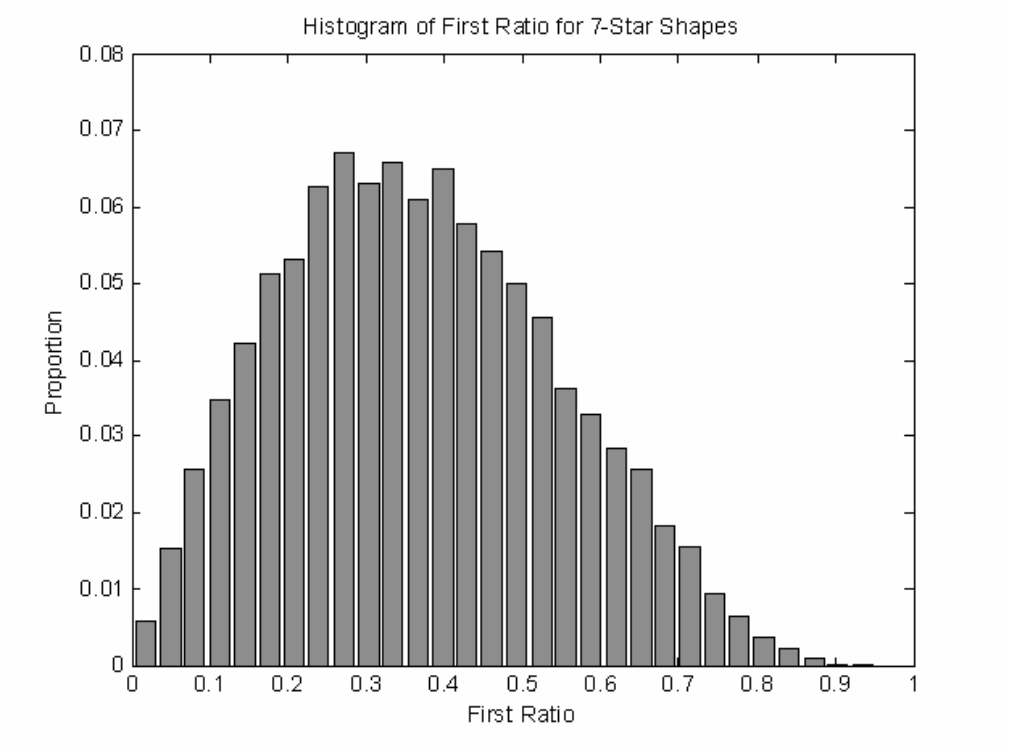

Figure 26: The distribution of the first ratio when 7-stars per shape were used.

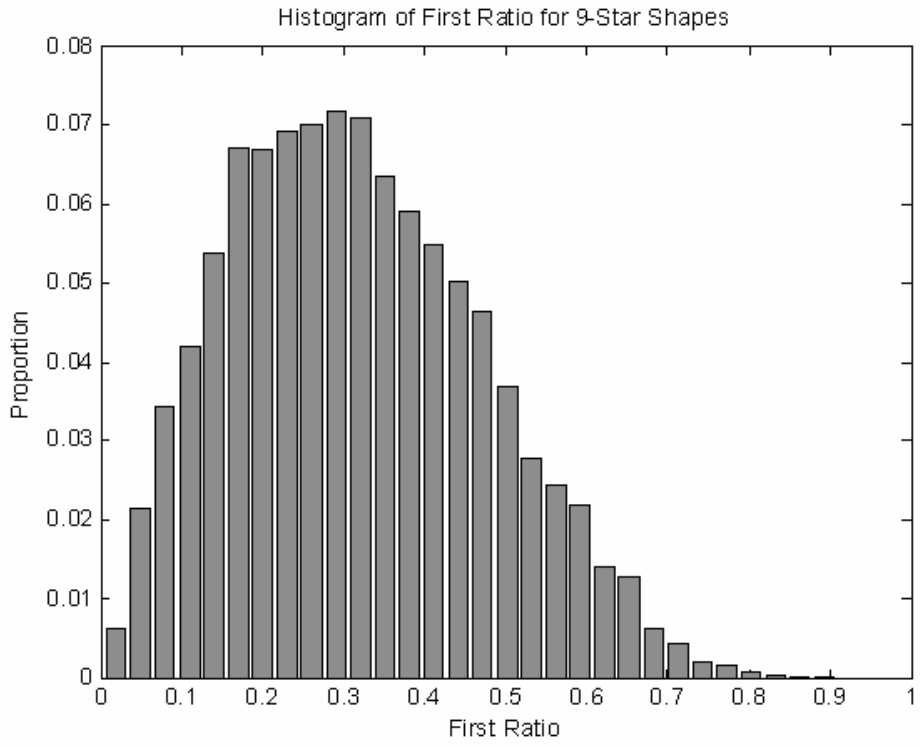

Figure 27: The distribution of the first ratio when 9-stars per shape were used.

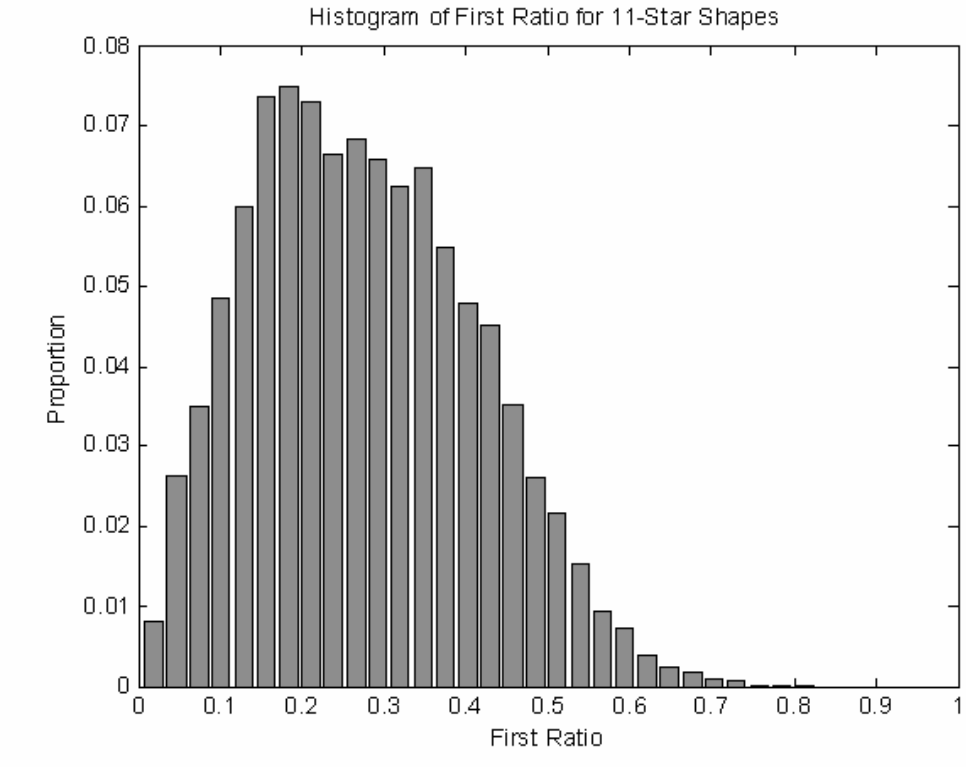

Figure 28: The distribution of the first ratio when 11-star s per shape were used.

wedge (and is therefore guaranteed to be zero—recall Figure 15 in Section 3.2) being saved in the index. By using the first non-zero angle we could employ a uniform distribution to lookup shapes in the index. However, if we employ an angle the range of lookup values will be limited by the angular size of the wedges used to generate the index.

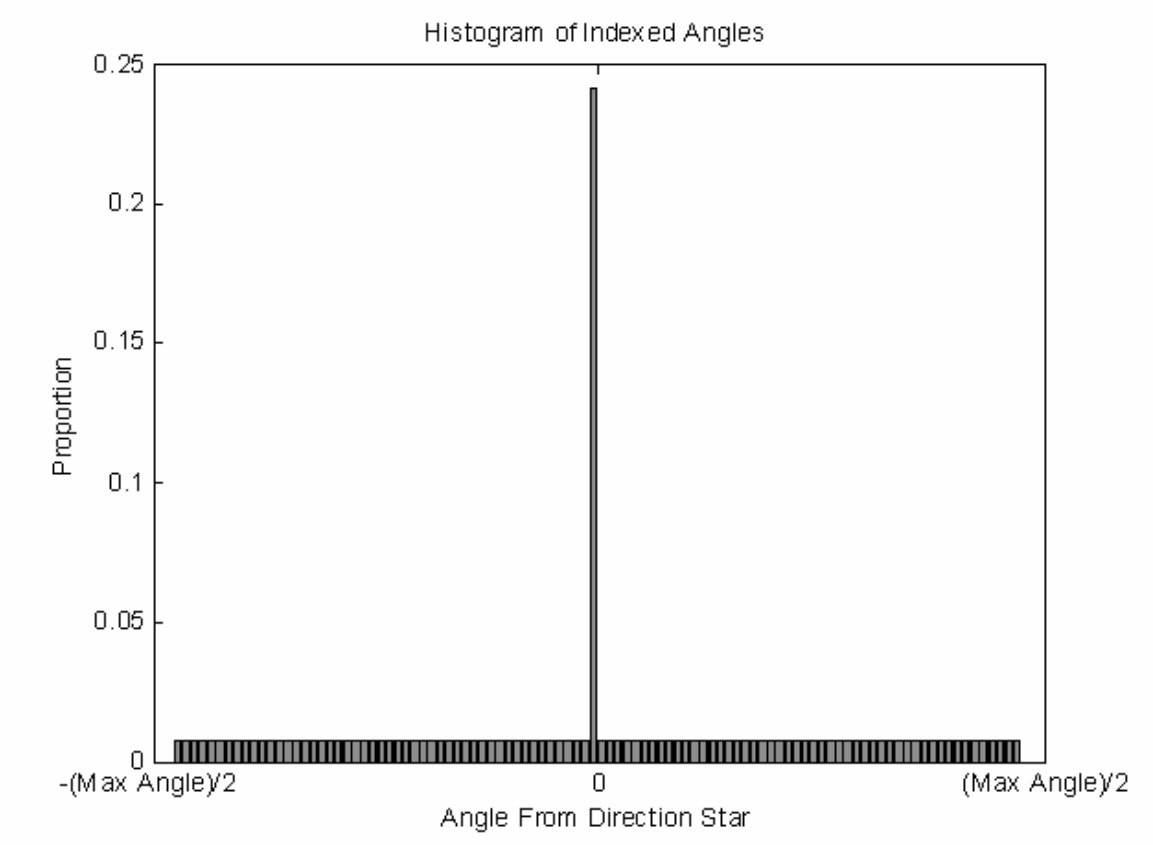

Figure 29: The distribution of angles among the stars in the indexes. The spike at zero is to be expected since the angle of the star used to orient the wedge is used in the index, and is always zero.

3.6.4 A Final Test

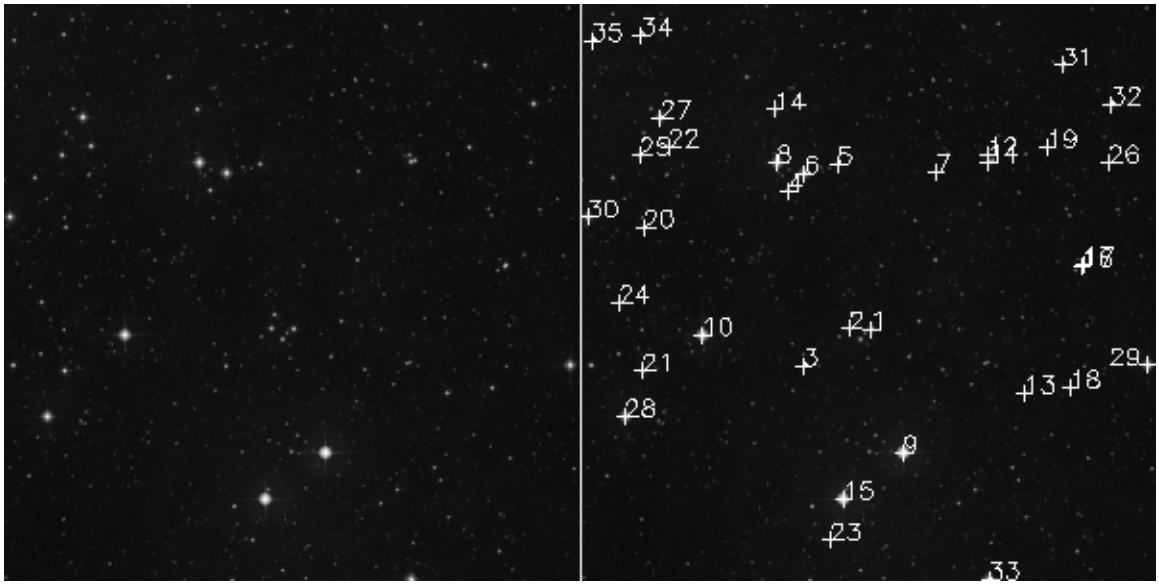

Figure 30: On the left, the raw image used for testing. On the right, the image with the brightest 35 stars marked.

As a final test of the applicability of the shape-based technique we attempted to solve a real sky image. The image was obtained from the SkyView Virtual Observatory [19] (see Figure 30). Specifically, it was generated using the Digitized Sky Survey, via orthographic projection over a 1×1 degree area at 0-degrees right ascension and 0-degrees declination. After obtaining the image the brightest star positions were then marked by hand. The Tycho2 catalog was then indexed and used to solve the image. Despite the fact that the stars were manually marked, and hence their positions were much less accurate than could be expected from an automated process, the correct position and field-of-view of the image were found.

3.7 Conclusions

The experimental results discussed above are very encouraging. Despite little optimization to ensure shapes are indexed across the entire sky, the set of indexes built

using heuristically set parameters performed very well. Although the success rate was not perfect, the algorithm managed to consistently solve both the orientation and the field-of-view for the test images.

To summarize then, the advantages of this approach are as follows:

i) The method allows us to use high order shapes with much faster search times.

ii) The system is robust to different fields-of-view, and can employ a set of different indexes to improve the process of solving images at different scales.

iii) The algorithm is resistant to interloping stars, and only requires extra processing to ensure that the true solution is always found when a complete shape is present in an image. Spurious matches caused by interlopers can be avoided by increasing the order of the shapes used for matching.

iv) The number of shapes found by the indexer can be predicted, and can therefore be tuned to control index size and performance.

There remain several issues with this approach. First, it is difficult to solve images containing a very small number of stars. Second, there is no direct solution for dealing with stars that drop out of the image—in fact, a single well-chosen drop-out can cause significant problems for the algorithm. Third, without sufficiently accurate positional measurements, the system is forced to use a large tolerance which results in it examining a significant proportion of the shapes in the index. Essentially, under the worst conditions the approach cannot quickly eliminate potential matches, and so must painstakingly examine many possibilities. Last but not least, the algorithm as it stands does not guarantee that any image taken of the catalog will contain a completely visible and

indexed shape. These drawbacks aside, the algorithm certainly demonstrates great potential for high performance practical use.

3.8 Future Work

The remarks in the Section 3.6 and 3.7 lay out a clear path for future work on this approach. Ensuring that every image above a certain size contains an indexed shape should be the first priority of any future work. This will certainly require tighter integration of multiple indexes rather than the *ad hoc* waterfall approach described above. The key hurdle here is ensuring that we always end up with one or more lookup tables that can provide fast elimination of improbable solutions.

There are myriad of smaller issues to address as well. A better designed approach to adjusting the matching tolerance than the current linear method should be forthcoming. When constructing shapes, it is probably better to use the brightest stars in the wedge rather than the nearest stars in the wedge. A complete solution for deciding on the indexing parameters, especially in the context where multiple indexes are to be used in conjunction, is another immediate goal. In addition, a more equally distributed lookup value which also retains scale invariance is wanting. On this theme, it would be nice to have a complex lookup value based on the several distances and angles—one more robust to positional noise that could be used to eliminate most candidates in a single stroke. Perhaps the work in [3], which employs the SVD of the direction vectors of stars, could be modified and added to our approach. Finally, from an efficiency perspective it would be preferable to more tightly integrate several indexes so that processing done to match shapes to one index could be reused and applied to others.

Lastly, dealing with drop-out stars is another major goal. Under the current system the only approach is to ensure that numerous shapes appear in every image and then hope that one of them is found. In a perfect world we would have a more robust matching algorithm; one willing to accept fewer stars than the shapes were built with, just as long as the remaining stars very closely match an indexed shape. A straightforward implementation of this idea is not hard to put together, but it fails miserably if either the star used for wedge orientation, or the star used to calculate the relative distances, are not present. Thus, a subtle approach is required if this improvement is to be robust.

Chapter 4: Conclusions

4.1 Final Comparison of Methods

With the grid-based and shaped-based methods described and explored, it may be informative to compare the relative worth of each as an identification method. The grid-based method has several nice advantages over the shape-based approach: first, it is specifically designed to handle the case of drop-out stars—something the shape-based approach is not especially robust to. Second, given the right starting pair, the grid-based approach's matching times are dependent on the number of bins in the test image, which is a function of the indexing process. The shape-based approach, on the other hand, must still work through various combinations of shapes even when the correct wedge has been found. On the other hand, the shape-based approach is well-suited to solving images at various scales—something the grid-based approach is currently incapable of addressing. Furthermore, in the experiments performed in this work the shape-based approach

required only 2/3 the space of the grid-based approach despite no effort to minimize the number of shapes indexed, and despite being used on a catalog more than three times larger. It seems that the shape-based approach has the advantage. In addition to being generally more robust, the improvements outlined for the shape-based approach are much easier to implement than those for the grid-based method. In summary then, it appears the shape-based approach is the more promising of the two. It is the opinion of the author, however, that the value and potential of the grid-based approach, and the more general concept of using coarse-grained features for indexing, should not be dismissed out of hand.

Appendix A: Pseudo-Code

Below, pseudo-code for the indexing and matching algorithms for both the grid-based and shape-based methods is presented. The code is intended to present an overview of the structure of the algorithms, rather than a line-by-line breakdown of how the code was actually written. To this end, an effort has been made to use as few variables as possible, to avoid complex array indexing, and to ignore the specifics of any data structure management. In addition, the details of various processes, such as generating perspective images, sorting data, or solving the final orientation and field-of-view, are only referenced by calls to function.

A.1 Grid-Based Indexing

Input

Catalog:	**A list of stars consisting of position and brightness in 3D-coordinates.**
KeyPairs:	**The pairs of stars generated via the code in Listing 1, in Section 2.2.**
⊙:	**The maximum angular distance in which to look for stars to construct the grid bins with.**
f:	**The fineness of the grid used for binning.**
MinStarsPerBin:	**The minimum number of stars above which a bin is considered 'full'.**

Output

BinTable:	**A table of bit-strings. The table is organized by row, and each bit represents the bin status far a particular pair of stars. See Section 2.4.1 for details.**

Variables

NearStars:	**A list of stars that are closer than ⊙ to the current star of interest.**
s1, s2, **s**:	**Single stars.**
Bins1, Bins2:	**Lists of bins built according to the fineness parameter 'f'.**

StarBin: The coordinates of a bin into which a star falls.

RotateAngle: The angle at which the pair of stars that define the grid are rotated relative to a 2D reference frame.

Functions

Angle3D(s1,s2): The angle between two stars s1 and s2 in 3D-coordinates.

Angle2D(s1,s2): The angle between two stars s1 and s2 when projected into a 2D image centered on s1, relative to a horizontal line.

BuildEmptyBins(f):
Returns an empty list of grid bins built according to the fineness parameter 'f'.

CalcBin(s1, angle, s, f):
Returns the bin index that the star 's' falls into relative to the center of a 2D-grid generated around star 's1', rotated at an angle 'angle', where the bins have fineness 'f'.

BinsToBitString(bins, MinStarsPerBin):
Converts the 2-dimensional list of bins 'bins' into a 1-dimensional list of bits where the bits are set according the guidelines in Section 2.4.1. MinStarsPerBin is the number of stars above which a bin is considered full.

Algorithm

```
BinTable ← {}

// Loop through all pairs
For All {s1,s2} ∈ KeyPairs
  RotateAngle ← Angle2D(s1,s2)

  // Prepare the grid information
  Bins1 ← BuildEmptyBins(f)
  Bins2 ← BuildEmptyBins(f)

  // See which stars in the catalog fall into which bins
  For All s ∈ Catalog

    // See if the star is close enough to bother performing a complete check of
    // its position in the grid
    If ( Angle3D(s1,s) ≤ θ )

      // calculate the position of the star 's' in the bins
      StarBin ← CalcBin(s1, -RotateAngle, s, f)
      // Increase the number of stars in the bin
      Bins1{StarBin} ← Bins1{StarBin}+1
    End If
    If ( Angle3D(s2,s) ≤ θ )

      // Calculate the position
      StarBin ← CalcBin(s2, Π-RotateAngle, s, f)
      // Increase the number of stars in the bin
      Bins2{StarBin} ← Bins2{StarBin}+1
    End If
```

```
  End For

  // Construct the final bit-strings
  BinTable ← BinTable ∪ {s1, s2, BinsToBitString(Bins1, MinStarsPerBin)}
  BinTable ← BinTable ∪ {s2, s1, BinsToBitString(Bins2, MinStarsPerBin)}
End For
```

A.2 Grid-Based Matching

Input

Catalog: A list of stars consisting of position and brightness in 3D-coordinates.

KeyPairs: The pairs of stars generated via the code in Listing 1, in Section 2.2.

f: The fineness of the grid used for binning.

MinStarsPerBin: The minimum number of stars above which a bin is considered full.

MaxPosError: The maximum positional error in the positions of the stars in the image.

Image: The test image to be solved, containing star positions and brightness information.

Output

Solutions: A table of bit-strings. The table is organized by row, and each bit represents the bin status far a particular pair of stars. See Section 2.4.1 for details.

Variables

ImagePairs: A list of pairs of stars in the image that can be used to orient a grid.

s1, s2, s: Single stars.

RotateAngle: The angle at which the pair of stars that define the grid are rotated relative to a 2D reference frame.

Bins1, Bins2: Lists of bins built according to the fineness parameter 'f'.

StarBin: The coordinates of the bins into which a star falls. We say bins because due to 'MaxPosError' the star might fall into several bins.

ZeroBins1, ZeroBins2: A list of bins whose bit-string value is set to zero.

ZeroCount1, ZeroCount2: The number of bins listed in 'ZeroBins1' and 'ZeroBins2' respectively.

candidates1, candidates2: Bit-strings representing the pairs in the index that are candidates for the identities of the current pair of image stars being examined.

***pairs1, pairs2, pairList*: The pairs of stars that are candidates for the identities of the current pair of image stars being examined.**

Functions

***Angle*(s1,s2):** Returns the angle between stars 's1' and 's2' relative to the horizontal of the image.

***BuildEmptyImageBins*(s1, angle, f):** Returns an empty list of grid bins based on the fineness parameter 'f', the position 's1' in the image, and an angle of rotation for the grid. The bins returned correspond to the bins strictly visible within the image boundaries.

***CalcBin*(s1, s2, angle, s, f, MaxPosError):** Returns the bin or bins into which image star 's' *could* fall into given a grid fineness of 'f', a grid center of 's1', and a grid rotation of 'angle'. MaxPosError and s2 are passed to the function to allow proper calculation of bins 's' could fall into based on the maximum of positional error of 's' and the maximum rotation error of the grid caused by the maximum positional error of 's1' and 's2'.

***FindZeroBins*(binList, MinStarsPerBin):** Returns a list of the bin positions that are set to zero, based on the MinStarsPerBin' parameter and the rules outlined in section 2.4.1.

***bitwiseAnd*(bits1, bits2):** Returns the result of a long bitwise-and of the bit-strings 'bits1' and 'bits2'.

***GetPairs*(bits, KeyPairs):** Returns a list of pairs from 'KeyPairs' which correspond to the bits set to 1 in the bit-string 'bits'.

***ReversePairs*(pairs):** Returns a list with the pairs in the list 'pairs' reversed.

***SolveOrientations*(pairList, s1, s2, Catalog):** Returns a list of image orientations based on the pairs in 'pairList' corresponding to the stars 's1' and 's2' in the test image. 'Catalog' is used to provide positional information about the stars.

***FinalCheck*(Solutions, Catalog):** Returns a list of final plausible orientation solutions produced by using the solution orientations in 'Solutions' to test the test image 'Image' against the star catalog 'Catalog'.

Algorithm

```
// Find all star pairs in the image and sort by decreasing brightness
ImagePairs ← {}
For All s1 ∈ Image
  For All s2 ∈ Image
    If s1 ≠ s2
      ImagePairs ← ImagePairs ∪ {s1,s2}
    End If
  End For
```

```
End For
ImagePairs ← SortPairDescending(ImagePairs)

// Examine all the pairs in order from brightest to dimmest
For All {s1,s2} ∈ ImagePairs
  Solutions ← {}
  RotateAngle ← Angle(s1,s2)

  // Prepare the grid information for grids with either star of the pair
  // at the center of the grid
  Bins1 ← BuildEmptyImageBins(s1, -RotateAngle, f)
  Bins2 ← BuildEmptyImageBins(s2, Π-RotateAngle f)

  // Place the star in the appropriate bin for each grid
  For All s ∈ Image
    StarBins ← CalcBin(s1, s2, -RotateAngle, s, f, MaxPosError)
    Bins1{StarBins} ← Bins1{StarBins}+1
    StarBins ← CalcBin(s2, s1, Π-RotateAngle, s, f, MaxPosError)
    Bins2{StarBins} ← Bins2{StarBins}+1
  End For

  // Determine the discriminating bins
  ZeroBins1 ← FindZeroBins(Bins1, MinStarsPerBin)
  ZeroCount1 ← |EmptyBins1|
  ZeroBins2 ← FindZeroBins(Bins2, MinStarsPerBin)
  ZeroCount2 ← |EmptyBins2|

  // If solutions resulted from both orientations
  If (|ZeroBins1| > 0) and (|ZeroBins2| > 0)

    // perform bitwise AND's over the rows of the index table to construct a
    // bit-string where 1's indicate potentional solution candidates
    candidates1 ← BinTable{ZeroBins[1]}
    For I ← 2 up to ZeroCount1
      candidates1 ← bitwiseAnd(candidates1, BinTable{ZeroBins1[I]})
    End
    candidates2 ← BinTable{ZeroBins[1]}
    For I ← 2 up to ZeroCount2
      candidates2 ← bitwiseAnd(candidates2, BinTable{ZeroBins2[I]})
    End
  End If

  // Get the actual pair identities and find the pairs that appear in both
  // lists
  pairs1 ← GetPairs(candidates1, KeyPairs)
  pairs2 ← ReversePairs(GetPairs(candidates2), KeyPairs)
  pairList ← pairs1 ∩ pairs2

  // Get the final list of solution orientations
  Solutions ← Solutions ∪ SolveOrientations(pairList, s1, s2, Catalog)
  Solutions ← FinalCheck(Solutions, Image, Catalog)
  If (|Solutions| > 0)
    return Solutions
  End If
End For
```

A.3 Shape-Based Indexing

Input

Catalog: A list of stars consisting of position and brightness in 3D-coordinates.

Θ: The maximum angular distance around which to look for stars to construct shapes.

Φ: The maximum angle of the wedges with which to look for stars to construct shapes with.

StarsPerShape: The number of stars used in the constructed shapes.

Output

ShapeTable: A table containing shapes formed from the stars in the catalog. The entries are organized according to the first distance ratio in the shape.

Variables

NearStars: A list of stars that are at least as close as the angle Θ to the current star of interest.

n, s, n1, n2: Single stars.

ShapeStars: A list of stars that compose a shape.

OrientingStar: The star used to orient the direction of the wedge in which shapes are looked for.

MaxDist: The distance of the star in a shape that is furthest from the apex of the wedge in which the shape was found.

ShapeDistances: The relative distances of the stars in a shape from the apex of the wedge in which they were found.

ShapeAngles: The angles of the stars in a shape relative to the star used to orient the wedge in which they were found.

I: A counting variable.

Functions

Angle(star1, star2):
A function that takes two stars in 3D-coordinates and returns the angle between them.

ABS(val):
Returns the absolute value of val.

CameraProjectedAngle(centerStar, star1, star2):
A function that takes a central star and two other stars. The two stars, 'star1' and 'star2' are projected into an image centered on 'centerStar', and then the angle between them is calculated in 2D-coordinates.

CameraProjectedDistance(star1, star2):
Takes two stars and returns the distance between them in a 2D-image centered on 'star1'.

SortIncreasingDistance(star1, starList):
Takes a single star and a list of stars. The function returns a increasing sorted list of the original star list, where distance is measured using CameraProjectedDistance().

Algorithm

```
ShapeTable ← {}
For All s ∈ Catalog

  // Find all the stars close enough to the star 's' to be used in shapes
  NearStars ← {}
  For All n ∈ Catalog
    If ( ABS(Angle(s,n)) ≤ θ )
      NearStars ← NearStars ∪ n
    End If
  End

  // Search for shapes among the set of stars previously generated
  For All n1 ∈ NearStars
    ShapeStars ← {}

    // Create a list of stars that all fall within the same narrow angular
    // wedge
    OrientingStar ← n1
    For All n2 ∈ NearStars
      If ( ABS( CameraProjectedAngle(s, OrientingStar, n1) ) ≤ Φ/2 )
        ShapeStars ← ShapeStars ∪ n2
      End If
    End For

    // If there are enough stars in the wedge to construct a shape, construct
    // the shape and save it
    If ( |ShapeStars| ≥ StarsPerShape )

      // Grab the nearest stars to the apex of the wedge
      ShapeStars ← SortIncreasingDistance(s,ShapeStars)
      ShapeStars ← ShapesStars[1… StarsPerShape]
      MaxDist ← CameraProjectedDistance(ShapeStars[StarsPerShape],s)

      // calculate the relative distances and angles and store the shape
      I ← 1
      For All n2 ∈ ShapeStars
        ShapeDistances[I] ← CameraProjectedDistance(ShapeStars[I],s)/MaxDist
        ShapeAngles[I] ← CameraProjectedAngle(OrientingVector, n1)
        I ← I+1
      End For
      ShapeTable ← ShapeTable ∪ {ShapeDistances, ShapeAngles, s ∪
                                                          ShapeStars}
    End If
  End For
End For
```

A.4 Shape-Based Matching

Input

Catalog:	A list of stars consisting of position and brightness in 3D coordinates.
Φ:	The maximum angle of the wedges in which to look for stars to construct shapes with.

StarsPerShape: **The number of stars used in the constructed shapes.**

***AngleTolMin*:** **The minimum tolerance to apply when matching angles.**

***AngleTolMax*:** **The maximum tolerance to apply when matching angles.**

***DistTolMin*:** **The minimum tolerance to apply when matching distances.**

***DistTolMax*:** **The maximum tolerance to apply when matching distances.**

***Image*:** **A list of the stars in the image to solve in 2D-coordinates.**

***ShapeTable*:** **The index of shapes to match shapes against.**

Output

***Solutions*:** **A list of solution orientations and fields-of-view that the matching algorithm believes may be the correct solution.**

Variables

***s, n1, n2, n3*:** **Single stars.**

***ShapeStars*:** **A list of stars that compose a shape.**

***OrientingStar*:** **The star used to orient the direction of the wedge in which shapes are looked for.**

α: **The tolerance to apply when matching a particular angle.**

D: **The tolerance to apply when matching a particular distance.**

***MaxDist*:** **The distance of the star in a shape that is furthest from the apex of the wedge in which the shape was found.**

***StarCount*:** **The number of stars contained in a wedge that may contain a shape.**

***MinLastStar*:** **The nearest star to the apex of a wedge that could be considered the farthest star in a shape.**

***MaxFirstStar*:** **The farthest start from the apex of a wedge that could be considered the first star in a shape.**

***ShapeDistances*:** **The relative distances of the stars in a shape from the apex of the wedge in which they were found.**

***ShapeAngles*:** **The angles of the stars in a shape relative to the star used to orient the wedge in which they were found.**

***ShapeList*:** **A list of shapes that might match the current shape in the image that is being examined.**

***shape*:** **The current shape from ShapeList that is being matching against the current shape in the image.**

***MatchCount*:** **The number of stars that have been matched in two shapes.**

***ShapePosition*:** **The current star position in *shape* that is being compared to a shape in the image.**

***ImageShape*:** **A list of the stars from an image shape that properly match**

```
                    stars in a shape in the index.

   I, J, K:         Counting variables.

Functions

Angle(star1, star2):
                    A function that takes two stars in 2D-coordinates and
                    returns the angle between them.

SortIncreasingDistance(star1, starList):
                    Takes a single star and a list of stars. The function
                    returns a increasing sorted list of the original star list,
                    where distance is measured using CameraProjectedDistance().

ABS(val):
                    Returns the absolute value of 'val'.

MAX(val1, val2):
                    Returns the maximum value between 'val1' and 'val2'.

Order(star, starList):
                    Returns the index position in which a star appears in a
                    list of stars.

Distance(star1, star2):
                    Returns the distance between two stars in 2D-coordinates.

AngleTolerance(star, origin):
                    Returns a tolerance with which to match angles. The
                    tolerance is based on the position of a star relative to
                    the origin.

DistanceTolerance(star, origin):
                    Returns a tolerance with which to match distances. The
                    tolerance is based on the position of a star relative to
                    the origin.

StarMatch(shapeStar, angle, distance, α, D):
                    Returns True if the angle and distance are fall within the
                    tolerances 'α' and 'D' of the corresponding angle and
                    distance of 'shapeStar'. Otherwise it returns False.

SolveOriention(imageShape, shape):
                    Returns a solution orientation and field-of-view based on a
                    correspondence between the stars in 'imageShape' and
                   'shape'.

Algorithm

Solutions ← {}
For All s ∈ Image
  ShapeStars ← {}
  For All n1 ∈ Image

    // Use 'n1' to find all the stars that fall within the same wedge
    OrientingStar ← n1
    For All n2 ∈ Image
      α ← AngleTolerance(n2, s, AngleTolMin, AngleTolMax)
      If ( ABS(Angle(s, OrientingStar, n2)) ≤ (Φ/2+ α) )
        ShapeStars ← ShapeStars ∪ n2
      End If
    End For
```

```
// If there are enough stars in the wedge to make a shape with,
// check all the combinations of shapes against the index
If ( |ShapeStars| ≥ StarsPerShape )
  ShapeStars ← SortIncreasingDistance(s,ShapeStars)
  StarCount ← |ShapeStars|
  MinLastStar ← MAX( Order(OrientingStar, ShapeStars), StarsPerShape)

  // Loop through all the stars that could be considered the 'furthest'
  // star in the shape.
  For I ← StarCount down to MinLastStar
    n2 ← ShapeStars[I]
    MaxDist ← Distance(s, n2)
    J ← 1

    // Calculate all the distances and angles that will be needed for shape
    // matching
    For All n3 ∈ ShapeStars
      ShapeDistances[J] ← Distance(s, n3) / MaxDist
      ShapeAngles[J] ← Angle(s, OrientingStar, n3)
      J ← J+1
    End For
    MaxFirstStar ← MAX( Order(OrientingStar, ShapeStars),
                        MinSearchStar-StarsPerShape)
    // loop through the different stars that could be the first star in the
    // shape.
    For K ← 1 up to MaxFirstStar
      α ← AngleTolerance(ShapeStars[K], s, AngleTolMin, AngleTolMax)
      D ← DistanceTolerance(ShapeStars[K], s, DistTolMin, DistTolMax)

      // Get the list of shapes that might match the current image shape
      // using some subset of the image stars
      ShapeList ← ShapeTable{ShapeDistances[1], D}

      // Try to match all the shapes in the list that was just genereated
      For All shape ∈ ShapeList

        // Match the first star. This star *must* match in order to
        // continue the matching since it was used to look up the list of
        // shapes we are now checking
        If ( StarMatch(shape{1}, ShapeDistances[K],
                    ShapeAngles[K], α, D) = False )
          Next shape
        End If
        MatchCount ← 1
        ShapePosition ← 2
        ImageShape ← {}

        // Attempt to match the remaining stars in the indexed shape using
        // an ordered subset of the stars in the wedge
        For J ← K+1 up to I
          α ← AngleTolerance(ShapeStars[J], s, AngleTolMin, AngleTolMax)
          D ← DistanceTolerance(ShapeStars[J], s, DistTolMin, DistTolMax)
          If ( StarMatch(shape[ShapePosition], ShapeDistances[J],
                      ShapeAngles[K], α, D) = True )
            MatchCount ← MatchCount+1
            ImageShape ← ImageShape ∪ ShapeStars[J]
            ShapePosition ← ShapePosition + 1
          Else If (J ← Order(OrientingStar)) or (ShapePosition > |shape|)
            Next shape
          End If
```

```
            End If

            // If enough stars were matched, solve for the final orientation
            If ( MatchCount = StarsPerShape )
              Solutions ← Solutions ∪ SolveOriention(ImageShape, shape,
                                                     Catalog)
            End If
          End For
        End For
      End For
    End If
  End For
End For
```

Bibliography

[1] Clouse, D. S., and Padgett, C. W., "Small Field-of-View Star Identification Using Bayesian Decision Theory," IEEE transactions on aerospace and electronics systems, Vol. 36, No. 3, July 2000, pp773

[2] Groth, E. J., "A Pattern-Matching Algorithm For Two-Dimensional Coordinate Lists," The Astronomical Journal, Vol. 91, No. 5, May 1986, pp. 1244-1248.

[3] Juang, J., Kim, H., Junkins, J. L., "An Efficient and Robust Singular Value Method for Star Pattern Recognition and Attitude Determination," the John L. Junkins Astrodynamics Symposium 2003 (Advances in the Astronautical Sciences), Vol. 115.

[4] Kim, K.T., and Bang, H., "Reliable Star Pattern Identification Technique by Using Neural Networks," the John L. Junkins Astrodynamics Symposium 2003 (Advances in the Astronautical Sciences), Vol. 115.

[5] Kim, H., Junkins, J. L., and Mortari, D., "A New Star Pattern Recognition Method: Star Pair Axis and Image Template Matrix Method", Core Technologies for Space Systems Conference (Communication and Navigation Session), 2001.

[6] Liebe, C.C., "Pattern recognition of star constellations for spacecraft applications", IEEE Aerospace and Electronic Systems Magazine, Vol. 8, No. 1, Jan 1993, pp 31-39.

[7] Lalitha Paladugu, Brian G. Williams, and Marco P. Schoen, "Star Pattern Recognition for Attitude Determination using Genetic Algorithms", Presented at the 17th AIAA/USU Conference on Small Satellites, August 11-14, in Logan, UT, 2003.

[8] Lalitha Paladugu, Marco P. Schoen, Brian G. Williams, "Intelligent Techniques for Star-Pattern Recognition", Proceedings of ASME, IMECE 2003, November 16-21, 2003.

[9] Malak A. Samaan, Daniele Mortari, and John L. Junkins, "Non-Dimensional Star Identification for Un-Calibrated Star Cameras", The AAS/AIAA Space Flight Mechanics Meeting. Ponce, Puerto Rico, 9-13 February 2003.

[10] Malak A. Samaan, Todd Griffith, Puneet Singla, and John L. Junkins, "Autonomous On-Orbit Calibration of Star Trackers," Core Technologies for Space Systems Conference (Communication and Navigation Session), 2001.

[11] Mortari, D., "Search-Less Algorithm for Star Pattern Recognition," Journal of the Astronautical Sciences, Vol. 45, No. 2, April-June 1997, pp. 179-194.

[12] Mortari, D., Junkins, J.L., and Samaan, M.A. "Lost-In-Space Pyramid Algorithm for Robust Star Pattern Recognition", Guidance and Control 2001 (Advances in the Astronautical Sciences) , Vol. 107.

[13] Samaan, M.A., Mortari, D., and Junkins, J.L., "Recursive Mode Star Identification Algorithms", Spaceflight Mechanics 2001 (Advances in the Astronautical Sciences), Vol. 108.

[15] Schonemann, P., "A Generalized Solution of the Orthogonal Procrustes Problem," Psychometrika, No, 31, 1966, 1-10.

[16] Sheela, B. V., and Shekhar, C., "New Star Identification Technique for Attitude Control," Journal of Guidance, Control, and Dynamics, Vol. 14, No. 2, Mar.-Apr. 1991, 477-480.

[17] Udomkesmalee, S., Alexander, J. W., and Tolivar, A. F., "Stochastic Star Identification," Journal of Guidance, Control, and Dynamics, Vol. 17, No. 6, Nov.-Dec. 1994, 1283-1286.

[18] "Astronomer's Control Program PinPoint Astrometric Engine," http://pinpoint.dc3.com.

[19] SkyView Virtual Observatory. http://skyview.gsfc.nasa.gov/.