Astrometry.net: Automatic recognition and calibration of astronomical images

by

Dustin Lang

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Astrometry.net: Automatic recognition and calibration of astronomical images

Dustin Lang

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2009

We present *Astrometry.net*, a system for automatically recognizing and astrometrically calibrating astronomical images, using the information in the image pixels alone. The system is based on the *geometric hashing* approach in computer vision: We use the geometric relationships between low-level features (stars and galaxies), which are relatively indistinctive, to create geometric features that are distinctive enough that we can recognize images that cover less than one-millionth of the area of the sky. The geometric features are used to generate rapidly hypotheses about the location—the pointing, scale, and rotation—of an image on the sky. Each hypothesis is then evaluated in a Bayesian decision theory framework in order to ensure that most correct hypotheses are accepted while false hypotheses are almost never accepted. The feature-matching process is accelerated by using a new fast and space-efficient kd-tree implementation. The *Astrometry.net* system is available via a web interface, and the software is released under an open-source license. It is being used by hundreds of individual astronomers and several large-scale projects, so we have at least partially achieved our goal of helping "to organize, annotate and make searchable all the world's astronomical information."

Acknowledgements

First, I want to thank Sam Roweis for accepting me as his student, for handing me the nascent *Astrometry.net* project which has been so fun to work on, for providing help, support, and encouragement throughout, and, most importantly, for his warm friendship.

I owe huge thanks to David Hogg, for serving as my semi-official co-supervisor, mentor, and guide to the world of astronomy. It is hard to imagine what my life would be like if I hadn't met him: no Delion-fuelled hack sessions in New York, no lovely summers in Heidelberg, and I almost certainly would not have made the jump into astronomy. He has shown me a great deal about how to be a good scientist and how to love my job and love my family at the same time. It has been a great privilege working with him.

I have been fortunate to work with a number of great people on the *Astrometry.net* team, in particular Keir Mierle, who taught me many things about the practice of software development, and Christopher Stumm, who has done excellent work in bringing *Astrometry.net* to the amateur astronomy community. I must also thank the army of alpha-testers who have provided a lot of bug reports, great ideas for improving the system, and some very encouraging feedback.

It is a pleasure to thank Iain Murray for many helpful ideas and insights; for listening politely while I rambled about some half-understood problem I was working on; and for being so amusingly English.

I thank Sven Dickinson and Rich Zemel for serving on my advisory committee. They provided excellent guidance, advice, and suggestions for improving this thesis. I thank Doug Finkbeiner for taking the time to read my thesis and provide thoughtful comments, and for travelling to Toronto to serve as my external examiner.

I thank my mom and my brother, Gerda and Devon Lang, who are two fantastic people who have always been there for me.

I give huge thanks to my lovely wife Micheline. A PhD is a team effort, and she is the foundation of Team Lang. She provided a huge amount of love and encouragement,

proof-read thesis drafts, forced me to practice my talks and listened patiently while I did, and kept our household running while I was embroiled in writing. I couldn't have done it without her. There are exciting times ahead and I'm so very happy to have her at my side.

Finally, I thank my baby son Julian for having brought so much joy into our lives, and for sleeping soundly while I write these words.

Contents

Chapter 1

Introduction

1.1 Preface

1.1 This thesis describes some of the research carried out by me and my colleagues in the *Astrometry.net* group. *Astrometry.net* started as a collaboration between Sam Roweis, a University of Toronto computer scientist, and David W. Hogg, a New York University astronomer. An idea that began as a crazy scrawl on the back of a napkin—probably in a bar—eventually grew into a real project and began attracting collaborators and students. Fast-forwarding a few years, we have achieved a rare feat: a computer vision software system that *works*, with little human intervention, and is being used by hundreds of real astronomers doing real research. We have also branched out and explored a number of other promising areas where ideas in computer vision and machine learning can be applied to astronomical data to great benefit.

1.2 This thesis is quite blatantly a collection of manuscripts. I have done a bit more than staple them together to produce the chapters of this thesis, but the original tone and style of each manuscript still shines through. I hope my readers will find the chapter-to-chapter variations a refreshing, rather than jarring, change of pace. One advantage of collecting manuscripts into a thesis is that each chapter should largely stand alone. Another advantage is that each manuscript was prepared with a list of authors, so it is simpler to identify the collaborators who have contributed to each chapter.

1.2 Introduction

1.3 The general idea of the *Astrometry.net* collaboration is to apply ideas in computer vision and machine learning to problems and data in astronomy. Astronomical images are a good domain in which to explore ideas in computer vision, because the problems to be solved are often much more constrained than in general imaging, the telescopes and cameras are often well-understood and calibrated, large volumes of data and "ground truth" information exist, and with new telescopes constantly coming online and producing

greater and greater volumes of imaging, there is a need for sophisticated computation. I briefly outline astronomical imaging for computer vision researchers in section 1.3.

1.4 The specific idea addressed in this thesis is that pattern recognition approaches can be applied to astronomical images. The goal is to produce a system that is able to recognize automatically the stars in any image of the night sky, using the information in the image pixels alone. Recognizing the stars in an image is equivalent to finding the pointing, scale, and rotation of the image on the sky in a standard coordinate system. The branch of astronomy concerned with measuring the positions and motions of celestial bodies is called *astrometry*, and the task of placing a new image in a standard reference frame is known as "calibrating the astrometry" of the image. The task of calibrating the astrometry of an image using only the image itself—not given a prior on the location—was historically known as *blind astrometric calibration*. We recognize[1], that this is an ableist phrase, and instead prefer *full-sky astrometric calibration*. The broad goal of the *Astrometry.net* project was to build a system that would allow us to create correct, standards-compliant astrometric meta-data for every useful astronomical image ever taken, past and future, in any state of archival disarray. This is part of a larger effort to organize, annotate and make searchable all the world's astronomical information.

1.5 After a brief introduction to astronomical imaging for computer scientists, the remainder of this chapter reviews the area of pattern recognition, and in particular the framework of *geometric hashing* for object recognition in images. One of the contributions of this thesis is to replace the simple hash table used in traditional geometric hashing with a kd-tree, so I review related work in hashing and other approaches for fast feature matching. Finally, I present the astrometric calibration task as an instance of object recognition, and review some previous approaches to the problem.

1.6 The remaining chapters present our approach to the astrometric calibration problem. Chapter 2 explains our approach and presents the results of large-scale tests of the system

[1]Retroactively, in the year 2020

on real-world data. Chapters 3 and 4 delve into details of the approach: Chapter 3 presents the Bayesian decision theory problem that lies at the heart of our approach, while chapter 4 explains the technical details of the kd-tree data structure implementation that is key to making our system fast enough to be a practical tool. Chapter 5 summarizes our results.

1.3 Astronomical imaging for computer scientists

1.7 Astronomical images are very different from images produced by typical consumer-grade cameras. This section is intended to give non-astronomers a sense of the typical parameters of contemporary astronomical imaging setups, and introduce some of the terminology and key ideas.

1.8 The Sloan Digital Sky Survey (SDSS) [69, 29] is one of the most important projects in contemporary astronomy. The primary imaging survey was carried out from 2000 through 2008, so the hardware is no longer cutting-edge, but it is still very impressive. The telescope's primary mirror is 2.5 meters in diameter. Incoming light first strikes the primary mirror, then a one-meter secondary mirror, then passes through two corrective lenses in front of the camera. The nearly distortion-free field of view is about 3 degrees wide. The camera is composed of 30 charge-coupled devices (CCDs), each 2048×2048 pixels, arranged in six columns. The five CCDs in each column have different bandpass filters, ranging from the near-infrared (913 nm) through the optical to the near-ultraviolet (354 nm). The CCDs and associated electronics are cooled with liquid nitrogen to $-80°$C and held under vacuum. Each CCD is about 5 cm square, and the whole focal plane of the camera is about 65 cm in diameter. The camera assembly has a mass over 300 kg.

1.9 The survey operates in "drift-scanning" mode: the telescope is pointed in a nearly fixed direction, and as the Earth rotates the sky drifts past the camera. The camera electronics shift the electrons along the CCD columns at the same rate. As an object

drifts through the field of view, the electrons it produces in the CCD pixels march along with it, and are read out as it disappears from view. In the SDSS camera, the object drifts past each of the five bandpass-filtered CCDs of the camera in turn, producing a five-color image. Over the course of a night of observing, the camera sweeps out a great circle, creating six columns of five-band images that are 2048 pixels wide and over a million pixels long: over 120 gigabytes of pixels per night. This clever drift-scanning scheme has many advantages, one being that the readout rate of the CCDs can be relatively slow, resulting in low readout noise. Calibration of the camera is simpler because charge is integrated over columns of the CCD: it is only necessary to calibrate whole columns rather than individual pixels. Note that for this scheme to work, the CCDs must be very carefully aligned in the focal plane, and the telescope control system must be capable of keeping the system pointed stably at the level of a pixel or better. The moving mass of the telescope is over $15,000$ kg, so this is no mean feat!

1.10 Astronomical imaging systems are designed so that the point-spread function (PSF)—the image on the CCD of a distant point source of light—is spread over several pixels. This allows the position of a point source to be measured to sub-pixel accuracy by fitting a model of the PSF to the observed pixel values. If the system focused a point source onto a single pixel, this would be impossible. The goal is typically for the full-width at half-maximum (FWHM) of the PSF to be a few pixels, so that the PSF is well-sampled but the signal is not spread out over too many pixels.

1.11 For ground-based telescopes, the moving atmosphere causes the image of a distant point source of light to dance around on the CCD; in typical exposures of many seconds this "speckling" is integrated into a point-spread function. The size of this PSF is called the "seeing" and a value of one arcsecond FWHM is considered quite good. The SDSS camera has pixels of size 0.396 arcseconds, and the point-spread function is dominated by the seeing: the optics of the system are good enough that the images are as clear as they can be given the atmosphere.

1.12 The positions of astronomical objects are usually measured in an *equatorial coordinate system*: essentially the projection of the Earth's latitudes and longitudes onto the celestial sphere at a specific instant. The latitudinal direction is called "declination" and abbreviated "Dec", and the longitudinal direction is called "right ascension" and abbreviated "RA". Declination is typically measured in degrees from -90 to $+90$, and right ascension is typically measured in degrees from 0 to 360, or hours from 0 to 24. There are 60 arcminutes per degree and 60 arcseconds per arcminute. For reference, the Moon as seen from Earth is about half a degree in diameter. The 0.396 arcsecond pixels of the SDSS camera view an area the size of a penny at a distance of ten kilometers; each 2048×2048 pixel CCD views an area of about one-millionth of the sky.

1.13 The SDSS is very impressive, but the next generation of telescopes currently in development dwarf its specifications. The Large Synoptic Survey Telescope (LSST) [35], for example, has an 8.4 meter primary mirror, with a 3.5 degree field of view almost completely covered by an array of nearly 200 CCDs, each with 4096×4096 pixels: a total of about 3.2 gigapixels. The camera is the size of a small car. It will take exposures of about 15 seconds, yielding over 10 terabytes of data per night.

1.4 Pattern recognition

1.14 *Pattern recognition* encompasses a huge class of problems and techniques in computer vision and machine learning. Many pattern recognition problems are of the type "identify this object" or "find things like this": the system is first told about a large set of objects, then is presented with a novel object and is asked to describe it or find similar objects from the set of known objects.

1.15 Pattern recognition systems often take a *feature matching* approach. A *feature* is a compact description of (problem-specific) relevant, invariant properties of an object, such that objects that are similar in the problem domain have similar features. Defining

how features are extracted from objects and defining the similarity between features are key factors in the success of such systems. Often the representation of a feature (the *feature descriptor*) is a point in a real coordinate space and a distance metric defines the similarity between features.

1.16 Feature-based approaches typically involve two phases: an *indexing* phase, in which the set of known objects is shown to the system, and a *test* phase, in which novel objects are presented. During indexing, features are extracted from the set of known objects. At test time, features are extracted from the novel object and compared with the set of features from the known objects. In many domains, the set of known objects is very large, so it is critical that features be stored in a manner that allows the system to rapidly find known features that match a given feature.

1.17 Often feature-based approaches suffer *feature aliasing* problems: two objects that are dissimilar in the problem domain may yield similar features. In this case, systems may use *voting* or *verification* schemes to ensure that aliased (false) feature matches are eliminated. Voting schemes typically assume that false feature matches are independently randomly distributed, so the probability of matching several features to the same incorrect object is the product of the individual probabilities; if we demand a sufficient number of matches, the probability of a false match becomes small. Verification or *hypothesize-test* schemes treat a feature match as a hypothesis that the novel object is similar to a known object, and then some test to decide if the hypothesis is correct. Such schemes can be seen as using feature matching as a search heuristic.

1.5 Visual pattern recognition

1.18 Pattern recognition in images is a particularly challenging and complex problem domain which has been studied extensively. Objects to recognize include faces, fingerprints, specific physical objects, or even general classes of physical objects such as mugs, chairs,

or elephants.

1.19 Visual pattern recognition of general physical (3-dimensional) objects is difficult for many reasons: objects may be deformable, can appear different when viewed from different angles or under different lighting conditions, and can be occluded by other objects. Features used in visual pattern recognition include edges (lines and curves), corners, or the color or texture of image patches.

1.20 An important consideration when designing features for visual pattern recognition is the size (*i.e.*, spatial extent in the image) of the feature. *Local features* gather information from a small region of the image, while *global features* incorporate information from the whole image. Local features can provide robustness to occlusion and deformation: if part of the object is hidden, features belonging to that part will be lost, but the features belonging to the remaining parts will still be found. Since local features aggregate information from only a small portion of the image, they may be more noisy or less distinctive than global features.

1.21 Since a single local feature may be insufficient to guarantee that an object has been correctly recognized, some systems build high-level features by combining multiple low-level features. When the objects to be recognized are rigid or articulated (composed of multiple rigid parts connected by simple joints), a common approach is to build features that describe the relative geometry of low-level features. This use of *geometric features* is often called *geometric hashing*, since the result is a feature descriptor or *hash code* which describes the geometric arrangement of the low-level features.

1.6 The geometric hashing framework

1.22 Lamdan *et al.* [42] and Wolfson and Rigoutsos [67] give an excellent overview of the geometric hashing approach. See Figure 1.1 for a block-diagram overview. Geometric hashing is based on the fact that when rigid objects undergo similarity transformations

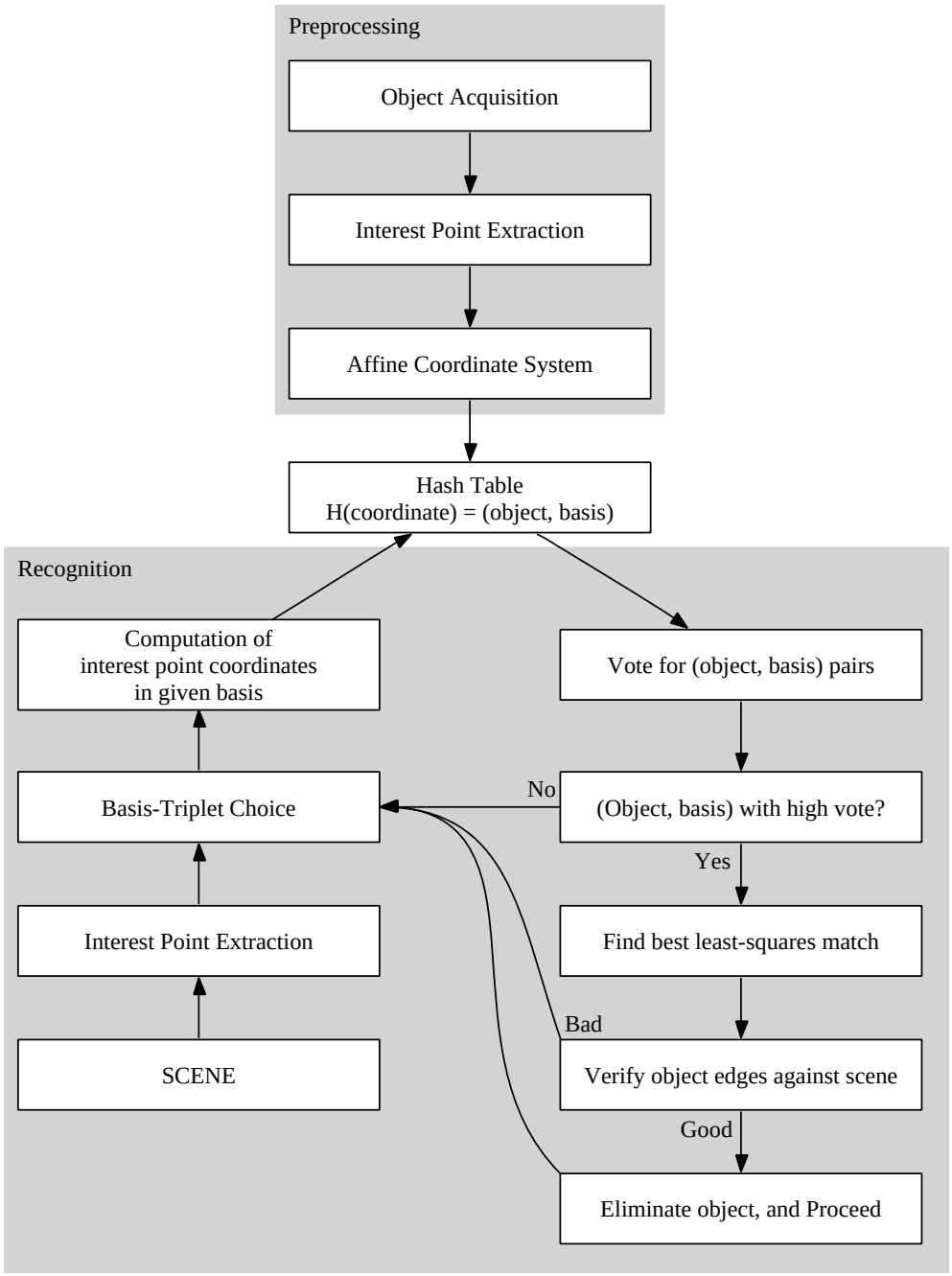


Figure 1.1: Outline of the Geometric Hashing scheme. This diagram is a reproduction of figure 2 in Lamdan *et al.* [42].

(translation, rotation, and scaling), angles between points on the object and relative distances between points do not change. Therefore, if we describe the relative arrangement of points on an object in terms of these invariants, our description will be invariant under these transformations of the object. Invariant geometric properties can be found for other types of transformations such as affine or perspective projection.

1.23 Geometric hashing builds *geometric features* out of lower-level features. We will call the low-level features *interest points* for clarity.

1.24 Given the types of transformations the system must handle, there is some sufficient number of interest points required to define an invariant local coordinate system; these are called the *basis set*. For example, for 2-D objects that can undergo similarity transformations, two interest points can be used to define a coordinate system. For 3-D objects with similarity transformations, three (non-collinear) interest points are required.

1.25 Geometric hashing follows an indexing approach. During the indexing phase, the system preprocesses a database of objects which are to be recognized. For each object, many interest points are extracted which are invariant under the transformations that are expected, and can be well localized. The system then enumerates all basis sets by looking at all combinations of interest points; each set of basis features defines a relative coordinate system. For each basis set, all remaining interest points are enumerated and their positions described in this coordinate system. Each relative coordinate vector forms a *geometric feature descriptor*. In this way, the relative geometric arrangement of a set of interest points is converted into a point in a vector space.

1.26 In the standard geometric hashing approach, the geometric feature descriptor is discretized and encoded as a hash table key. A grid is placed over the feature descriptor space, and the hash table key of a geometric feature is the identifier of the grid cell containing it. Each entry in the hash table maps back to the basis set plus the feature whose position is encoded.

1.27 Given a new image, a geometric hashing system extracts all interest points from

the image and, as at indexing time, proceeds to enumerate all basis sets and computes the positions of the remaining interest points in each basis set. This generates many geometric features whose descriptors are discretized and converted to hash table keys, as before. Similar arrangements of interest points will lead to similar geometric feature descriptors, which–with luck—will be mapped to the same hash table key. The hash table values associated with each key are retrieved, and each value that is found is considered a match. Each match votes for the correspondence between the interest points from the database and the interest points from the image that were used to define the geometric feature.

1.28 After all features from the image have been processed, many votes will have accumulated. False matches (due to aliasing or hash table collisions) should be uniformly distributed over the set of known objects, so should not generate very many votes for any particular object. Objects that are actually present in the image should generate many votes. Objects that receive an insufficient number of votes are rejected, and the remainder are subjected to a verification process.

1.29 The verification process requires computing a transformation from the object to image coordinates, using the corresponding interest points from the known object and the image. Using this transformation, all the interest points belonging to the known object are projected into image coordinates. If the match is correct, we expect the image to contain features at these locations. Some researchers add an extra verification stage by projecting the known object into the image and checking that the predicted edges of the object are found in the image (eg, [42, 34]).

1.30 The geometric hashing approach is flexible and applicable to many problems, but there are some issues with the basic version described here. The hash function described above simply places a grid over the relative coordinate system and returns the identifier of the grid cell in which the interest point lands. This is prone to edge effects: if an interest point is near the edge of a grid cell, a small change in its position due to noise

can move it across the edge into a different grid cell, which causes the hash code to change and a different set of feature matches to be found. This problem can be overcome by detecting interest points that are near the edges of their cells and also searching the hash table using the hash codes of neighbouring cells. This results in more false feature matches, and can become expensive as the dimensionality of the geometric feature space increases (there are more edges, and the proportion of the hypervolume of feature space that is near an edge increases). Using a data structure other than the hash table may be beneficial.

1.31 Another issue with uniform grid-based hashing is that it assigns equal importance (and storage space) to each grid cell in the feature space. Consider an interest point that is far from the basis set that defines its local coordinate system. Small positional errors in the basis set can cause the coordinate system to be rotated or scaled, which results in large changes in the interest point's relative coordinates. This means that edge effects become more pronounced in such regions of the feature space. One response to this concern is to increases the size of the grid cells far from the basis set, but then if the interest points are uniformly distributed, these larger hash table bins will contain more features, so every match to these bins will be accompanied by more false positives.

1.32 Finally, a hash table with equal-sized cells will only be uniformly utilized if interest points are uniformly distributed in feature space, but some domains may yield interest points that are far from uniformly distributed. In other domains, it may be beneficial to place constraints on the geometric features that are used (for example, to avoid regions that are prone to large errors, or that are relatively indistinctive).

1.33 There is nothing in the geometric hashing framework that requires a standard hash table to be used for feature matching. Other methods of feature matching are described in the next section.

1.7 Related work in fast feature matching

1.34 For feature matching in a geometric hashing framework, the database of known objects is converted into a set of points in a feature descriptor space, which is a vector space. To find matches to a new feature (extracted from an image in which we want to recognize objects), we need to find all points in the vector space that are within some tolerance. Usually the Euclidean distance metric is applied, so the tolerance is a radius in the vector space.

1.7.1 Bloom filters

1.35 A Bloom filter [12] is a data structure that can answer queries of the form "is feature q a member of the set of known features?" A Bloom filter is a set of n bits, initially zero. A set of k hash functions must be defined which take a feature as input and produce an integer in $[0, n)$ as output. In the indexing phase, we examine each of the known features, and apply each of the hash functions, producing k hash codes. Each code is used to reference a bit and *set* it (*i.e.*, turn it on). Each known feature results in k bits being set; several features may request that a particular bit be set.

1.36 Given a new feature to match, we run the k hash functions on it and test whether each of the resulting bits are set. If the feature is equal to a feature seen at indexing time (according to the hash functions), then all k of the bits will have been set; thus the Bloom filter does not produce false negatives. If the query feature is not equal to a known feature, then some of the bits may have been set due to *collision* of the hash functions, but the feature will only be accepted if all k hash functions result in collision; this results in a false positive. The rate of false positives increases as the number of known features increases relative to the number of bits in the filter. The Bloom filter is ideally suited to cases where the number of known features is small, and a verification step can be applied to eliminate any false positives that are produced.

1.37 For feature matching, we want to know not only whether the query feature q matches a known feature, we want to know *which* known feature matches. One way to answer such queries is to use a *Bloomier filter* [14]. The simplest Bloomier filter uses multiple Bloom filters to categorize a given feature into two different categories, and uses slightly more space than two Bloom filters. A set of n such Bloomier filters can be used to categorize a feature into 2^n different categories by assigning one Bloomier filter to each bit of the result.

1.38 The standard Bloom filter only finds exact matches, but we want to find *nearby* matches. An extension of Bloom filters to allow proximity searches is given by Kirsch and Mitzenmacher [40]. This variant uses *locality sensitive hashing* (LSH, discussed below) as its hashing function, so it inherits some limitations from LSH. Most importantly, LSH only guarantees with some probability that nearby features will hash to the same bin. As a result, the locality-sensitive Bloom filter can produce false negatives as well as false positives.

1.39 In order to answer feature matching queries (find all known features near a given query feature), one could combine these Bloom filter extensions to build a locality-sensitive Bloomier filter. However, the false negative rate of the locality-sensitive filter would be magnified because the Bloomier filter requires $2\lceil\log(n)\rceil$ separate filters to represent n known features, and all of these filters must produce the correct answer. It seems that Bloom filters are not particularly well suited to this task.

1.7.2 Locality Sensitive Hashing

1.40 Locality Sensitive Hashing (LSH) [16, 25, 3], is a technique for answering approximate nearest neighbour queries: it returns a feature whose distance is within a constant factor of the distance to the nearest neighbour. The LSH scheme can be extended to handle exact nearest-neighbour queries efficiently if the set of known features satisfies some *bounded growth* criteria. It is intended for use in high-dimensional spaces (tens to

thousands of dimensions), and ℓ_p distance metrics for $p \in (0, 2]$.

1.41 The core idea of LSH is to use several hash functions with the property that the probability of collision—features being placed in the same bin—is much higher for nearby features than for distant ones. In the indexing stage, we populate the hash table by running each hash function on each known feature. To perform a query, we run the hash functions on the query feature and retrieve the features in the same hash table bin as the query. One of these features is likely to be an approximate nearest neighbour of the query.

1.42 In practice, in [16] each hash function is composed of several (typically 10) simpler hash functions. Each simple hash function performs a random projection in the feature space: the hash value is the discretized value of the dot product of the feature vector with the hash function's random vector. Combining several of these simple hash functions is equivalent to projecting the feature vectors onto a grid in a 10-dimensional, non-orthogonal, subspace. The hash code is simply the identifier of the grid cell in which the feature falls.

1.43 This scheme suffers from edge effects: a query feature may land in a bin other than the bin containing its nearest neighbour due to small positional noise in any of the dimensions. Attempting to reduce this effect by making the bin size (discretization level) larger fails because the number of hash functions must be increased to maintain the same total number of hash bins.

1.44 In order to reduce the number of false negatives, we can choose many (typically 30) different hash functions and place a reference to each feature in the hash table bin chosen by each function. This increases the probability that at least one of the hash functions will select a 10-dimensional subspace in which the query is close to its nearest neighbour.

1.45 The standard LSH scheme does not seem particularly well suited for the purpose of feature matching in a geometric hashing system, since the dimensionality of the feature vectors is usually moderate (2 to 6 in the *Astrometry.net* system). Since some feature

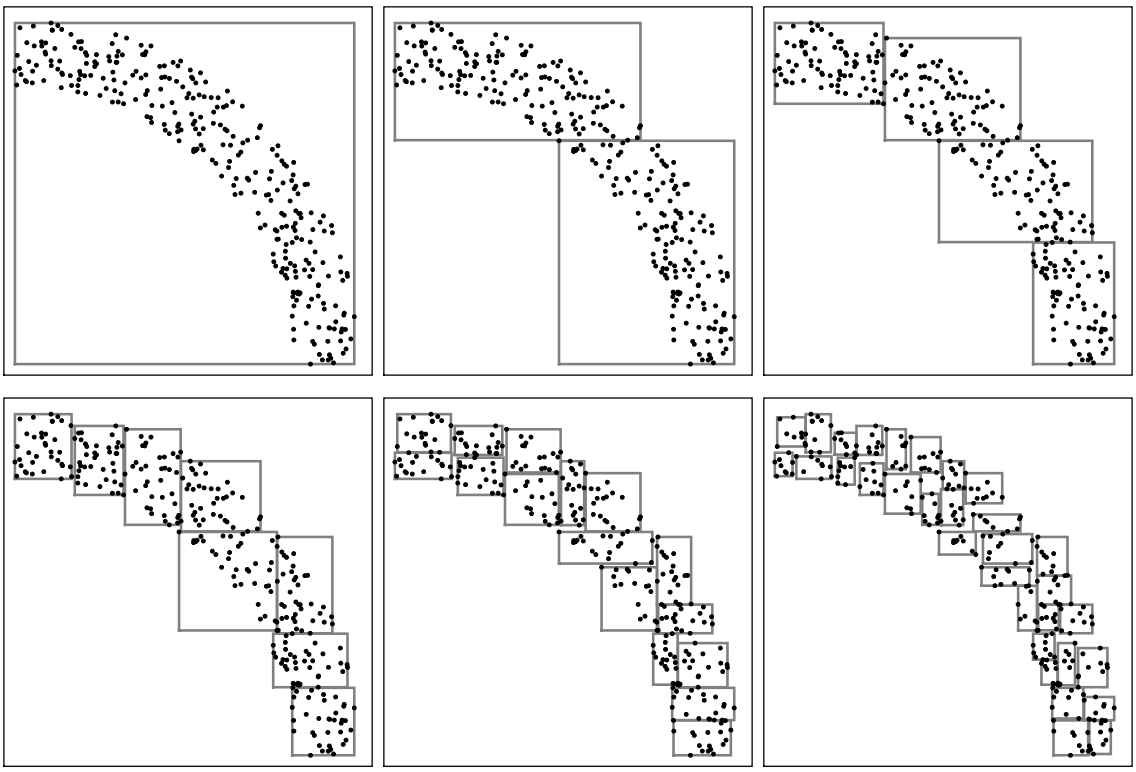


Figure 1.2: A kd-tree partitioning of space. Each panel shows the nodes at one level of the tree. The points are 200 samples from a ring with uniformly-distributed radius in $(0.8, 1)$ and uniformly-distributed angle in $\left(0, \frac{\pi}{2}\right)$. Nodes are split along the dimension with the largest range, and the splitting position is chosen so that the two child nodes will contain an equal number of points. Note how the tree adapts to the distribution of the data: each node tightly bounds the data points it owns. This is a slightly different version of the kd-tree than Bentley's original description.

aliasing is expected in geometric hashing systems, the nearest neighbour may not be the correct match: we would prefer to get all neighbours within a given search radius.

1.7.3 Kd-trees

1.46 The k-dimensional tree (kd-tree) was introduced by Bentley in 1975 [9], and several extensions have been described [24, 60]. The idea is to build a binary tree over the feature

space, where each node "owns" a region of the space which is disjoint from the region owned by its sibling node. Each non-leaf node has an axis-aligned splitting hyper-plane; its left child owns the subspace to the left of the splitting plane, and the right child the subspace to the right. In this way, the kd-tree defines a hierarchical subdivision of the feature space into axis-aligned hyper-rectangles. See figure 1.2.

1.47 For the purposes of indexing the features in a geometric hashing system, the set of features is known beforehand and is static, so we do not need to insert or delete features from the tree, and Bentley's *optimized* kd-tree construction method can be used. That is, the tree can be built to adapt to the particular set of features we have.

1.48 Tree construction proceeds by first assigning all features to the root node, then recursively splitting nodes until each leaf node owns at most L features, with L perhaps 10 or 20. Splitting is done by selecting a splitting plane and using it to partition the features. Sproull [60] enumerates several strategies for choosing the splitting hyperplane. Traditionally, kd-trees use axis-aligned splitting hyperplanes, and the splitting dimension is chosen by simply iterating through the dimensions (as in Bentley's original paper), or by selecting the dimension with the largest *range* or *variance*. One can also use an arbitrary splitting hyperplane, for example by finding the principal eigenvector of the covariance matrix of the feature vectors. Typically the splitting hyperplane is positioned so that the two children own an equal number of data points. This leads to a balanced (and therefore short) tree, but in some applications it can be beneficial to bias the splitting location [47].

1.49 Depending on the application, it can be beneficial to store at each node the tight axis-aligned bounding box (and other summary statistics [19]) of the points owned by that node (as shown in figure 1.2). This is particularly useful where there is significant correlation between the dimensions of the data, because splitting the data points along one dimension implies that the bounding box will shrink in other dimensions as well, and this can result in faster searches. If the bounding boxes are stored, then it is no longer

strictly necessary to store the splitting hyperplane, though it may be useful to speed up some computations.

1.50 For feature matching, we typically want to do *range search*: find all features within radius r of a given query feature q. This is easily achieved with a kd-tree using a recursive algorithm. Starting at the root, we must decide whether we need to recurse on each of the node's children. (How we make this decision is explained below.) Once we reach a leaf, we enumerate the features owned by the leaf and compute the distance to the query feature; any feature with distance less than r is accepted.

1.51 We decide whether we must recurse on a node's children based on their bounding boxes. We compute the *mindist*—the lower-bound of distance—between the query point and each child's bounding box. The mindist is simply the distance to the corner or edge of the bounding box, and can be computed quickly. Any node whose mindist to the query point is less than the query radius r must be visited. If the kd-tree does not explicitly store the tight bounding-boxes of the nodes, loose bounding-boxes can be constructed during the recursion based on the splitting planes of the ancestors.

1.52 Chapter 4 presents the kd-tree in more detail.

1.7.4 Other approaches

1.53 In contrast to the kd-tree's rectangular decomposition of space, there is a family of space decompositions that use hyperspheres. An example is the *ball-tree* [64], in which each non-leaf node is associated with one of the known features f and a radius r; the left child owns all features within radius r of feature f, and the right child owns the remaining features. Similarly, in the *anchors hierarchy* [53], each non-leaf node is represented by a feature (called its *anchor*), and a node owns all the features that are closer to its anchor than the anchor of its sibling. These sphere-based trees are supposed to better capture the structure of high-dimensional data sets, though the results seem to depend quite strongly on the particular data distributions and search parameters.

1.54 For two-dimensional feature spaces, algorithms based on planar point location (eg, [39]) and the Voronoi tesselation of space yield $\mathcal{O}(n \log n)$ preprocessing time, a data structure of size $\mathcal{O}(n)$, and query time of $\mathcal{O}(\log n)$. Unfortunately, in higher dimensions the space required to store the Voronoi tesselation is $\mathcal{O}(n^{\lfloor d/2 \rfloor})$, which quickly becomes untenable [4].

1.55 The design of spatial indexing data structures and search algorithms is itself a large research area. Dozens of different tree structures have been proposed: in two surveys Böhm [13] and Hjaltason [31] identify B-, B^+-, ball-, bisector-, BSP-, DABS-, fq-, gh-, GNA-, hB-, hB^π-, hybrid-, IQ-, kd-, kd-B-, LSD^h-, M-, mb-, mb^*-, mvp-, oct-, post-office-, pyramid-, quad-, R-, R^*-, R^+-, sa-, slim-, sphere-, SR-, SS-, TV-, vp-, vp^*-, and X-trees. There is an equally mind-boggling variety of exact and approximate search algorithms. Each data structure and algorithm may work well in some region of parameter space, but there is no clear winner for general-purpose use.

1.8 Astrometric calibration as a pattern recognition task

1.56 For modern astronomers, astrometric calibration is often one of the first steps toward getting useful information out of an image of the sky. Aligning a new image with an *astrometric reference catalog* allows the astronomer to place the image within a standard coordinate frame. This allows stars, galaxies, and other objects (*sources*) in the new image to be identified with known sources, which in turn allows astronomers to calibrate other properties of the new image, and allows the positions of new sources to be described in a standard reference frame.

1.57 The task of full-sky astrometric calibration—automatically finding the astrometric calibration of an image, using only the information in the image pixels—can be seen as a pattern recognition problem. As Bertin [10] notes, "astrometric and photometric

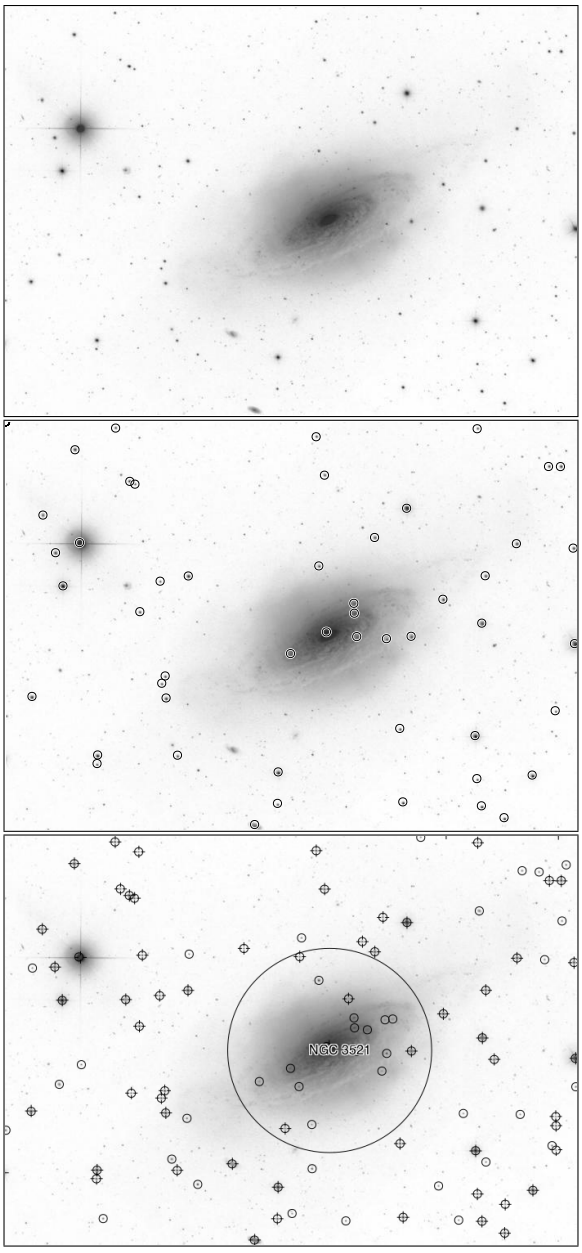


Figure 1.3: **Top:** Input image (credit: Sloan Digital Sky Survey). **Middle:** The brightest 100 sources extracted from the image. **Bottom:** Reference sources, transformed into the image coordinate system (crosshairs). Many of the image and reference sources are aligned, but there are many image sources without reference sources. Our system knows about the positions of many objects of interest on the sky, and has labelled the galaxy NGC 3521.

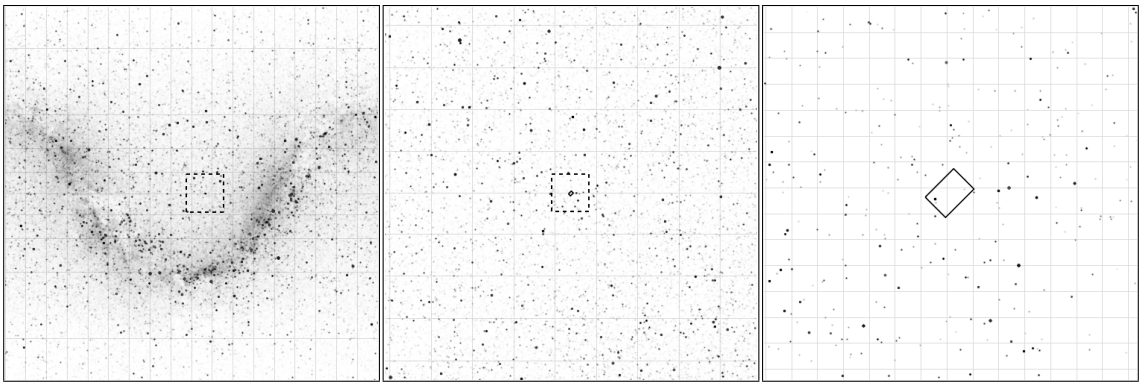


Figure 1.4: The location of the input image on the sky. **Left:** The whole sky, in Mercator projection. The sinusoid-shaped feature is the Milky Way. The dashed box shows the zoomed-in region. **Middle:** Zoomed in by a factor of 10. **Right:** Zoomed in by a factor of 100. The box shows the outline of the input image. These images are rendering of stars from the Tycho-2 catalog [32].

calibrations have remained the most tiresome step in the reduction of large imaging surveys," so this is not only an interesting problem to solve, but one with practical implications for astronomers.

1.58 For the purposes of astrometric calibration, we can think of the sky as a large two-dimensional surface: the stars are very distant, so our viewpoint is effectively fixed. We are moving, as are the stars, but these motions are small relative to the precision at which we typically work. The sky contains many stars, galaxies, and other astronomical sources. The stars and distant galaxies are effectively point sources, while closer galaxies can be resolved. Astrometric reference catalogs list the positions, motions, and brightnesses of these sources and serve as the "ground truth" or database of known (reference) objects. The USNO-B1 catalog [52, 70], for example, lists over one billion objects. As many as a few percent of these are false detections or other artifacts [7], and some objects that should be visible are missing.

1.59 Extra sources can be due to planets, comets, satellites, or aircraft. Missing or poorly localized source can be due to imperfections in the imaging sensor, saturation, cosmic ray

interference, or (rarely) occlusion. Errors in the image processing that detects sources can lead to sources being gained or lost. We call sources that appear in only the input image or reference catalog *distractors* and *dropouts*, respectively. The existence of distractors and dropouts means that we can never assume that all the objects in the reference catalog will be contained in an image to be recognized, or vice versa.

1.60 The images to be recognized are subregions of the sky. Image sizes range from nearly half the celestial sphere down to 10^{-7} of the area and smaller. The input images measure unknown bands of the electromagnetic spectrum, and various nonlinear functions may have been applied to the pixel values. We cannot rely on absolute brightness or color to recognize individual stars or galaxies. At best we can hope that there is some positive correlation in the relative brightness ordering of objects in the image and the corresponding objects in our catalog.

1.61 Astrometric calibration is an ideal task for exploring geometric ideas in pattern recognition. Most celestial objects are effectively point sources, and can be found and localized to sub-pixel accuracy using relatively simple image-processing procedures. But since the individual features are characterized only by their positions and brightnesses, we must examine collections of features in order to build distinctive patterns. In chapter 2 we present *Astrometry.net*, which applies the geometric hashing framework to the task of astrometric calibration. An example of our results in shown in figures 1.3 and 1.4.

1.9 Related work in astrometric calibration

1.62 There seem to be two distinct groups of researchers who have worked on astrometric calibration. The first are professional astronomers, whose images are typically of small angular extent, long exposure time, and high quality. They typically assume that there is a good initial guess of the astrometric calibration and the goal is to produce a very accurate calibration, including image distortion. The second group of researchers are

spacecraft engineers who want to use the stars to estimate the attitude of a camera attached to a spacecraft. Here the images are of wide angular extent, have short exposure time, and are very noisy. Primary concerns include weight, power consumption, and robustness (especially in avoiding false positives), while a high degree of accuracy is neither required nor possible given the hardware.

1.9.1 Ballpark astrometric calibration

1.63 "Ballpark" astrometric calibration (as opposed to "full-sky") requires that an initial estimate of the calibration to be provided. Groth [28] presents an algorithm for matching two lists of coordinates (eg, image coordinates in pixels and reference star coordinates on the celestial sphere), assuming that the lists contain a significant proportion of objects in common. With the limited computing resources available at the time, he suggests taking the brightest 20 or 30 objects in each list.

1.64 Once the two lists of objects have been compiled and the brightest objects selected, all sets of three objects are enumerated and the triangles they form are described by a scale-, rotation-, and translation-invariant descriptor, composed of the length ratio of the longest to shortest edges, plus the cosine between these edges and the *sense* or *parity* of the triangle. The tolerances associated with these features are computed (by propagating their positional errors through the feature descriptor process) and stored, along with the logarithm of the triangle's perimeter. Triangles with large length ratio are rejected since they are relatively indistinctive.

1.65 After triangle features are extracted from both point lists, feature matching is performed by checking whether the distance between each pair of features is less than their corresponding tolerances. This process is accelerated by sorting the features on one of the feature dimensions. If multiple matches are found for a particular feature, only the closest is considered. After all the features have been compared, many correct matches should be found, along with some false matches. For each match, the difference of the

log-perimeters of the two triangles is computed. This gives the relative scales of the two triangles and hence their coordinate frames, assuming the match is correct. False matches are rejected by iterative outlier detection: the mean difference of log-perimeters is computed and matches far from the mean are rejected.

1.66 This approach is essentially an application of the geometric hashing method, though instead of using hashing to accelerate feature matching, the features are simply kept in lists that are sorted on one dimension. Much of the subsequent work in this area follows essentially the same path (eg, [65], [21]).

1.67 Pál and Bakos [57] adapt the triangle-matching approach to images containing many more objects (of order 10^4). Since it would become prohibitively expensive to enumerate all triangles in such an image, they reduce the number of triangles created by using only the triangles created by a Delauney triangulation. This vastly reduces the number of triangles created, but makes the algorithm more sensitive to dropout and distractor stars: a single extra star causes a completely disjoint set of triangles to be created in its neighbourhood. To compensate for this shortcoming, they define an *extended Delauney triangulation*: for each point, they select all points at distance ℓ in the Delauney triangulation, and triangulate this set. This process is repeated for each point. The first extended triangulation ($\ell = 2$) skips over the nearest set of stars and builds larger triangles by using the next-further set of stars. The intent is that if some of the nearby stars are distractors that they will be ignored when the extended triangulations are used.

1.68 Pál and Bakos introduce a new triangle parameterization which is continuous and sensitive to chirality (parity); see figure 1.5. This two-dimensional parameter space is used as the geometric feature space. When matching triangles between the two images, they demand a *symmetric point match*: each triangle must be the other triangle's nearest neighbour in feature space. Matching two images is done by creating the lists of triangles and attempting to find symmetric point matches. This process is accelerated by sorting each list by one of the coordinates. Each triangle match is considered to be a vote for

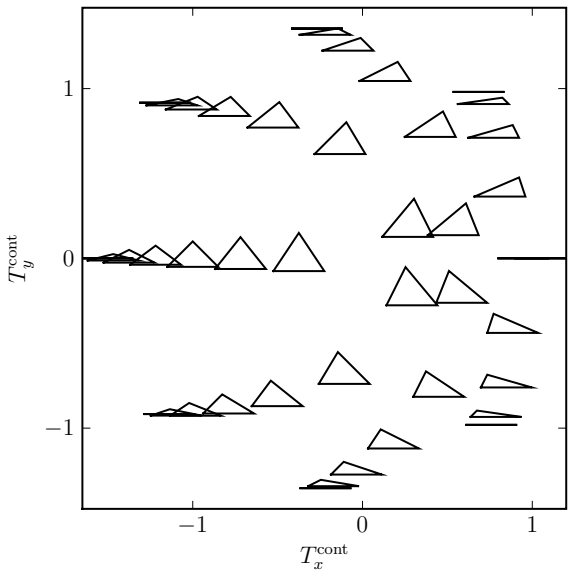


Figure 1.5: The triangle parameterization used by Pál and Bakos [57].

the correspondence of the three pairs of points composing the triangles. These votes are accumulated in a sparse matrix where element (i, j) contains the number of votes for a correspondence between object i in the first image and object j in the second image. After all matches are considered, the top 40% of the nonzero matrix elements are assumed to contain the true correspondences. A transformation based on these correspondences is computed and the unitarity of the transformation matrix is used to judge whether the match is true or false.

1.69 A different approach is taken by Kaiser *et al.* [38]. They assume they are given two lists of source positions that differ by a rigid transformation involving scaling, rotation, and translation. In each image, they iterate over each pair of points and compute the vector difference of their positions. For each list, the log-length and angle of the pairwise difference vectors are accumulated in a two-dimensional histogram. Observe that if the whole list of points is scaled up by a constant factor, then the log-distance between each pair of points increases by a constant amount. Similarly, if the whole list is rotated then the angles shift by a constant amount. Once the two histograms have been computed,

their cross-correlation is computed. If the two lists contain a significant number of corresponding points, the cross-correlation signal will be strongest at a shift corresponding to the difference in log-scale and rotation between the lists. This process is similar to a Hough transform [22, 5], except that instead of finding the peak of a single parameter-space histogram, we are searching for a peak in the similarity of two histograms as we shift them with respect to each other in parameter space.

1.70 Once the scaling and rotation between the lists has been found, the translation can be found by scaling and rotating one of the lists into the frame of the other list, then histogramming the vector difference between points across the two lists. This is a standard (generalized) Hough transform.

1.9.2 Full-sky astrometric calibration

1.71 The majority of previous work on full-sky astrometric calibration is motivated by the problem of spacecraft attitude estimation. Sometimes called the "lost in space" or "stellar gyroscope" problem, the task is to estimate the pose of a spacecraft by using an image of the sky captured by an onboard camera. Although similar in general spirit, the requirements and limitations of this application are quite different than astronomical applications. Mass, power consumption, and robustness of the system are primary concerns, and as a result the optical designs are very different from science-grade astronomical instruments, and the available processor and memory resources are very restricted. Typical fields of view are tens of degrees across, and the exposure time is kept short to allow the system to function while the spacecraft is rotating. As a result, image quality is typically quite poor: often only a handful of the brightest stars are visible. Since the field of view is large, a reference catalog of a few thousand stars is sufficient to ensure that any view of the sky contains many reference stars. Since the system will only process images from a single camera, the whole system can be customized and calibrated to that camera. For example, the nonlinear distortions of the optical system can be measured, and the scale

and bandpass of the imaging system are known, so the reference catalog can be tailored to match.

1.72 For example, Liebe *et al.* [45] describe a system design with a 56 degree field of view and exposure time of 50 ms. The resulting images contain tens of stars if the spacecraft is not rotating, but on average only three stars will be detectable when the rotation rate is 50 degree/sec. The paper does not describe a particular algorithm for star identification, with the implication that it is not a particularly difficult problem since absolute brightness information will be available, and the total number of stars that are visible to the camera is only a few hundred.

1.73 In earlier work [44], Liebe describes a star identification system. The reference catalog is composed of the 1539 brightest stars, with brightness calibrated to the camera used in the system. The field of view is 30 degrees, and the system uses a feature-matching approach, using the brightest star in the field and its two nearest neighbours to define a geometric feature. The feature descriptor is composed of the distance from the brightest star to its nearest and second-nearest neighbours, along with the angle between these neighbours. Note that this feature is not scale-invariant: the system is only meant to recognize images taken by one camera, so the scale is known. An index is constructed by enumerating all such features that could possibly be detected, given the detection limits of the camera. These features are coarsely quantized and stored in a table. To account for noise in the feature descriptors, all neighbouring cells in the quantized feature space are also stored in the table. This generates $185,000$ features. At test time, the features in the image are enumerated and the table of features is scanned; an exact feature match in the quantized space is assumed to be correct.

1.74 This approach is a fairly straightforward geometric hashing technique, except that it does not use hashing as such, and there is no voting or verification scheme because feature aliasing is assumed not to happen.

1.75 In contrast to systems that build features from the precise locations of a small number

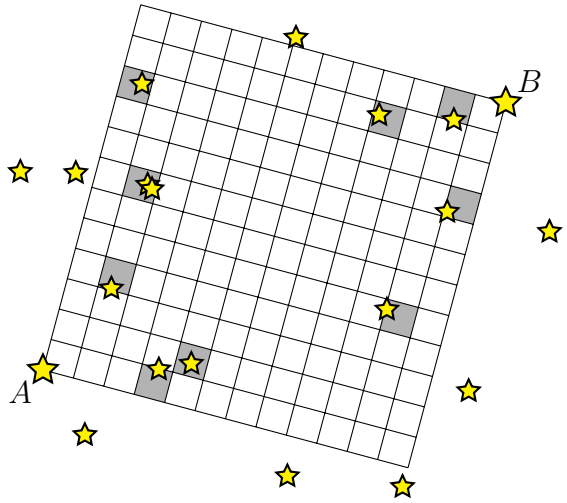


Figure 1.6: A grid-based feature: two sources (labelled A and B here) are used to define a local coordinate system which is discretized into a grid of cells. Each cell becomes a bit in the feature descriptor; if a cell is occupied by a star its bit is set. This figure is inspired by MacKay & Roweis [48].

of stars, Padgett and Kreutz-Delgado [56] present an approach that incorporates information from a large portion of the image. Their grid-based feature is defined by first choosing one star as the reference star to define the center of the grid, then selecting its nearest neighbour (outside an exclusion radius) to define the orientation of the grid. Once the center and orientation are determined, a grid is defined, and each remaining stars in the image is assigned to a grid cell. The feature is a bit vector, one bit per grid cell, where the bit is set only if the cell contains a star. See figure 1.6.

1.76 The index is constructed by scanning the sky and selecting, for each field of view, the n brightest stars. For each of these stars a feature is computed and stored. At test time, they examine the brightest cn stars (for some safety factor c), computing the feature for each one and searching the index for a match. A pair of features is defined to match if their dot product is above a threshold. This is equivalent to taking the bitwise AND of the bit vectors and counting the number of bits that are set. This search can be implemented efficiently by using lookup tables: for each bit, they maintain a list of the features for which that bit is set. Note that this is equivalent to using a grid-based geometric hashing approach: each grid cell is equivalent to a discretized relative coordinate vector, which becomes a hash key, and the "lookup table" is a hash table. After all the features in the test image have been extracted and matched to the index, the algorithm proceeds to a voting (or perhaps "consensus") phase where it rejects matches that propose a field of view that does not overlap that of the majority.

1.77 The system is designed to operate on images of diameter 8 degrees, and performs well on simulated data. They use a grid size of 40×40, and find that on average 25 grid cells are filled. Their index contains $13,000$ patterns, which means that false positives are quite rare. However, misidentification of the nearest neighbour, or edge effects (assigning a star to the wrong grid cell due to positional noise) mean that failures to find a match are not uncommon, and are more likely in regions of the sky with high stellar density.

1.78 In later work, Clouse and Padgett [15] extend this approach by using Bayesian decision

theory instead of a simple threshold on the dot product to define how well features match. This extension, along with smaller noise levels in the (simulated) imaging system, allows them to extend the approach down to fields of view 2 degrees in diameter.

1.79 Harvey [30] presents two different approaches, one grid-based and the other shape-based. The grid-based approach is similar to the Padgett–Kreutz-Delgado and Clouse–Padgett approaches [56, 15]. A coarser grid is used, so more stars are likely to appear in each bin. To compensate, a grid cell is only considered "occupied" if it contains more than some threshold number of stars. The other major change is that he expects a test image to be an "overexposure" or "underexposure" relative to the reference catalog. This implies that the test image should contain either a subset or a superset of the stars in the index, and therefore the image feature vector must be either greater than or less than an index feature at each bit. This allows simple bit operations to be used to find feature matches, and feature matching is performed by a linear scan through the index.

1.80 Harvey's shape-based approach uses constrained n-star constellations in order to aggregate information from a large number of stars without allowing the number of potential features to grow combinatorially. Specifically, Harvey uses a pair of stars to define a narrow wedge in the image, then describes the relative positions and angles of a fixed number of nearby stars within that wedge. Unfortunately, this makes the feature highly sensitive to distractor and dropout stars, since the feature depends on the stars in the feature being enumerated in a particular order. As a result, all potential features (allowing any combination of stars to drop out) must be checked; the number of features grows exponentially as the density of stars increases.

1.81 Harvey makes the useful observation that a cascade of indices can be built, where each index is designed to recognize images with a particular range of scales.

1.82 MacKay and Roweis [48] point out that a grid-based feature such as that used by Harvey leads naturally to a hashing-based strategy. Each grid cell is associated with a bit that is turned on if the cell is occupied. This value is placed in a hash table with a

mapping back to its position on the sky. Since false positives can occur as a result of feature aliasing or hash collision, a voting scheme is employed: several feature matches must accumulate before the match is accepted. Since false negatives can occur as a result of dropouts and distractors (*any* missing or extra star causes the hash code to change completely), many features must be extracted for each region of the sky.

1.9.3 Fine-tuning astrometric calibrations

1.83 In order to "co-add" or "stack" astronomical images (combine pixels from different images to produce a higher signal-to-noise image), or to do proper-motion studies (measure the movement of stars over time), it is necessary to fine-tune the astrometric calibration of the images. This is similar to the *bundle adjustment* problem in computer vision [63], in that it involves the simultaneous optimization of the various camera and telescope parameters and the estimated positions of objects in the world. Fine-tuning astrometric calibrations is easier because it is essentially two-dimensional, but more difficult because the camera parameters include polynomial distortion terms to model the image distortion introduced by telescope optics.

1.84 The software package `Scamp` by Bertin [10] is a popular tool used by astronomers to fine-tune simultaneously the astrometric calibrations of a large collection of images. For each image, it is assumed that the center of the image is known to about 25% of the size of the image, and the scale is known to within a factor of two. The histogram-alignment method of [38] is used to find the translation, scaling, and rotation between the image and a reference catalog, which allows the correspondences between image and reference catalog stars to be determined.

1.85 The core of the `Scamp` system is a chi-squared minimization of the total weighted distance between the projected positions of all star correspondences among the set of images and the reference catalog. The parameters to be adjusted are the center, scale, rotation, and polynomial distortion coefficients of each image. A key feature is the ability

to share image distortion parameters among subsets of the images, since images taken with the same telescope are expected to share some distortion terms due to the telescope optics. This reduces the degrees of freedom of the fitting process, resulting in more robust fits. `Scamp` can fine-tune thousands of images to sub-pixel accuracy.

1.10 Summary

1.86 The task of automatically finding the astrometric calibration of an astronomical image—this is, recognizing the stars in the image, or locating the image on the sky—is ideal for a geometric hashing-based approach. Geometric hashing (or more generally, geometric feature matching) is a widely-applicable technique for problems in which it is desirable to build distinctive geometric features out of simple interest-point features. The approach is ideally suited to the astrometric calibration problem, since images of celestial objects can be localized to sub-pixel accuracy but have few other distinguishing features. Indeed, much of the previous work in "full-sky" and "ballpark" astrometric calibration can be placed within a geometric feature matching framework.

Chapter 2

Astrometry.net: recognition of arbitrary astronomical images[1]

[1]This chapter was originally prepared as a manuscript by me, David W. Hogg, Keir Mierle, Michael Blanton, and Sam Roweis.

2.1 Introduction

2.1 Although there are hundreds of ground- and space-based telescopes currently operating, and petabytes of stored astronomical images (some fraction of which are available in public archives), most astronomical research is conducted using data from a single telescope. Why do we as astronomers limit ourselves to using only small subsets of the enormous bulk of available data? Three main reasons can be identified: we don't want to share our data; it's hard to share our data; and it's hard to use data that others have shared. The latter two problems can be addressed by technological solutions, and as sharing data becomes easier, astronomers will likely become more willing to do so.

2.2 Historically, astronomical data—and, indeed, important scientific results—have often been closely guarded. In more recent times, early access to data has been seen as one of the rewards for joining and contributing to telescope-building collaborations, and proprietary data periods typically accompany grants of observing time on observatories such as the Hubble Space Telescope. However, this seems to be changing, albeit slowly. One of the first large astronomical data sets to be released publicly in a usable form was the Hubble Deep Field [66]. The Sloan Digital Sky Survey (SDSS) [69] is committed to yearly public data releases, and the members of upcoming projects such as the Large Synoptic Survey Telescope (LSST) [35] have recognized that the primary advantage of contributing to the collaboration is not proprietary access to the data, but rather a deep understanding and familiarity with the telescope and data, and have (in principle) decided to make the data available immediately.

2.3 Putting aside the issue of *willingness* to share data, there are issues of our *ability* to share data effectively. Making use of large, heterogeneous, distributed image collections requires fast, robust, automated tools for calibration, vetting, organization, search and retrieval.

2.4 The Virtual Observatory (VO) [61] establishes a framework and protocols for the organization, search, and retrieval of astronomical images. The VO is structured as a

distributed system in which many "publishers" provide image collections and interfaces that allow these images to be searched. This distributed framework allows the VO to scale, but it also means that any property we might want the images published through the VO to have must be specified by the standards, and we must trust all publishers to implement the standards correctly. Of particular importance are calibration meta-data [58]. The draft Simple Image Access Protocol [62] states that "an image should be a calibrated object frame" and specifies some loose requirements for astrometric meta-data. However, there is neither a requirement, nor a specified method, for communicating more detailed information about the calibration processes that have been applied to the image. A VO user who wants to know exactly how the raw CCD frame was reduced to the pixel values in the image must use processes outside the VO framework—most likely by reading papers and contacting the image publisher—and this will take much longer than finding and retrieving the image.

2.5 The VO cannot be expected to impose a minimum standard of "quality" on images published through VO protocols, for several reasons. First, doing so would require making a tradeoff between the quality and quantity of images that are publishable. Since different users of the VO have different needs, there is no objective way to make this tradeoff. For example, one researcher might only want images taken during photometric conditions, while one studying a transient event might want all available imaging, regardless of quality. Second, there is no objective measure of the quality of an image: different aspects of an image are important to different users. Finally, even the most well-intentioned and skilled publisher will occasionally make mistakes, and this effect will become more pronounced as surveys become more automated and data rates increase. Thus, users of the VO cannot in general rely on the quality or correctness of image data or calibration meta-data, and individually hand-checking each image does not scale to the future in which the VO provides access to any significant fraction of the world's astronomical images. For the goals of the VO movement to be achieved, the tools that allow users

to vet, verify, and recalibrate images must be developed, and ideally these tools will be integrated into the VO system.

2.6 In this chapter we present a system, *Astrometry.net*, that automatically produces astrometric meta-data for astronomical images. That is, given an image, our system produces the pointing, scale, and orientation of the image—the astrometric calibration meta-data or World Coordinate System (WCS). The system requires no first guess, and works with the information in the image pixels alone. The success rate is above 99.9% for contemporary near-ultraviolet and visual imaging survey data, with no false positives.

2.7 Our system enables an immense amount of "lost" astronomical imagery to be used for scientific purposes. This includes photographic plate archives, an immense and growing number of images taken by amateur astronomers, as well as data from individual professional astronomers and ground-based observatories whose meta-data are non-existent, lost, or simply wrong. While many modern telescopes do produce correct, standards-compliant meta-data, many others have control systems that drift relative to the sky, yielding only approximate astrometric meta-data. Still others produce no meta-data, or produce it in some ideosyncratic, non-standards-compliant form. Even sophisticated and highly-automated surveys such as SDSS occasionally have failures in the systems that produce astrometric calibration information, resulting in perfectly good but "lost" images. A system that makes these data available for study will effectively recover a significant amount of lost observing time, fill in gaps in the astronomical record, and make observers more productive by eliminating the tedious and often unenlightening task of fixing the astrometry. Furthermore, a robust, fully-automated system allows the data to be trusted, because the calibration meta-data, and their associated error estimates, have been derived from the images themselves, not from some unknown, undocumented, unverified or untrustworthy external source.

2.8 Our system can be seen as a specialized kind of image-based search. Given an image, we can identify and label the objects that appear in the image, with a very high success

rate and no false positives. We have achieved the ultimate goal of computer vision, within the domain of astronomical images. Our system is based solely on the contents of the image, in sharp contrast to most contemporary image search systems (such as Google Image Search), which rely on contextual information—the text surrounding the image on a web page—rather than the information in the image itself.

2.9 In the literature, the task of recognizing astronomical images is known as the "lost in space" problem, since an early application was for estimating the attitude of a spacecraft using a camera mounted on the spacecraft. By identifying the stars that are visible, the pose of the camera can be determined [44]. In such systems, triangles of stars are typically used as geometric features (e.g. [37]). Triangles are effective in this regime because the images typically span tens of degrees and contain only dozens of very bright stars: the search space is small. Furthermore, these systems use significant prior information, in that they are designed for particular cameras, the specifications of which are available to the system designers.

2.10 Triangle-based approaches have also been used to fine-tune the astrometry problem when a good initial estimate is available [57]. Because the search domain is limited to a small area around the initial estimate, the triangle-based approach is effective. Full-sky systems have been attempted previously (for example, [30] and references therein) but none we know of have been able to achieve the scalability and fidelity of our approach.

2.11 Both the triangle matching approach and ours (described below) are based on "geometric hashing" (for example, [42], [34]). Our system uses the same two-step approach used by these systems, in which a set of hypotheses are generated from sparse matches, then a second stage does detailed verification of the hypotheses.

2.12 There are several automated calibration systems that refine the astrometric calibration of an image to produce a high-precision alignment to a reference catalog given a good first guess (for example, [65, 51, 10]). These systems are reliable and robust, but they require a reasonable first guess about the image pointing, orientation, and scale.

Our system can be used to *create* that good first guess.

2.2 Methods

2.13 Our approach involves four main components. First, when given a query image, we detect astronomical sources ("stars", hereafter) by running a number of image-processing steps. This typically yields a few hundred or more stars localized to sub-pixel accuracy. Next, the system examines subsets of these stars, producing for each subset a geometric hash code that describes their relative positions. We typically use subsets of four stars, which we call "quads." Having computed the hash code for the query quad, the system then searches in a large pre-computed index for almost identical hash codes. Each matching hash code that is found corresponds to a hypothesized alignment between the quad in the query image and the quad in the index, which can be expressed as a hypothesized location, scale, and orientation of the image on the sky. The final component is a verification criterion, phrased as a Bayesian decision problem, which can very accurately decide if the hypothesized alignment is correct. The system continues generating and testing hypotheses until we find one that is accepted by the verification process; we then output that hypothesis as our chosen alignment. In some cases, we never find a hypothesis in which we are sufficiently confident (or we give up searching before we find one), but our thresholds are set conservatively enough that we almost never produce a false positive match. Each component of our approach is outlined below.

2.14 The primary technical contributions of our system include the use of the "geometric hashing" [42, 67] approach to solve the huge search problem of generating candidate calibrations; an index-building strategy that takes into account the distribution of images we wish to calibrate; the verification procedure which determines whether a proposed astrometric calibration is correct; and good software engineering which has allowed us to produce a practical, efficient system. All our code is publicly available under the GPL

license, and we are also offering a web service.

2.2.1 Star detection

2.15 The system automatically detects compact objects in each input image and centroids them to yield the pixel-space locations of stars. This problem is an old one in astronomy; the challenge *here* is to perform it robustly, with no human intervention, on the large variety of input images the system faces. Luckily, because the rest of the system is so robust, it can handle detection lists that are missing a few stars or have some contaminants.

2.16 The first task is to identify (detect) localized sources of light in the image. Most astronomical images exhibit sky variations, sensitivity variations, and scattering; we need to find peaks on top of such variations. First we subtract off a median-smoothed version of the image to "flatten" it. Next, to find statistically significant peaks, we need to know the approximate noise level. We find this by choosing a few thousand random pairs of pixels separated by five rows and columns, calculating the difference in the fluxes for each pair, and calculating the variance of those differences, which is approximately twice the variance σ^2 in each pixel. At this point, we identify pixels which have values in the flattened image that are $> 8\,\sigma$, and connect detected pixels into individual detected objects.

2.17 The second task is to find the peak or peaks in each detected object. This determination begins by identifying pixels that contain larger values than all of their neighbors. However, keeping all such pixels would retain peaks that are just due to uncorrelated noise in the image and individual peaks within a single "object." To clean the peak list, we look for peaks that are joined to smaller peaks by saddle points within $3\,\sigma$ of the larger peak (or 1 % of the larger peak's value, whichever is greater), and trim the smaller peaks out of our list.

2.18 Finally, given the list of all peaks, the third task is to centroid the star position at sub-pixel accuracy. Following previous work [46], we take a 3×3 grid around each star's

peak pixel, effectively fitting a Gaussian model to the nine values, and using the peak of the Gaussian. Occasionally this procedure produces a Gaussian peak outside the 3×3 grid, in which case we default to the peak pixel, although such cases are virtually always caused by image artifacts. This procedure produces a set of x and y positions in pixel coordinates corresponding to the position of objects in the image.

2.19 Compared to other star detection systems such as SExtractor [11], our approach is simpler and generally more robust given a wide variety of images and no human intervention. For example, while we do a simple median-filtering to remove the background signal from the image, SExtractor uses sigma-clipping and mode estimation on a grid of subimages, which are then median-filtered and spline-interpolated.

2.2.2 Hashing of asterisms to generate hypotheses

2.20 Hypotheses about the location of an astronomical image live in the continuous four-dimensional space of position on the celestial sphere (pointing of the camera's optical axis), orientation (rotation of the camera around its axis), and field of view (solid angle subtended by the camera image). We want to be able to recognize images that span less than one-millionth the area of the sky, so the effective number of hypotheses is large; exhaustive search will be impractical. We need a fast search heuristic: a method for proposing hypotheses that almost always proposes a correct hypothesis early enough that we have the resources to discover it.

2.21 Our fast search heuristic uses a continuous geometric hashing approach. Given a set of stars (a "quad"), we compute a local description of the shape—a geometric hash code—by mapping the relative positions of the stars in the quad into a point in a continuous-valued, 4-dimensional vector space ("code space"). Figure 2.1 shows this process. Of the four stars comprising the quad, the most widely-separated pair are used to define a local coordinate system, and the positions of the remaining two stars in this coordinate system serve as the hash code. We label the most widely-separated pair of stars "A" and "B".

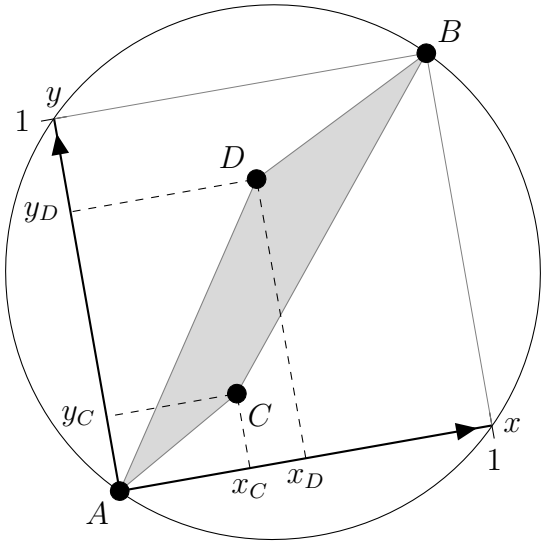


Figure 2.1: The geometric hash code for a "quad" of stars, A, B, C, and D. Stars A and B define the origin and $(1, 1)$, respectively, of a local coordinate system, in which the positions of stars C and D are computed. The coordinates (x_C, y_C, x_D, y_D) become our geometric hash code that describes the relative positions of the four stars. The hash code is invariant under translation, scaling, and rotation of the four stars.

These two stars define a local coordinate system. The remaining two stars are called "C" and "D", and their positions in this local coordinate system are (x_C, y_C) and (x_D, y_D). The geometric hash code is simply the 4-vector (x_C, y_C, x_D, y_D). We require stars C and D to be within the circle that has stars A and B on its diameter. This hash code has some symmetries: swapping A and B converts the code to $(1-x_C, 1-y_C, 1-x_D, 1-y_D)$ while swapping C and D converts (x_C, y_C, x_D, y_D) into (x_D, y_D, x_C, y_C). In practice, we break this symmetry by demanding that $x_C \leq x_D$ and that $x_C + x_D \leq 1$; we consider only the permutation (or relabelling) of stars that satisfies these conditions (within noise tolerance).

2.22 This mapping has several properties that make it well suited to our indexing application. First, the code vector is *invariant* to translation, rotation and scaling of the star positions so that it can be computed using only the relative positions of the four stars in any conformal coordinate system (including pixel coordinates in a query image). Second, the mapping is *smooth*: small changes in the relative positions of any of the stars result in small changes to the components of the code vector; this makes the codes resilient to small amounts of positional noise in star positions. Third, if stars are uniformly distributed on the sky (at the angular scale of the quads being indexed), codes will be uniformly distributed in (and thus make good use of) the 4-dimensional code-space volume.

2.23 Noise in the image and distortion caused by the atmosphere and telescope optics lead to noise in the measured positions of stars in the image. In general this noise causes the stars in a quad to move slightly with respect to each other, which yields small changes in the hash code (*i.e.*, position in code space) of the quad. Therefore, we must always match the image hash code with a *neighborhood* of hash codes in the index.

2.24 The standard geometric hashing "recipe" would suggest using triangles rather than quads. However, the positional noise level in typical astronomical images is sufficiently high that triangles are not distinctive enough to yield reasonable performance. An im-

portant factor in the performance of geometric hashing system is the "oversubscription factor" of code space. The number of hash codes that must be contained in an index is determined by the effective number of objects that are to be recognized by the system: if the goal is to recognize a million distinct objects, the index must contain at least a million hash codes. Each hash code effectively occupies a volume in code space: since hash codes can vary slightly due to positional noise in the inputs (star positions), we must always search for matching codes within a volume of code space. This volume is determined by the positional noise levels in the input image and the reference catalog. The oversubscription factor of code space, if it is uniformly populated, is simply the number of codes in the index multiplied by the fractional volume of code space occupied by each code. If triangles are used, the fractional volume of code space occupied by a single code is large, so the code space becomes heavily oversubscribed. Any query will match many codes by coincidence, and the system will have to reject all of these false matches, which is computationally expensive. By using quads instead of triangles, we nearly *square* the distinctiveness of our features: a quad can be thought of as two triangles that share a common edge, so a quad essentially describes the co-occurrence of two triangles. A much smaller fraction of code space is occupied by each quad, so we expect fewer coincidental (false) matches for any given query, and therefore fewer false hypotheses which must be rejected.

2.25 We could use quintuples of stars, which are even more distinctive than quads. However, there are two disadvantages to increasing the number of stars in our asterisms. The first is that the probability that all k of the stars in an indexed asterism appear in the image and that all k of the stars in a query asterism appear in the index both decrease with increasing k. For images taken at wavelengths far from the catalog wavelength, or shallow images, this consideration can become severe. The second disadvantage is that near-neighbor lookup, even with a kd-tree, becomes more time-consuming with increasing dimensionality. The dimensionality of the code space for quintuples is 6-dimensional,

compared to the 4-dimensional code space of quads. We test triangle- and quintuple-based indices in section 2.3.1.7 below.

2.26 When presented with a list of stars from an image to calibrate, the system iterates through groups of four stars, treating each group as a quad and computing its hash code. Using the computed code, we perform a neighborhood lookup in the index, retrieving all the indexed codes that are close to the query code, along with their corresponding locations on the sky. Each retrieved code is effectively a *hypothesis*, which proposes to identify the four reference catalog stars used to create the code at indexing time with the four stars used to compute the query code. Each such hypothesis is evaluated as described below.

2.27 The question of which hypotheses to check and when to check them is a purely heuristic one. One could chose to wait until a hypothesis has two or more "votes" from independent codes before checking it or check every hypothesis as soon as it is proposed, whichever is faster. In our experiments, we find that it is faster, and much less memory-intensive, to simply check every hypothesis rather than accumulate votes.

2.2.3 Indexing the sky

2.28 As with all geometric hashing systems, our system is based around a pre-computed index of known asterisms. Building the index begins with a reference catalog of stars. We typically use an all-sky (or near-all-sky) optical survey such as USNO-B1 [52, 7] as our reference catalog, but we have also used the infrared 2MASS catalog [59] and the ultraviolet catalog from GALEX [49], as well as non-all-sky catalogs such as SDSS. From the reference catalog we select a large number of quads (using a process described below). For each quad, we store its hash code and a reference to the four stars of which it is composed. We also store the positions of those four stars. Given a query quad, we compute its hash code and search for nearby codes in the index. For each nearby code, we look up the corresponding four stars in the index, and create the hypothesis that

the four stars in the query quad correspond to the four stars in the index. By looking up the positions of the query stars in image coordinates and the index stars in celestial coordinates, we can express the hypothesis as a pointing, scale, and rotation of the image on the sky.

2.29 In order for our approach to be successful, our index must balance several properties. We want to be able to recognize images from any part of the sky, so we want to choose quads uniformly over the sky. We want to be able to recognize images with a wide range of angular sizes, so we want to choose quads of a variety of sizes. We expect that brighter stars will be more likely to be found in our query images, so we want to build quads out of bright stars preferentially. However, we also expect that some stars, even the brightest stars, will be missing or mis-detected in the query image (or the reference catalog), so we want to avoid over-using any particular star to build quads.

2.30 We handle the wide range of angular sizes by building a series of sub-indices, each of which contains quads whose quads have scales within a small range (for example, a factor of two). At some level this is simply an implementation detail: we could recombine the sub-indices into a single index, but in what follows it is helpful to be able to assume that the sub-index will be asked to recognize query images whose angular sizes are similar to the size of the quads it contains. Since we have a set of sub-indices, each of which is tuned to an overlapping range of scales, we know that at least one will be tuned to the scale of the query image.

2.31 We begin by selecting a spatially-uniform and bright subset of stars from our reference catalog. We do this by placing a grid of equal-area patches ("HEALPixels" [26]) over the sky and selecting a fixed number of stars, ordered by brightness, from each grid cell. The grid size is chosen so that grid cells are a small factor smaller than the query images. Typically we choose the grid cells to be about a third of the size of the query images, and select 10 stars from each grid cell, so that most query images will contain about a hundred query stars. Figure 2.2 illustrates this process on a small patch of sky.

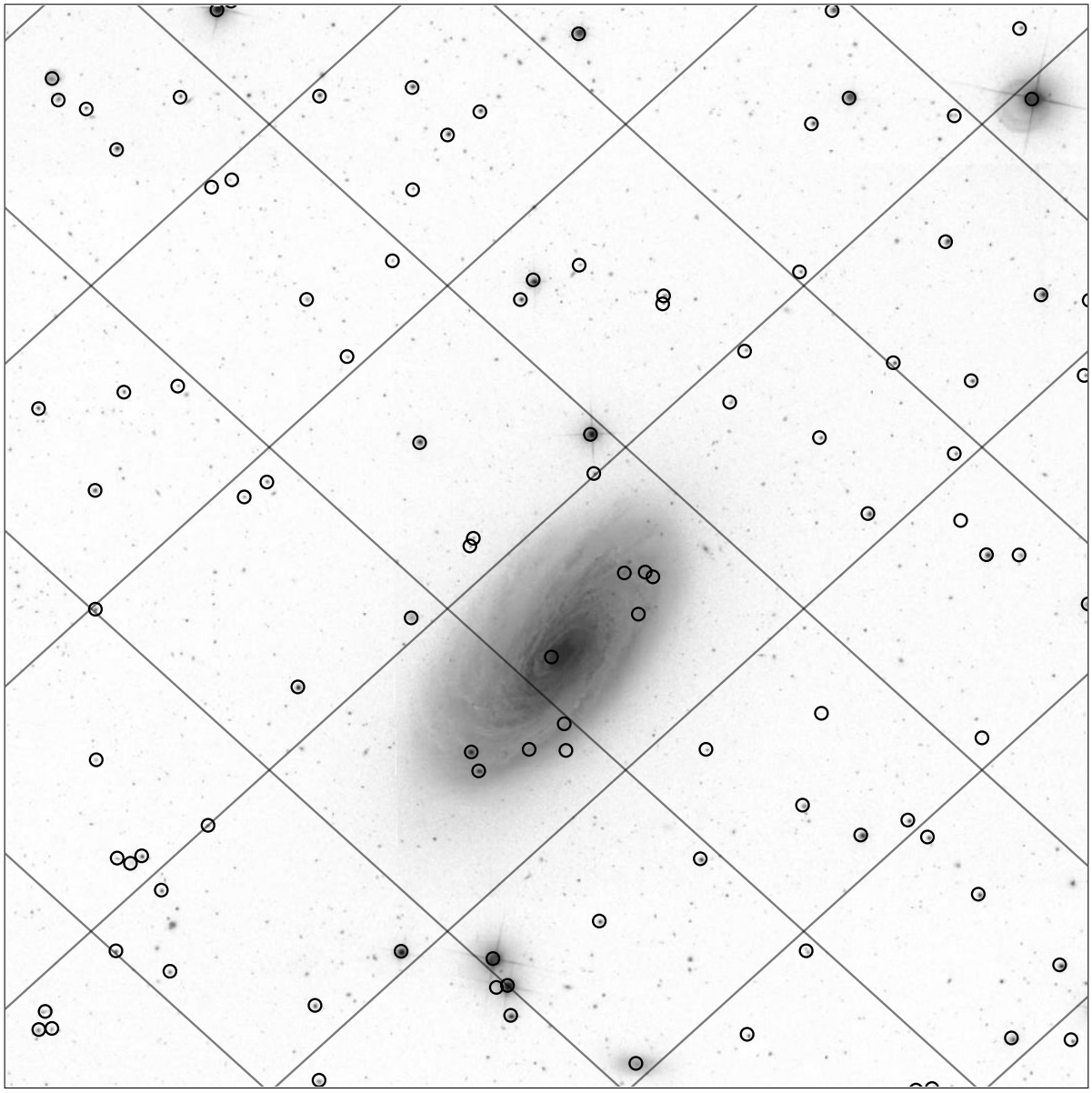


Figure 2.2: A small region of sky (about 0.3×0.3 degrees centered on $(\mathrm{RA}, \mathrm{Dec}) = (188, 14.45)$ degrees), showing the HEALPix grid, and the brightest 5 stars that we select from each cell. The image shown is from the Sloan Digital Sky Survey.

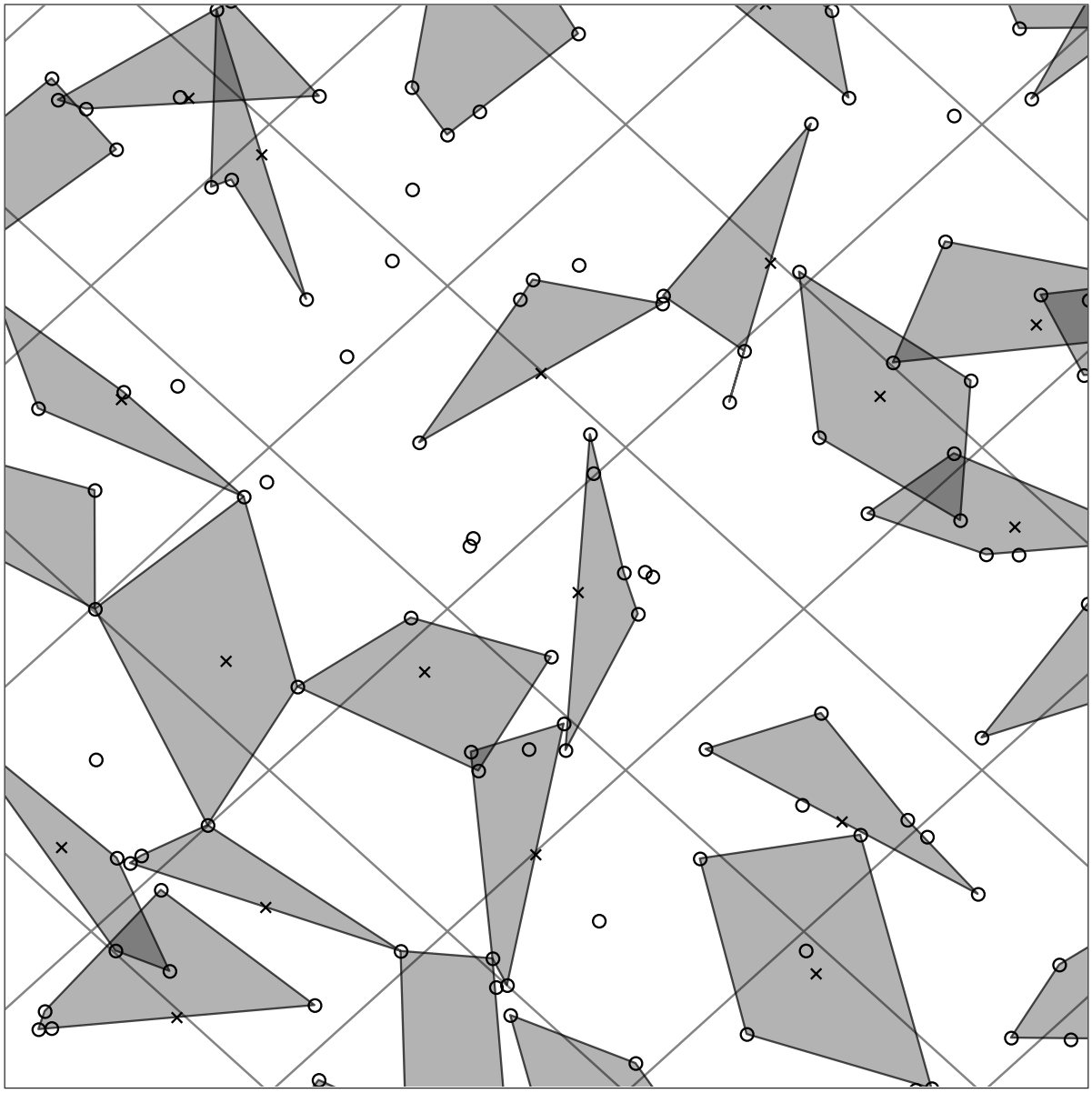


Figure 2.3: The same region of sky as shown in the previous figure, showing the HEALPix grid, and the quads that are created during the first pass through the grid cells. The quads must have a diameter (the distance between the two most distant stars) within a given range—in this case, 1 to $\sqrt{2}$ times the side-length of the grid cells. In each grid cell, the system attempts to build a quad whose center (*i.e.*, midpoint of the diameter line)—marked with an $\times$ in the figure—is within the grid cell.

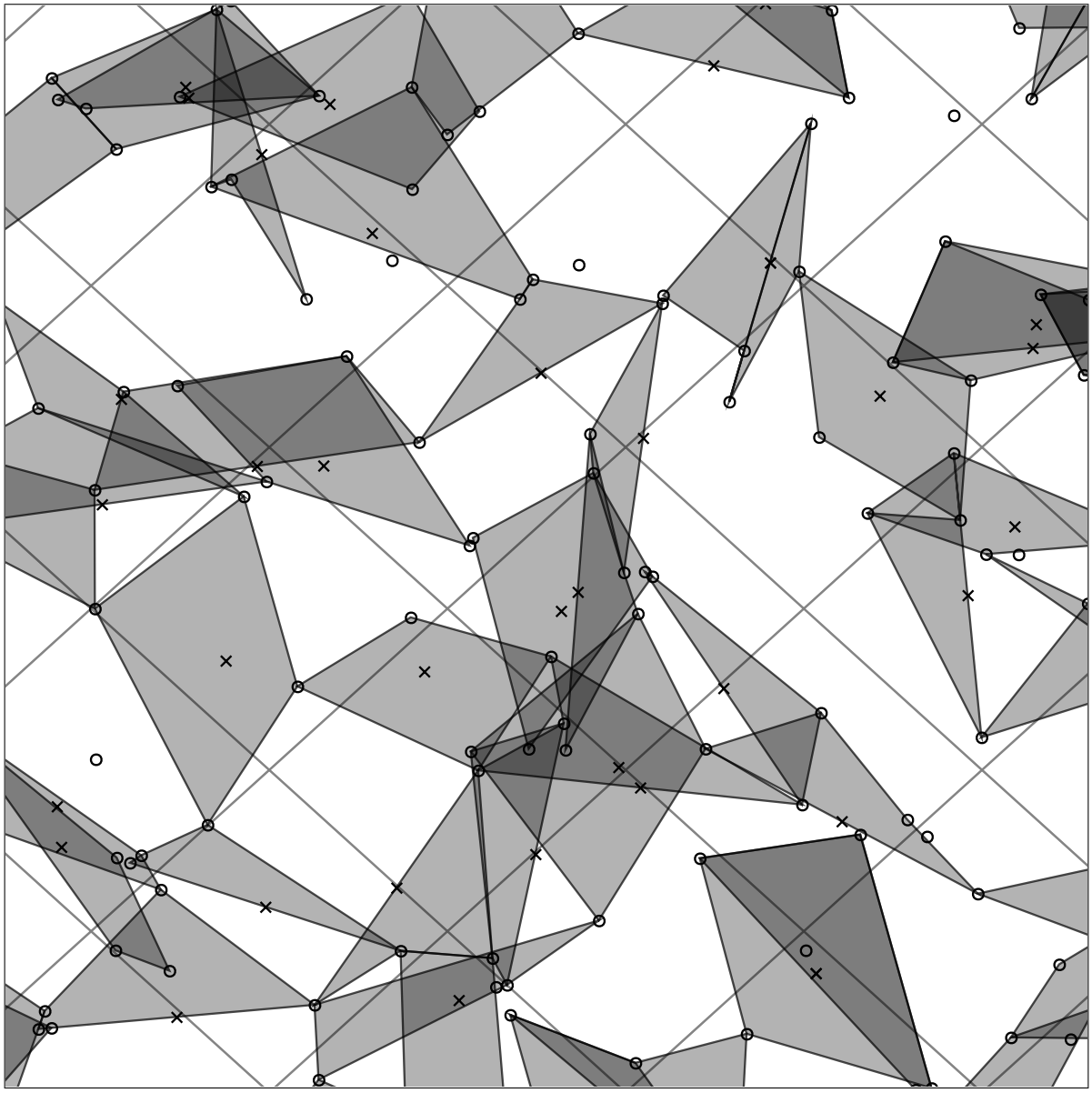


Figure 2.4: The same region of sky as in the previous figures, showing the quads that are created after the second round of attempting to build a quad in each grid cell.

2.32 Next, we visit each grid cell and attempt to find a quad within the acceptable range of angular sizes and whose center lies within the grid cell. We search for quads starting with the brightest stars, but for each star we track the number of times it has already been used to build a quad, and we skip stars that have been used too many times already. We repeat this process, sweeping through the grid cells and attempting to build a quad in each one, a number of times. In some grid cells we will be unable to find an acceptable quad, so after this process has finished we make further passes through the grid cells, removing the restriction on the number of times a star can be used, since it is better to have a quad comprised of over-used stars than no quad at all. Typically we make a total of 16 passes over the grid cells, and allow each star to be used in up to 8 quads. Figures 2.3 and 2.4 show the quads built during the first two rounds of quad-building in our running example.

2.33 In principle, an index is simply a list of quads, where for each quad we store its geometric hash code, and the identities of the four stars of which it is composed (from which we can look up their positions on the sky). However, we want to be able to search quickly for all hash codes near a given query hash code. We therefore organize the hash codes into a kd-tree data structure, which allows rapid retrieval of all quads whose hash codes are in the neighborhood of any given query hash code. In order to carry out the verification step we also keep the star positions in a kd-tree, since for each matched quad we need to find other stars that should appear in the image if the match is true. Since none of the available kd-tree implementations were satisfactory for our purposes, we created a fast, memory-efficient, pointer-free kd-tree implementation. Our kd-tree implementation is presented in detail in chapter 4.

2.34 Given a query image, we detect stars as discussed above, and sort the stars by brightness. Next, we begin looking at quads of stars in the image. For each quad, we compute its geometric hash code and search for nearby codes in the index. For each matching code, we retrieve the positions of the stars that compose the quad in the index, and

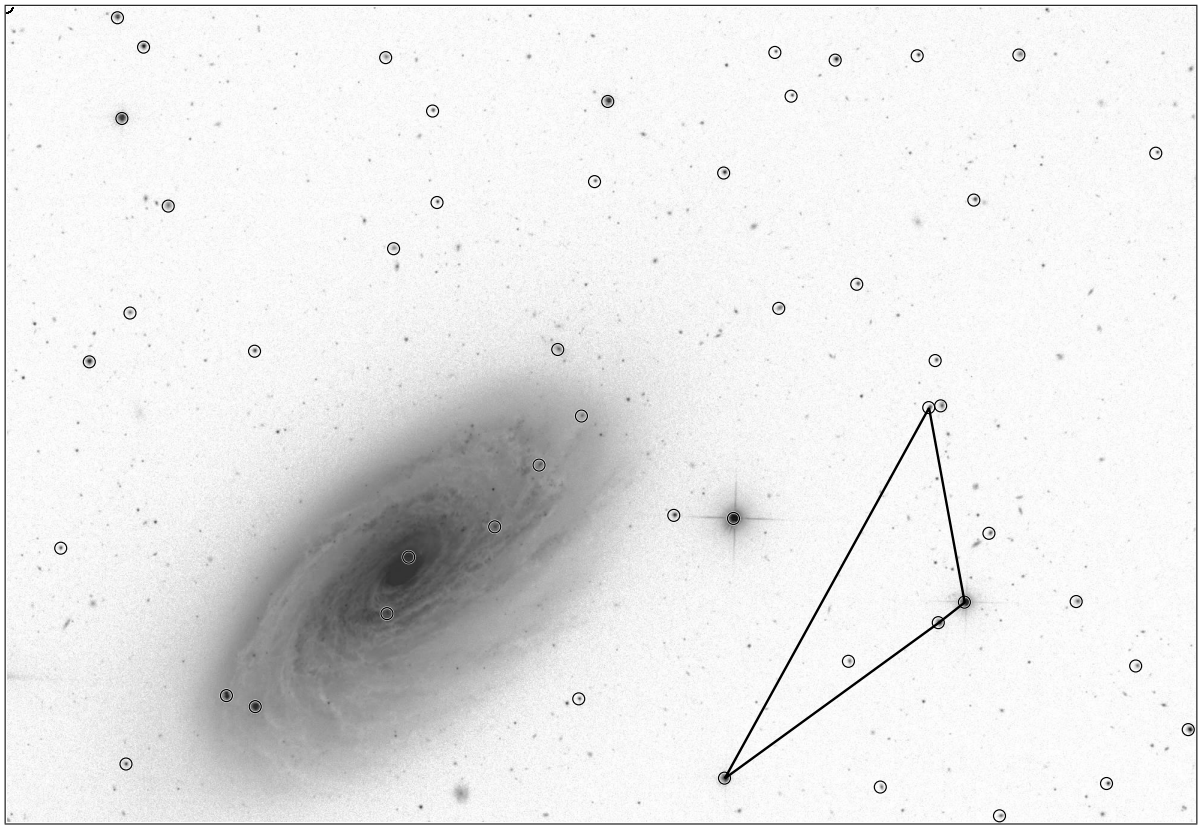


Figure 2.5: A sample query image, with the brightest 100 sources our system detects (circles), and a quad in the image to which our system will search for matches in the index. This quad looks like a triangle because two of its stars are nearly collinear. Image credit: Sloan Digital Sky Survey.

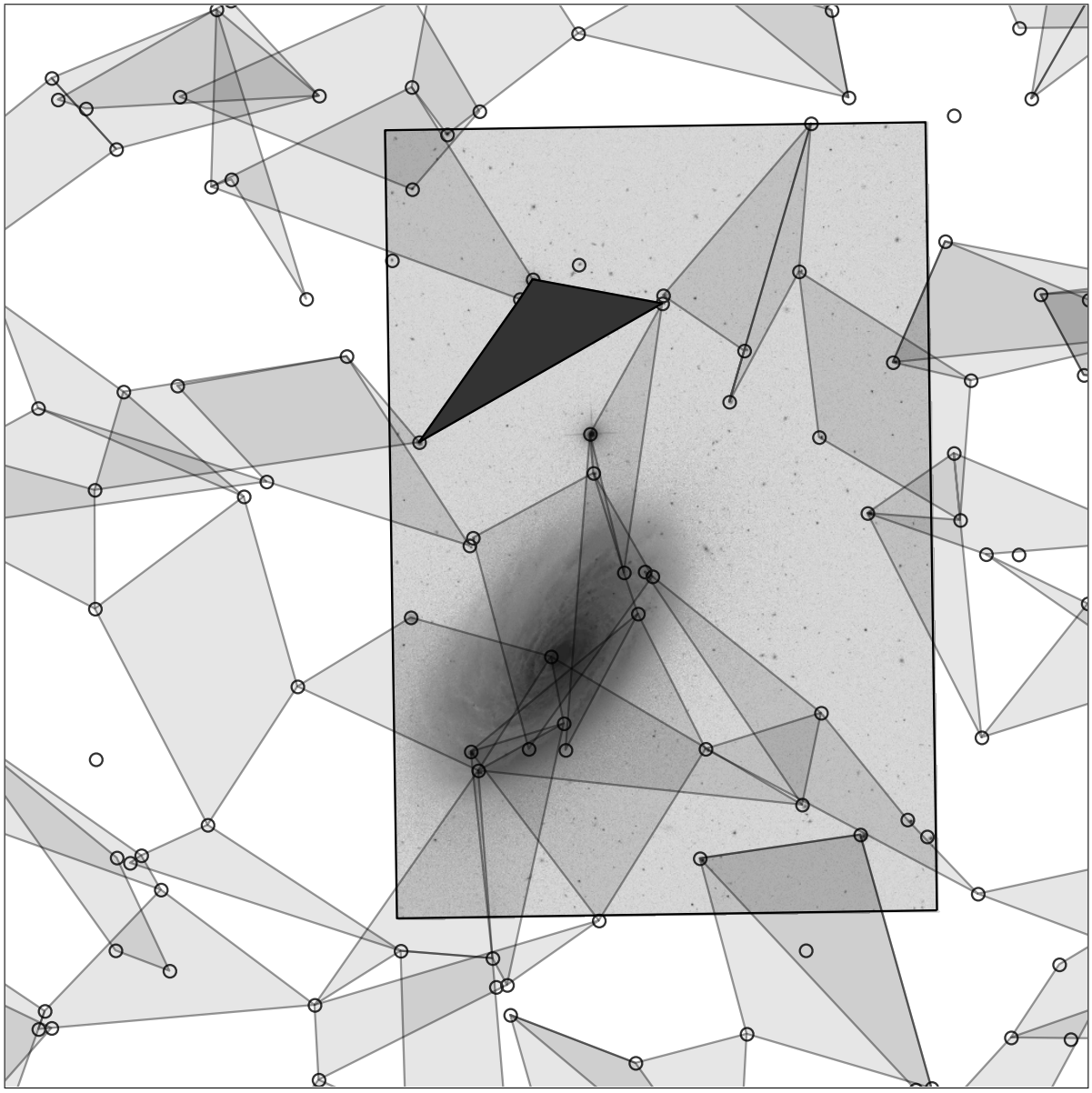


Figure 2.6: Our example index, showing a quad in the index that matches a quad from the query image (shaded solid). The image is shown projected in its correct orientation on the sky.

compute the hypothesized alignment—the World Coordinate System—of the match. We then retrieve other stars in the index that are within the bounds of the image, and run the verification procedure to determine whether the match is true or false. We stop searching when we find a true match in which we are sufficiently confident, or we exhaust all the possible quads in the query image, or we run out of time. See figure 2.5.

2.2.4 Verification of hypotheses

2.35 The indexing system generates a large number of hypothesized alignments of the image on the sky. The task of the verification procedure is to reject the large number of false matches that are generated, and accept true matches when they are found. Essentially, we ask, "if this proposed alignment were correct, where else in the image would we expect to find stars?" and if the alignment has very good predictive power, we accept it.

2.36 We have framed the verification procedure as a Bayesian decision process. The system can either accept a hypothesized match—in which case the hypothesized alignment is returned to the user—or the system can reject the hypothesis, in which case the indexing system continues searching for matches and generating more hypotheses. In effect, for each hypothesis we are choosing between two models: a "foreground" model, in which the alignment is true, and a "background" model, in which the alignment is false. In Bayesian decision-making, three factors contribute to this decision: the relative abilities of the models to explain the observations, the relative proportions of true and false alignments we expect to see, and the relative costs or *utilities* of the outcomes resulting from our decision.

2.37 The *Bayes factor* is a quantitative assessment of the relative abilities of the two models—the foreground model F and the background model B—to produce or explain the observations. In this case the observations, or data, D, are the stars observed in the

query image. The Bayes factor

$$K = \frac{p(D \mid F)}{p(D \mid B)} \tag{2.1}$$

is the ratio of the marginal likelihoods. We must also include in our decision-making the prior $p(F)/p(B)$, which is our *a priori* belief, expressed as a ratio of probabilities, that a proposed alignment is correct. Since we typically examine many more false alignments than true alignments (because we stop after the first true alignment is found), this ratio will be small.

2.38 The final component of Bayesian decision theory is the *utility* table, which expresses the subjective value of each outcome. It is good to accept correctly a true match or reject correctly a false match ("true positive" and "true negative" outcomes, respectively), and it is bad to reject a true match or accept a false match ("false negative" and "false positive" outcomes, respectively). In the *Astrometry.net* setting, we feel it is very bad to produce a false positive: we would much rather fail to produce a result rather than produce a false result, because we want the system to be be able to run on large data sets without human intervention, and we want to be confident in the results.

2.39 Applying Bayesian decision theory given our desired operating characteristics, we find that we should accept a proposed alignment only if the Bayes factor is exceedingly large. That is, the foreground model in which the alignment is true must be far better at explaining the observed positions of stars in the query image than the background model that the alignment is false. We typically set the Bayes factor threshold to 10^9 or 10^{12}, but as will be shown in the experiments below, we could set it even higher. This threshold is computed from our (subjective) desired operating characteristics, so is not derivable from first principles.

2.40 In the foreground model, F, the four stars in the query image and the four stars in the index are aligned. We therefore expect that other stars in the query image will be close to other stars in the index. However, we also know that some fraction of the stars in the query image will have no counterpart in the index, due to occlusions or artifacts in

the images, errors in star detection or localization, differences in the spectral bandpass, or because the query image "star" is actually a planet, satellite, comet, or some other non-star, non-galaxy object. True stars can be lost, and false stars can be added. Our foreground model is therefore a mixture of a uniform probability that a star will be found anywhere in the image—a query star that has no counterpart in the index—plus a blob of probability around each star in the index, where the size of the blob is determined by the combined positional variances of the index and query stars.

2.41 Under the background model, B, the proposed alignment is false, so the query image is from some unknown part of the sky; the index is not useful for predicting the positions of stars in the image. Our simple model therefore places uniform probability of finding stars anywhere in the test image.

2.42 The verification procedure evaluates stars in the query image, in order of brightness, under the foreground and background models. The product of the foreground-to-background ratios is the Bayes factor. We continue adding query stars until the Bayes factor exceeds our threshold for accepting the match, or we run out of query stars.

2.43 There are some subtleties in the verification process which are explored in depth in chapter 3.

2.3 Results

2.3.1 Astrometric calibration of the Sloan Digital Sky Survey

2.44 We explored the potential for automatically organizing and annotating a large real-world data set by taking a sample of images generated by the Sloan Digital Sky Survey and considering them as an unstructured set of independent queries to our system. For each SDSS image, we discarded all meta-data, including all positional and rotational information and the date on which the exposure was taken. We allowed ourselves to look only at the two-dimensional positions of detected "stars" (most of which were in fact stars

but some of which were galaxies or detection errors) in the image. Normally, our system would take *images* as input, running a series of image processing steps to detect stars and localize their positions. The SDSS data reduction pipeline already includes such a process, so for these experiments we used these detected star positions rather than processing all the raw images ourselves. Further experiments have shown that we would likely have achieved similar, if not better, results by using our own image processing software.

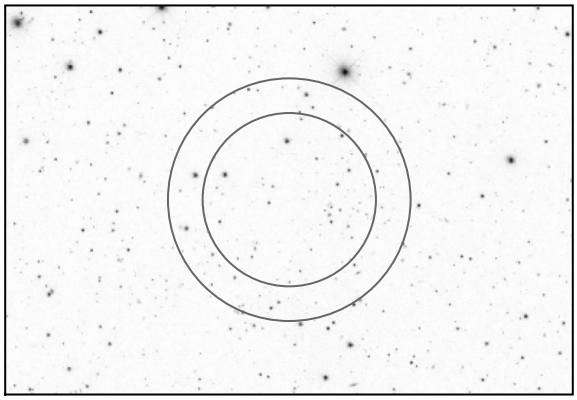


Figure 2.7: A typical image from the Sloan Digital Sky Survey (SDSS). The range of quad diameters that we use in the experiments below is shown by the circles. Image credit: Sloan Digital Sky Survey.

2.45 Each SDSS image has 2048×1489 pixels and covers 9×13 arcmin^2, slightly less than one-millionth the area of the sky. Each image measures one of five bandpasses, called u, g, r, i, and z, spanning the near-infrared through optical to near-ultraviolet range. Each band receives a 54-second exposure on a 2.5-meter telescope. A typical image is shown in figure 2.7.

2.46 The SDSS image-processing pipeline assigns to each image a quality rating: "excellent", "good", "acceptable", or "bad". We retrieved the source positions (*i.e.*, the list of objects detected by the SDSS image-processing pipeline) in every image within the main survey (Legacy and SEGUE footprints), using the public Catalog Archive Server (CAS)

interface to Data Release 7 [1]. We retrieved only sources categorized as "primary" or "secondary" detections of stars and galaxies, and required that each image contained at least 300 objects. The images that are excluded by these cuts contain either very bright stars or peculiarities that cause the SDSS image-processing pipeline to balk. The number of images in Data Release 7 and our cut is given in the table below.

Quality	**Total number of images**	**Number of images in our cut**
excellent	183,359	182,221
good	101,490	100,763
acceptable	48,802	48,337
bad	93,692	89,219
total	427,343	420,540

2.3.1.1 Performance on excellent images

2.47 In order to show the best performance achievable with our system, we built an index that is well-matched to SDSS r-band images. Starting with the red bands from a cleaned version of the USNO-B catalog, we built an index containing stars drawn from a HEALPix grid with cell sizes about 4×4 arcmin^2, and 10 stars per cell. We then built quads with diameters of 4 to 5.6 arcmin. For each grid cell, we searched for a quad whose center was within the grid cell, starting with the brightest stars but allowing each star to be used at most 8 times. We repeated this process 16 times. The index contains a total of about 100 million stars and 150 million quads. Figure 2.8 shows the spatial distribution of the stars and quads in the index.

2.48 We randomized the order of the excellent-quality r-band images, discarded all astrometric meta-data—leaving only the pixel positions of the brightest 300 stars—and asked our system to recognize each one. We allowed the system to create quads from only the brightest 50 stars. All 300 stars were used during the hypothesis-checking step, but since the Bayes factors tend to be overwhelmingly large, we would have found similar results

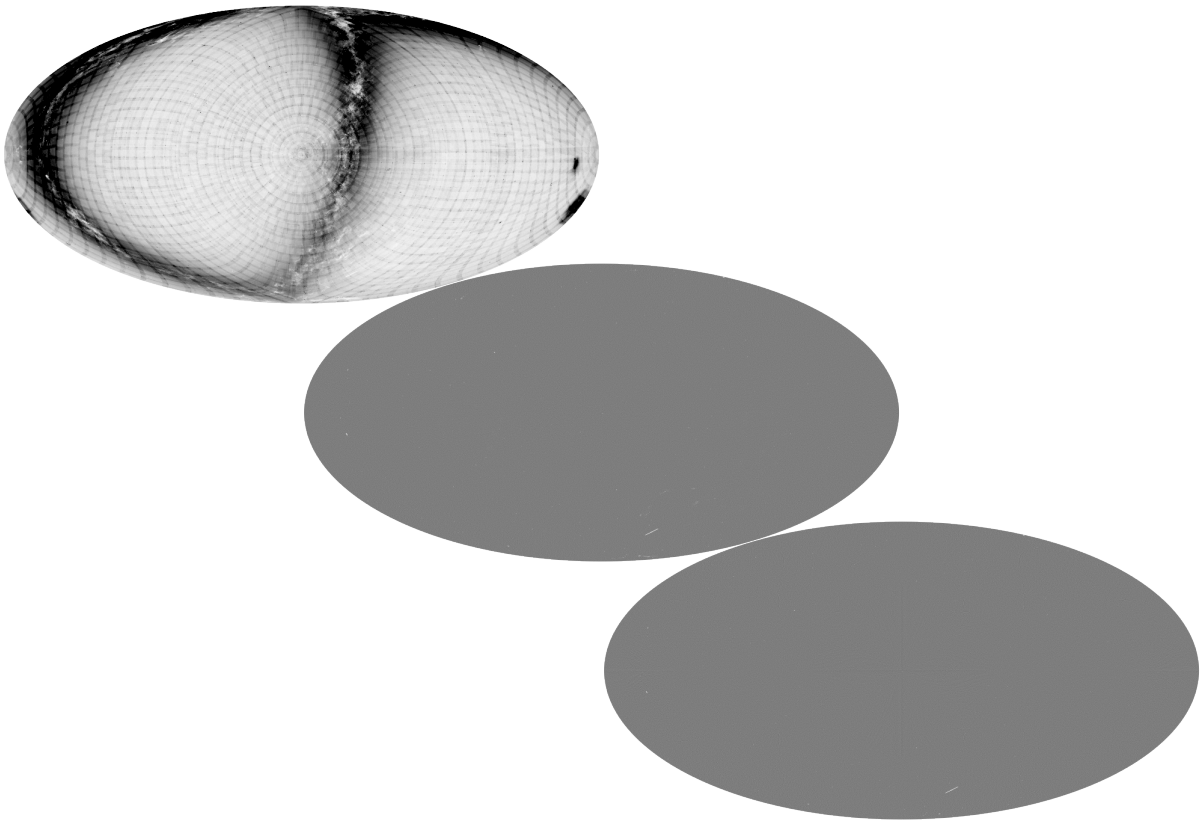


Figure 2.8: **Top:** Density of sources in the USNO-B catalog (in Hammer-Aitoff projection). Dark colors indicate high density. The north celestial pole is in the center of the image. The dark strip through the center and around the edges is the Milky Way; lower-density dust lanes can be seen. The USNO-B catalog was created by scanning photographic plates, and the places where the plates overlap are clearly visible as concentric rings and spokes of overdensities. **Middle:** Density of sources in our spatially uniform cut of the USNO-B catalog. Most of the sky is very uniformly covered. A few small bright (low-density) areas are visible, including a line near the bottom. These are areas where the USNO-B catalog is underdense due to defects. **Bottom:** Density of quads in the index used in most of the experiments presented here. Again, most of the sky is uniformly covered with quads.

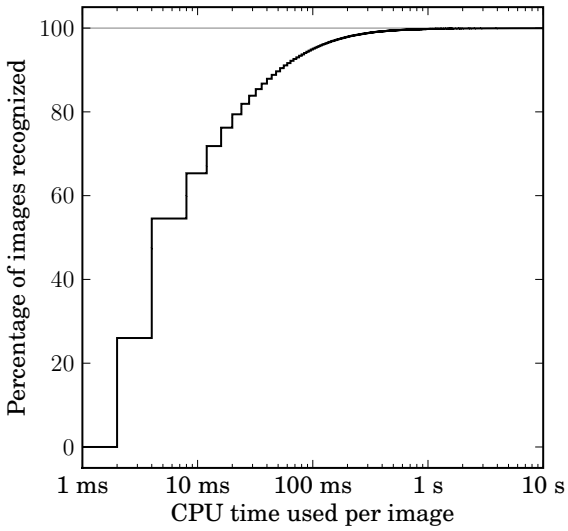


Figure 2.9: Results from the excellent r-band SDSS images and USNO-B-based index. The percentage of images that are recognized correctly (*i.e.*, astrometrically calibrated) with respect to the CPU time spent per image. Many images are recognized rapidly, but there is a heavy tail of "hard" images. Spending more and more CPU results in sharply diminishing returns. After 1 second, over 99.7 % of images are recognized, and after 10 seconds, over 99.97 % are recognized. The steps are due to the finite resolution of the CPU timer we are using.

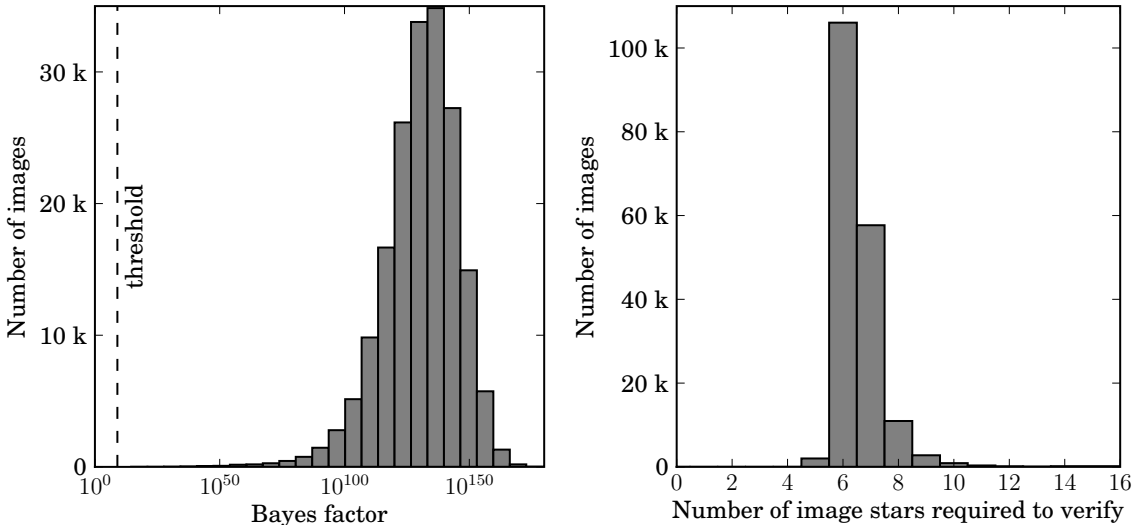


Figure 2.10: Results from the excellent r-band SDSS images, continued. **Left:** The Bayes factor of the foreground model versus the background model for the hypotheses that we accept. The dashed line shows the threshold implied by our desired operating characteristics. These excellent-quality images yield incredibly high Bayes factors—when we find a correct match is it unequivocal. **Right:** The number of stars in the query image that had to be examined before the Bayes-factor threshold was reached.

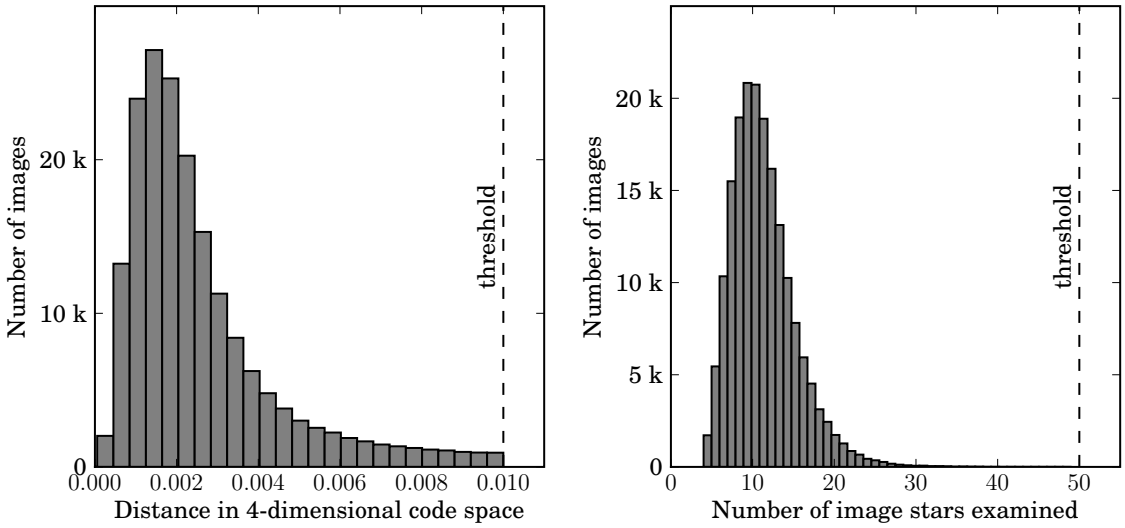


Figure 2.11: Results from the excellent r-band SDSS images, continued. **Left:** The distance in four-dimensional geometric hash code space between the query quad and the first correctly-matched index quad. In this experiment we searched for matches within distance 0.01: well into the tail of the distribution. **Right:** The number of stars in the query image that the system built quads from before finding the first correct match. In a few cases, the brightest 4 stars formed a valid quad which was matched correctly to the index: we got the correct answer after our first guess!

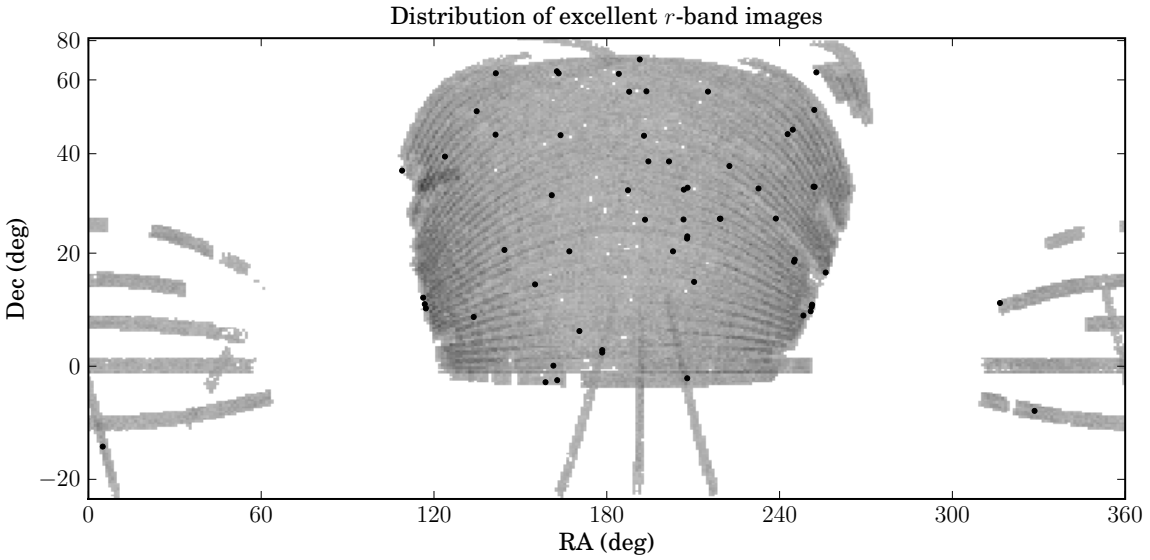


Figure 2.12: Distribution of the excellent-quality r-band SDSS images on the sky. The gray footprint represents the correctly-recognized images, while the black dots show the images that were not recognized using the USNO-B-based index. There is a slight over-density of failures near the beginnings and ends of runs, but otherwise no apparent spatial structure.

if we had kept only the 50 brightest stars. We also told our system the angular size of the images to within about 1 %, though we emphasize that this was merely a means of reducing the computational burden of this experiment: we would have achieved exactly the same results (after more compute time) had we provided no information about the scale whatsoever; we show this in section 2.3.1.5 below.

Phase	**Images recognized**	**Unrecognized**	**Percent recognized**
USNO-B: 0.1 s	172,882	9,339	94.87
USNO-B: 1 s	181,826	395	99.78
USNO-B: 10 s	182,158	63	99.97
USNO-B: 15 s	182,160	61	99.97
2MASS	182,211	10	99.99
Original images	182,221	0	100.00

2.49 The results, shown in the table above and in figures 2.9, 2.10, 2.11, and 2.12, are that we can successfully recognize over 99.97 % of the images. We then examined our reference catalog, USNO-B, at the true locations of the images that were unrecognized. Some of these locations contained unusual artifacts. For example, figure 2.13 shows a region where the USNO-B catalog contains "worm" features. The cause of these artifacts is unknown (David Monet, private communication), but they affect several square degrees of one of the photographic plates that were scanned to create the USNO-B catalog.

2.50 In order to determine the extent to which our failure to recognize images is due to problems with the USNO-B reference catalog, we built an index from the Two-Micron All-Sky Survey (2MASS) catalog, using the same process as for the USNO-B index, using the 2MASS J-band rather than the USNO-B red bands. We then asked our system to recognize each of the SDSS images that were unrecognized using the USNO-B-based index. Of the 61 images, 51 were recognized correctly, leaving only 10 images unrecognized. Examining these images, we found that some contained bright, saturated stars which had been flagged as unreliable by the SDSS image-reduction pipeline. We retrieved

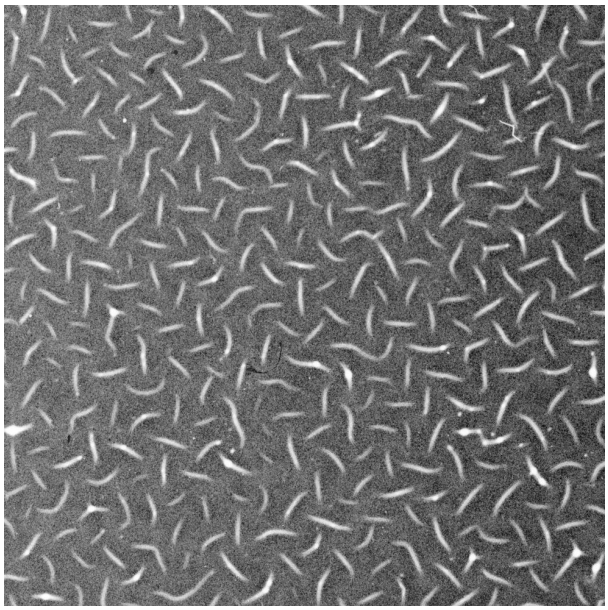


Figure 2.13: "Worms" in USNO-B. We found these unusual artifacts by looking at one of the places where our system failed to recognize SDSS images. The image is of the photographic plate POSS-IE 275, centered on $(\mathrm{RA}, \mathrm{Dec}) = (243, 36)$ degrees and 15×15 arcmin in size. Image credit: Copyright Palomar Observatory, National Geographic Society, and California Institute of Technology; courtesy of USNO Image and Catalogue Archive.

the original data frames and asked our system to recognize them. All 10 were recognized correctly: our source extraction procedure was able to localize the bright stars correctly, and with these the indexing system found a correct hypothesis. With these three processing steps, we achieve an overall performance of 100 % correct recognition of all 182, 221 excellent images, with no false positives. This took a total of about 80 minutes of CPU time. The index is about 5 gigabytes in size, and once it is loaded into memory, multiple CPU cores can use it in parallel, so the wall-clock time can be a fraction of the total CPU time. During this experiment, a total of over 180 million quads were tried, resulting in about 77 million matches to quads in the index. Many of these matches were found to result in image scales that were outside the range we provided to the system, so the verification procedure was run only 6 million times.

2.51 For completeness, we also checked the images that were rated as excellent but failed our selection cut. We retrieved the original images and used our source extraction routine and both the USNO-B- and 2MASS-based indexes. Our system was able to recognize correctly all 1138 such images.

2.3.1.2 Performance on images of varying bandpass

2.52 In order to investigate the performance of our system when the bandpass of the query image is different than that of the index, we asked the system to recognize images taken through the SDSS filters u, g, r, i, and z. We used only the images rated "excellent".

CPU time	**Percentage of images recognized**				
(per image)	u	g	r	i	z
0.1 s	87.80	93.88	94.87	93.59	94.36
1 s	98.58	99.73	99.78	99.73	99.75
10 s	99.82	99.96	99.97	99.96	99.96
60 s	99.84	99.96	99.97	99.96	99.96

2.53 The results, shown in the table above and in figure 2.14, demonstrate that as the

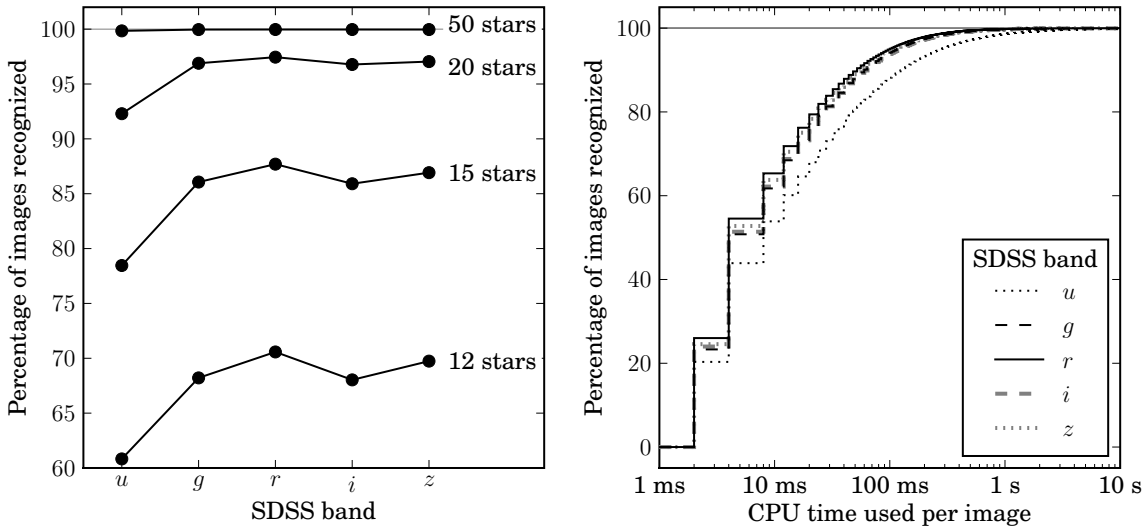


Figure 2.14: Performance on images taken through different SDSS bandpass filters. **Left:** The percentage of images recognized after building quads from a given number of the brightest stars in each image. The r-band is the best match to the USNO-B-based index we are using. Generally the recognition rate drops with the distance between the bandpass of the index and the bandpass of the image. The i-band performance in this instance is lower than expected. **Right:** The percentage of images recognized as the amount of CPU time spent per image increases. The r-band images are most quickly recognized, g-, i-, and z-band images take slightly more effort, and u-band images take considerably more CPU time. The asymptotic recognition rates are nearly identical except for u-band, which is slightly lower.

difference between the query image bandpass and the index bandpass increases, the amount of CPU time required to recognize the same fraction of images increases. This performance drop is more pronounced on the blue (u) side than the red (z) side. After looking at the brightest 50 stars, the system is able to recognize essentially the same fraction of images. As shown by the first experiment in this section, this asymptotic recognition rate is largely due to defects in the reference catalog from which the index is built.

2.3.1.3 Performance on images of varying quality

2.54 In order to characterize the performance of the system as image quality degrades, we asked the system to recognize r-band images that were classified as "excellent", "good", "acceptable", or "bad" by the SDSS image-reduction pipeline.

CPU time	**Percentage of images recognized**			
(per image)	Excellent	Good	Acceptable	Bad
0.1 s	94.87	94.85	94.57	84.11
1 s	99.78	99.74	99.64	96.58
10 s	99.97	99.94	99.94	99.11
60 s	99.97	99.94	99.95	99.18

2.55 The results, shown in the table above and in figure 2.15, show almost no difference in performance between excellent, good, and acceptable images. Bad images show a significant drop in performance, though we are still able to recognize over 99 % of them.

2.3.1.4 Performance on images of varying angular size

2.56 We investigated the performance of our system with respect to the angular size of the images by cropping out all but the central region of the excellent-quality r-band images and running our system on the sub-images. Recall that the original image size is 13×9 arcmin^2, and that the index we are using contains quads with diameters between

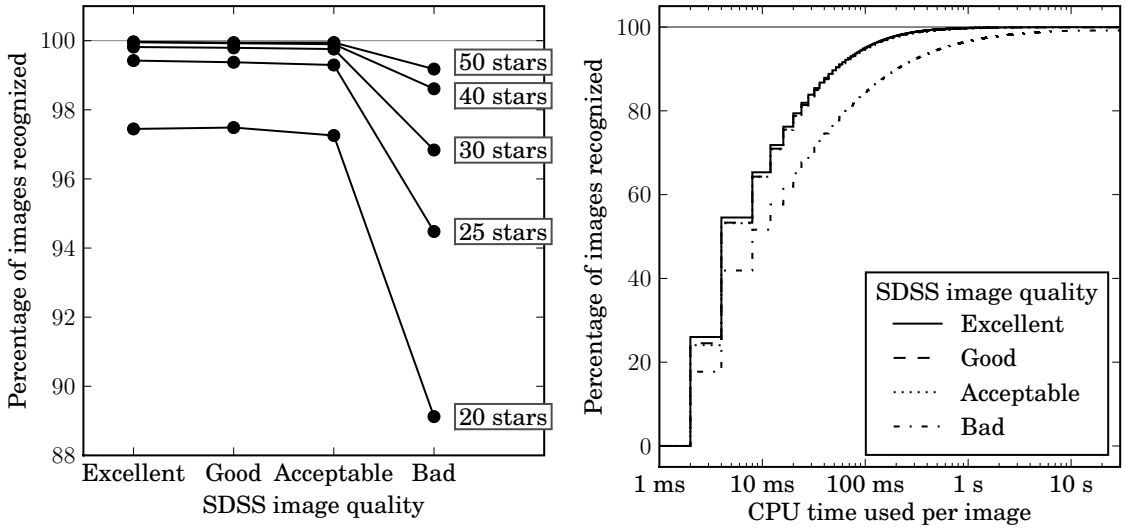


Figure 2.15: Performance of the system given images of varying quality. **Left:** The percentage of images recognized after looking at a given number of stars in each image, for excellent-, good-, acceptable-, and bad-quality images from SDSS. There is a small drop in performance for good and acceptable images, and a more significant drop for bad ones; all but the bad reach approximately the same asymptotic recognition rate. **Right:** CPU time per image for each quality rating. All but the bad images show nearly indistinguishable performance.

4 and 5.6 arcmin^2. We cut the SDSS images down to sizes 9×9, 8×8, 7×7, and 6×6 arcmin^2. The 7×7 and 6×6 images required much more CPU time, so we ran only small random subsamples of the images of these sizes.

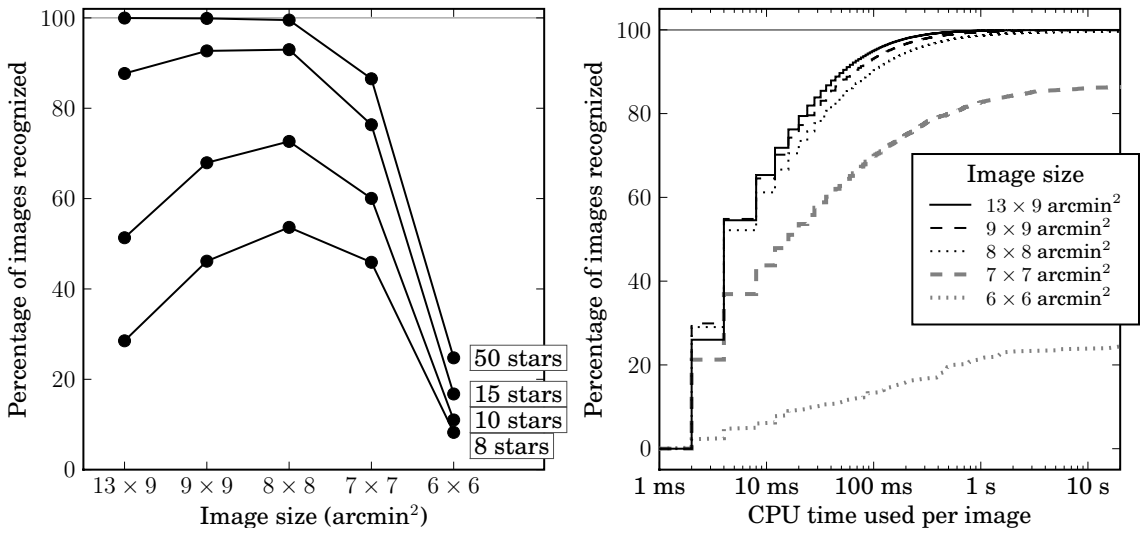


Figure 2.16: Performance of the system given images of varying angular sizes. **Left:** The percentage of images recognized after looking at a given number of stars in each image, for images of the given sizes. Perhaps surprisingly, more of the 8×8 images are recognized correctly during the first few milliseconds, but as more time elapses, the larger-scale images are more likely to be recognized. **Right:** CPU time per image required to recognize images of each angular size. After 10 ms, the larger images are recognized more quickly. The 7×7 and 6×6 images appear to be reaching asymptotic recognition rates far below 100 %.

Image size (arcmin^2)	**Percentage of images recognized**
13×9	99.97
9×9	99.88
8×8	99.52
7×7	86.53
6×6	24.75

2.57 The results, presented in the table above and in figure 2.16, show that performance degrades slowly for images down to 8×8 arcmin^2, and then degrades sharply. This is not surprising given the size of quads in the index used in this experiment: in the smaller images, only stars near the edges of the image can possibly be 4 to 5.6 arcmin away from another star, so the set of stars that can form the 'backbone' of a quad is small. Observe that images below 2.8×2.8 arcmin^2 cannot possibly be recognized by this index, since no pair of stars can be 4 arcmin or more away from each other.

2.58 This does not imply that small images cannot be recognized by our system: given an index containing smaller quads, we may still be able to recognize them. The point is simply that for any given index there is some threshold of angular size below which the image recognition rate will drop, and another threshold below which the recognition rate will be exactly zero.

2.3.1.5 Performance with varying image scale hints

2.59 In all the experiments above, we told our system the angular scale (in arcseconds per pixel) to within $\pm 1.25\,\%$ of the true value, and we used an index containing only quads within a small range of diameters. In this experiment, we show that these hints merely make the recognition process faster without affecting the general results.

2.60 We created a set of sub-indices, each covering a range of $\sqrt{2}$ in quad diameters. The smallest-scale sub-index contains quads of 2 to 2.8 arcmin, and the largest contains quads of about 20 to 30 degrees in diameter. Each sub-index is built using the same

methodology as outlined above for the 4 to 5.6 arcmin index, with the scale adjusted appropriately. The smallest-scale sub-index contains only 6 stars per HEALPix grid cell rather than 10 as in the other sub-indices, because the USNO-B catalog does not contain enough stars: a large fraction of the smallest cells contain fewer than 10 stars. In the smallest-scale sub-index we then do 9 rounds of quad-building, reusing each star at most 5 times, as opposed to 16 rounds reusing each star at most 8 times as in the rest of the sub-indices.

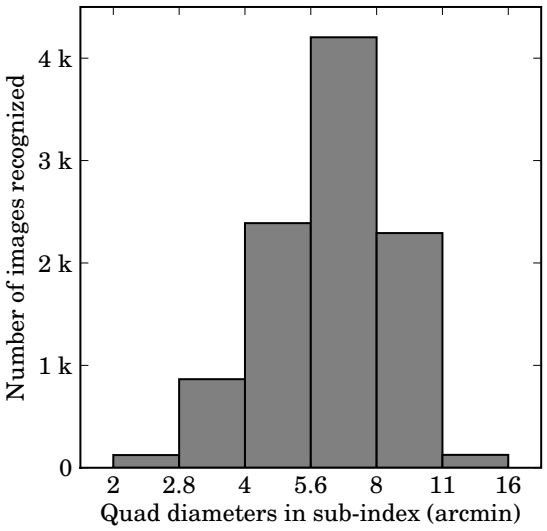


Figure 2.17: The sub-index that is first to recognize SDSS images, when all the sub-indices are run in lock-step. Although we used sub-indices containing quads up to 30 degrees in diameter, clearly only the ones that contain quads that can possibly be found in an SDSS image (which are about 16 arcmin across the diagonal) can generate correct hypotheses that will recognize the image. Perhaps surprisingly, each of these sub-indices is first to recognize some subset of the images, though a strong tuning effect is clear.

2.61 Each time our system examines a quad in the image, it searches each sub-index in turn for matching quads, and evaluates each hypothesized alignment generated by this process. The system proceeds in lock-step, testing each quad in the image against each

sub-index in turn. The first quad match that generates an acceptably good alignment is taken as the result and the process stops. Note that several of the sub-indices may be able to recognize any given image. Indeed, figure 2.17 shows that every sub-index that contains quads that can possibly be found in SDSS images is first to recognize some of the images in this experiment. Different strategies for ordering the computation—for example, spending equal amounts of CPU time in each sub-index rather than proceeding in lock-step—might result in better overall performance.

2.3.1.6 Performance with varying index quad density

2.62 The fiducial index we have been using in these experiments contains about 16 quads per HEALPix grid cell. Since each SDSS image has an area of about 7 cells, we expect each image to contain about 100 quad centers. The number of complete quads (*i.e.*, quads for which all four stars are contained in the image) in the image will be smaller, but we still expect each image to contain many quads. This gives us many chances of finding a correct match to a quad in the image. This redundancy comes at a cost: the total number of quads in the index determines the rate of false matches—since the code-space volume is fixed, packing more quads into the space results in a larger number of matches to any given query point—which directly affects the speed of each query. By building indices with fewer quads, we can reduce the redundancy but increase the speed of each query. This does not necessarily increase the overall speed, however: an index containing fewer quads may require more quads from the image to be checked before a correct match is found. In this experiment, we vary the quad density and measure the overall performance.

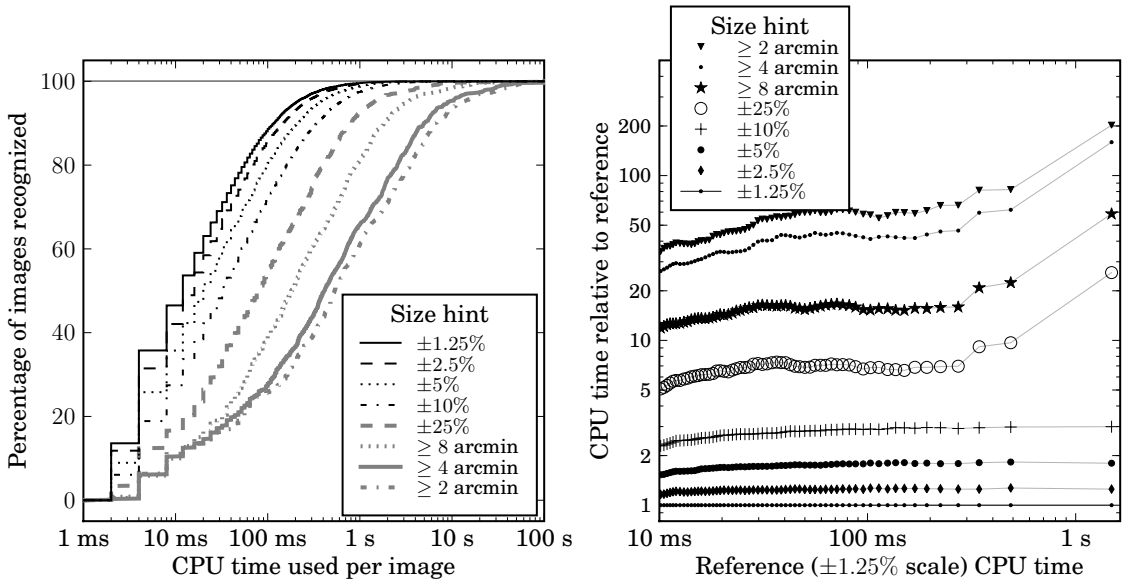


Figure 2.18: Performance of the system given varying limits on the image size. **Left:** CPU time per image required to recognize images, given various limits on the angular size of the image. Our system achieves the same asymptotic recognition rate in each case: giving the system less information about the true scale of the image simply means that it must evaluate and reject more false hypotheses before finding a true one. The "$\geq$ 8 arcmin" hint indicates that we told the system that the image width is between 8 arcmin and 180 degrees. The upper limit has much less impact on performance than the lower limit, since the sub-indices that cover large angular scales contain many fewer quads and are therefore much faster to search, and generate fewer coincidental matches. **Right:** CPU time per image relative to the $\pm 1.25\,\%$ case. We divided the CPU times for each case into percentiles; the mean time within each percentile is plotted. Generally, giving the system less information about the size of the images results in an approximately constant-factor increase in the CPU time required. Although there appears to be a sharp upward trend for the "loosest" four size ranges, this may be an effect of small sample size: since these cases take so long to run, we tested only 1000 images, while for the rest of the cases we tested $10,000$ images. The CPU time distribution is heavy-tailed, so the expected variance is large.

CPU time	**Percentage of images recognized**				
(per image)	16 quads/cell	9 quads/cell	4 quads/cell	3 quads/cell	2 quads/cell
0.1 s	94.44	96.32	95.89	94.30	90.94
1 s	99.76	99.84	99.61	99.36	97.35
10 s	99.96	99.95	99.79	99.65	97.92

2.63 As shown in the table above and figure 2.19, reducing the density from 16 to 9 quads per HEALPix grid cell has almost no effect on the recognition rate but takes only two-thirds as much CPU time. Reducing the density further begins to have a significant effect on the recognition rate, and actually takes *more* CPU time overall.

2.3.1.7 Performance on indices built from triangles and quints

2.64 We tested the performance of our quad-based index against a triangle-based index and a quintuple-based ("quint") index. The index-building processes were exactly as in our quad-based indices.

2.65 In the experiments above we searched for all matches within a distance of 0.01 in the quad feature space. Since the triangle- and quint-based indices have feature spaces of different dimensionalities (2 for triangles, 6 for quints), we first ran our system with the matching tolerance set to 0.02 in order to measure the distribution of distances of correct matches in the three feature spaces. We then set the matching tolerance to include 95 % of each distribution. For the triangle-based index, we found this matching tolerance to be 0.0064, for the quad-based index it was 0.0095, and for the quint-based index, 0.011. These experiments were performed on a random subset of 4000 images, because they are quite time-consuming.

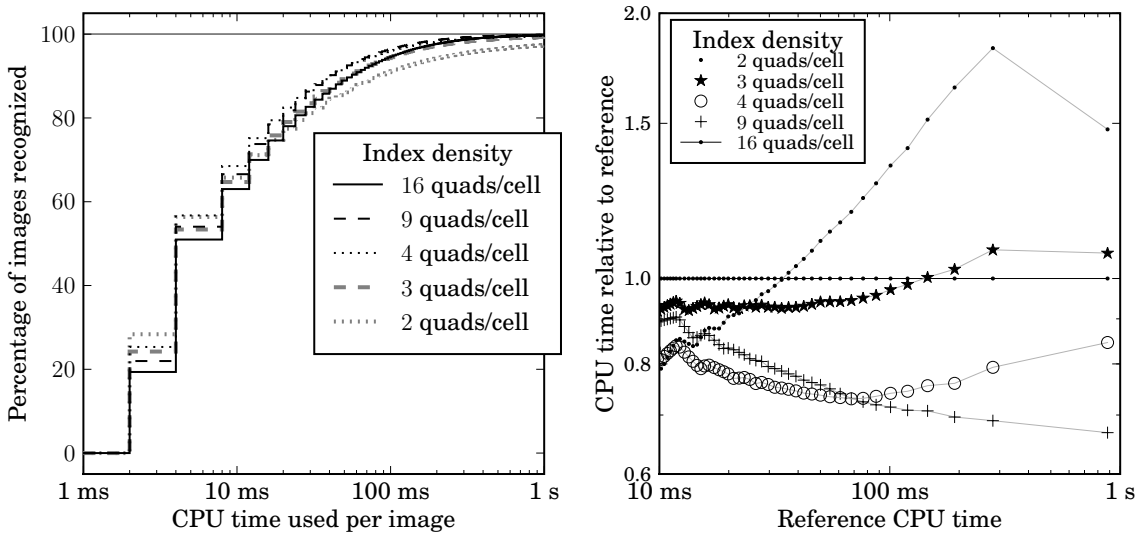


Figure 2.19: Performance of the system using indices containing varying densities of quads. (The total number of quads in an index is proportional to the density of quads.) **Left:** CPU time per image required to recognize images. **Right:** Relative CPU time to recognize images. We split the set of images into percentiles and have plotted the mean time within each percentile, relative to the 16-quad-per-cell reference index. The indices containing fewer quads are faster to search per quad query, but may require more quads to be tried before a correct match is found. The smaller indices are also able to recognize fewer images, because some images will simply not contain a quad that appears in the index. For the high-quality SDSS images we are using, the smallest of the indices here results in a 2 % drop in recognition rate (from nearly 100 % to about 98 %), but for poorer-quality images the drop could be larger.

CPU time	**Percentage of images recognized**		
(per image)	Triangles	Quads	Quints
0.1 s	0.20	57.45	23.35
1 s	28.07	92.20	36.67
10 s	78.58	99.28	72.75
100 s	99.15	99.33	95.08
1000 s	99.97	99.33	96.25

2.66 The results are given in the table above and in figure 2.20. After looking at the brightest 50 stars in each image, the triangle-based index is able to recognize the largest number of images, but both the triangle- and quint-based indices take significantly more time than the quad-based index. It seems that quad features strike the right balance between being distinctive enough that any given query does not generate too many coincidental (false) matches—as the triangle-based index does—but containing few enough stars that it does not take long to find a feature that is in both the image and the index—as the quint-based index does.

2.67 We expect that the relative performance of triangle-, quad-, and quint-based indices depends strongly on the angular size of the images to be recognized. An index designed to recognize images of large angular size requires fewer features, so the code space is less densely filled and fewer false matches are generated. For this reason, we expect that above some angular size, a triangle-based index will recognize images more quickly than a quad-based index.

2.3.2 Astrometric calibration of Galaxy Evolution Explorer data

2.68 To show the performance of our system on significantly larger images, at a bandpass quite far from that of our index, we experimented with data from the Galaxy Evolution

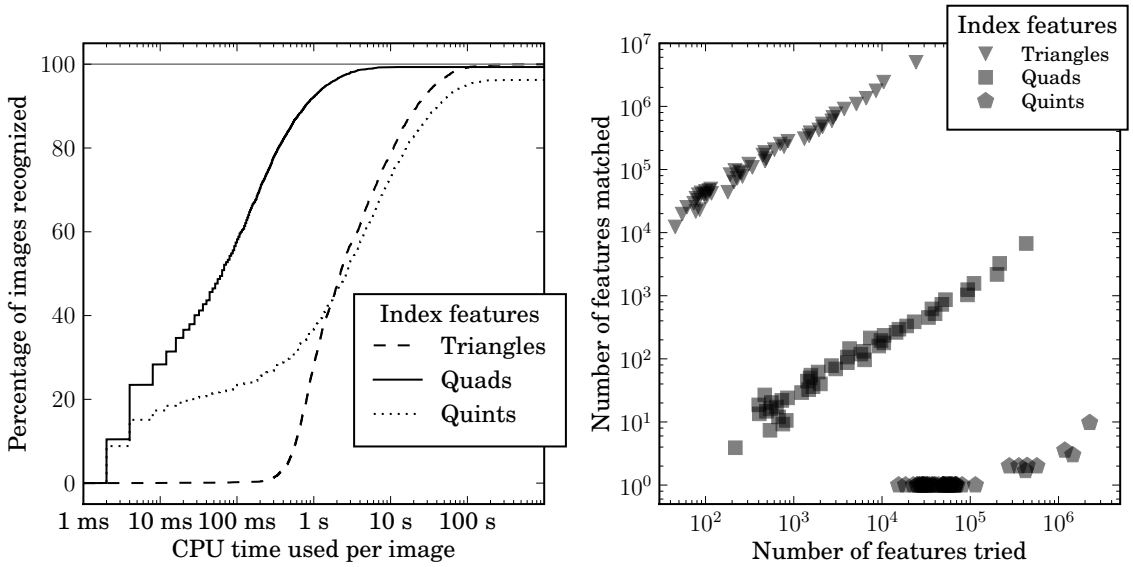


Figure 2.20: Performance of the system using indices containing triangles, quads, and quintuples of stars. **Left:** CPU time per image required to recognize images. The quad-based index vastly outperforms the triangle- and quint-based indices. **Right:** The number of features tried (*i.e.*, the number of triangles, quads, or quints of stars from the image that were tested), and the number of matches to features in the index that resulted. Each plotted point summarizes 2 % of the correctly-recognized images, sorted by the number of features tried. As expected, the triangle-based index produces many more matches for any given query, because the same number of features are packed into a lower-dimensional feature space. Fewer features have to be tried before the first correct matches are found, because only three corresponding stars have to be found. Quints, on the other hand, are very distinctive: for over 80 % of the images that were correctly recognized, the first matching quint ever found was a correct match. However, many quints from the image have to be tested before this match is found.

Explorer (GALEX). GALEX is a space telescope that observes the sky through near- and far-ultraviolet bandpass filters. Although it is fundamentally a photon-counting device, the GALEX processing pipeline renders images (by collapsing the time dimension and histogramming the photon positions), and these are the data products used by most researchers. The images are circular, with a diameter of about 1.2 degrees. See figure 2.21. In this experiment, rather than using the images themselves, we use the catalogs (lists of sources found in each image) that are released along with the images. These catalogs are produced by running a standard source extraction program (SExtractor) on the images. We retrieved the near-UV catalogs for all 28,182 images in Galex Release 4/5 that have near-UV exposure.

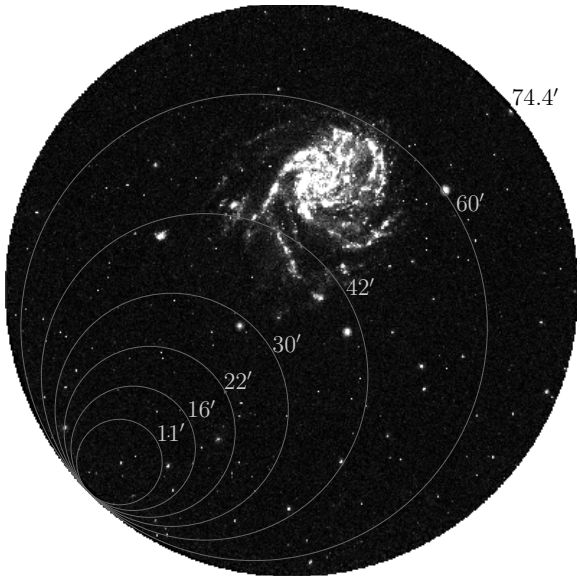


Figure 2.21: A sample GALEX field (AIS-101-sg32-nuv), with circles showing the sizes of the quads in the indices we use in this experiment. The images produced by the GALEX pipeline are 3840 × 3840 pixels, and the near-UV field of view is a circle of diameter 1.24 degrees, or about 74.4 arcmin. Image credit: Courtesy NASA/JPL-Caltech.

2.69 We built a series of indices of various scales, each spanning roughly $\sqrt{2}$ in scale. The smallest contained quads with diameters from 11 to 16 arcmin, the next contained quads

between 16 and 22 arcmin, followed by 22 to 30, 30 to 42, and 42 to 60 arcmin. Each index was built according to the "standard recipe" given above, using stars from the red bands of USNO-B as before. In total these indices contain about 30 million stars and 36 million quads.

CPU time (per image)	**Percentage of images recognized**
1 s	74.46
10 s	93.56
100 s	98.95
1000 s	99.74

2.70 In this experiment, we told our system that the images were between 1 and 2 degrees wide, and we allowed it to build quads from the first 100 sources in each image. The results are shown in the table above and figure 2.22. The recognition rate is quite similar to that of the excellent-quality SDSS r-band fields, suggesting that even though the near-UV bandpass of these images is quite far from the bandpass of the index, which we would expect to make the system less successful at recognizing these images, their larger angular size seems to compensate. The system is significantly slower at recognizing GALEX images, but this is partly because we gave it a fairly wide range of angular scales, and because we used several indices rather than the single one whose scale is best tuned to these images.

2.3.3 Astrometric calibration of Hubble Space Telescope data

2.71 In order to demonstrate that there is nothing intrinsic in our method that limits us to images of a particular scale, we retrieved a set of Hubble Space Telescope (HST) images and built a custom index to recognize them. We chose the All-wavelength Extended Groth strip International Survey (AEGIS) [17] footprint because it has many HST exposures and is within the SDSS footprint.

2.72 To build the custom index for this experiment, we retrieved stars and galaxies from

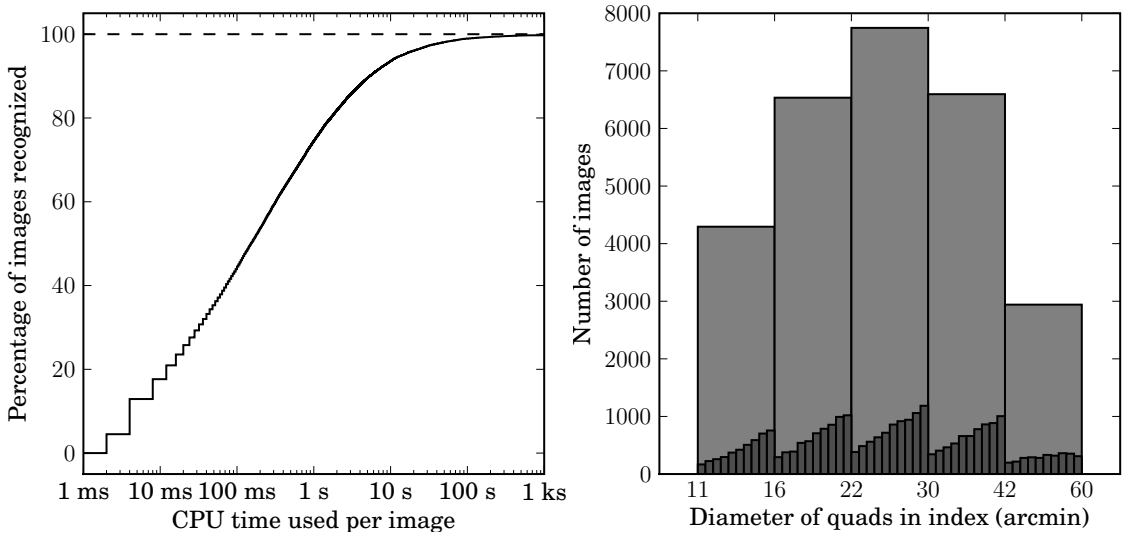


Figure 2.22: Results on GALEX near-UV images. **Left:** CPU time used per image. The shape of the curve is very similar to that in the previous SDSS experiments, though the "knee" of diminishing returns occurs after more CPU time (possibly because we gave the system much less information about the correct scale of the images). **Right:** The number of images recognized by each of the indices (identified by the range of sizes of quads they contain). For each quad in the image, we search for matches in each of the indices, stopping after the first match that is confirmed by the verification test. The histogram therefore shows only which index recognized the image *first*, rather than which indices might have recognized it given more time. The inset histograms show the distribution of quad sizes within each index (on a linear scale). Generally the larger quads are more successful, except in the largest index, where the size of the largest quads approaches that of the whole image.

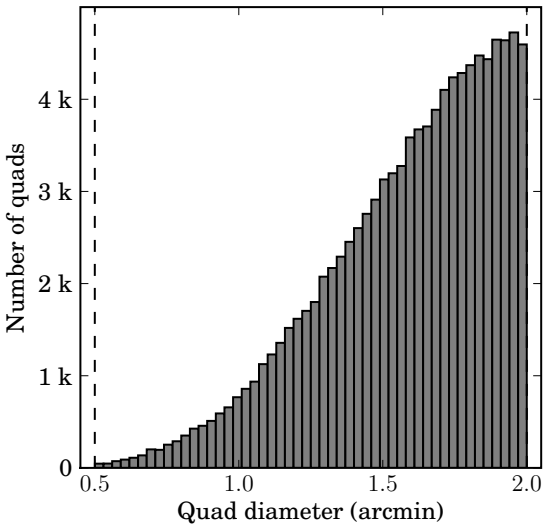


Figure 2.23: The diameters of quads in our custom index, built from SDSS stars, for recognizing Hubble Space Telescope Advanced Camera for Surveys (ACS) images in the AEGIS footprint. Although we allowed quads with diameters between 0.5 and 2 arcmin, the size distribution of the quads that were created is heavily biased toward the large end of the range, because the density of SDSS stars—about 4 per square arcminute—is not high enough to build many tiny quads.

SDSS within a 2×2 degree square centered on RA = 215 degree, Dec = 52.7 degree with measured r-band brightness between 15 and 22.2 mag, yielding about $57,000$ sources. We created the index as usual, using a HEALPix grid with cells of size 0.5 arcmin, and building quads with diameters between 0.5 and 2 arcmin. This yielded just over $100,000$ quads. Since the density of stars and galaxies in our index is only about 4 sources per square arcminute, the system tended to produce quads with sizes strongly skewed toward the larger end of the allowed range of scales. See figure 2.23.

2.73 We queried the Hubble Legacy Archive [36] for images taken by the Hubble Advanced Camera for Surveys (ACS) [23] within the AEGIS footprint. Since individual ACS exposures contain many cosmic rays, we requested only "level 2" images, which are created from multiple exposures and have cosmic rays removed. A total of 191 such images were found. We retrieved 600×600-pixel JPEG previews—see figure 2.24 for an example—and gave these images to our system as inputs.

2.74 Our system successfully recognized 100 % of the 191 input images, taking an average of 0.3 seconds of CPU time per image (not including the time required to perform source extraction). The footprints of the input images are shown in figure 2.24. Although there are 191 images, there are only 64 unique footprints, because images were taken through several different bandpass filters for many of the footprints. Although the index contained quads with diameters from 0.5 to 2 arcmin, the smallest quad that was used to recognize a field was about 0.9 arcmin in diameter.

2.3.4 Astrometric calibration of other imagery

2.75 The Harvard Observatory archives contain over $500,000$ photographic glass plates exposed between 1880 and 1985. See figure 2.25 for an example. The Digital Access to a Sky-Century at Harvard (DASCH) project is in the process of scanning these plates at high resolution [50, 27]. Since the original astrometric calibration for these plates consists of hand-written entries in log books, a full-sky astrometric calibration system was

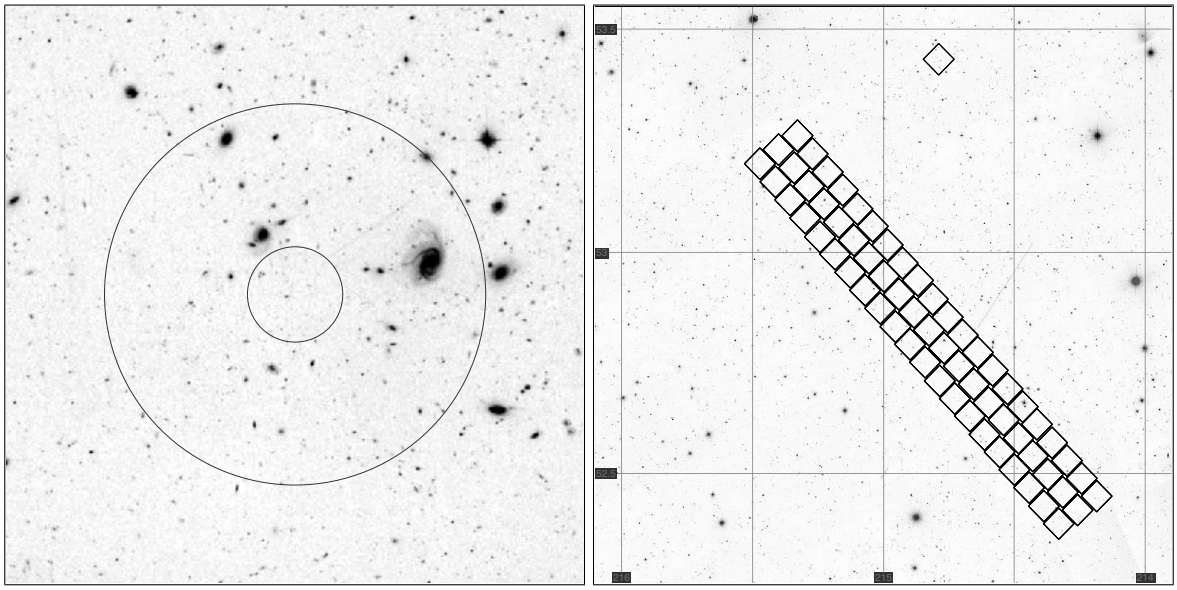


Figure 2.24: **Left:** A typical cutout of a Hubble Space Telescope Advanced Camera for Surveys (ACS) image as used in our experiment. The overlaid circles show the range of diameters of the quads in the index we use to recognize these images. ACS images are about 3.4 arcmin square and 4096×4096 pixels, but the cutouts we use in our experiment are about 3 arcmin and have been downsampled to 600×600 pixels. The quad diameters are from 0.5 to 2 arcmin. Image credit: Courtesy NASA/JPL-Caltech. **Right:** A ~ 2 square degree region (part of the ~ 4 square degree region of SDSS from which we built our index), overlaid with the footprints of the 191 ACS images that were recognized by our system. There are only 64 unique footprint regions because some of the 191 images are observations of the same region through different bandpass filters. The grid lines show RA and Dec.

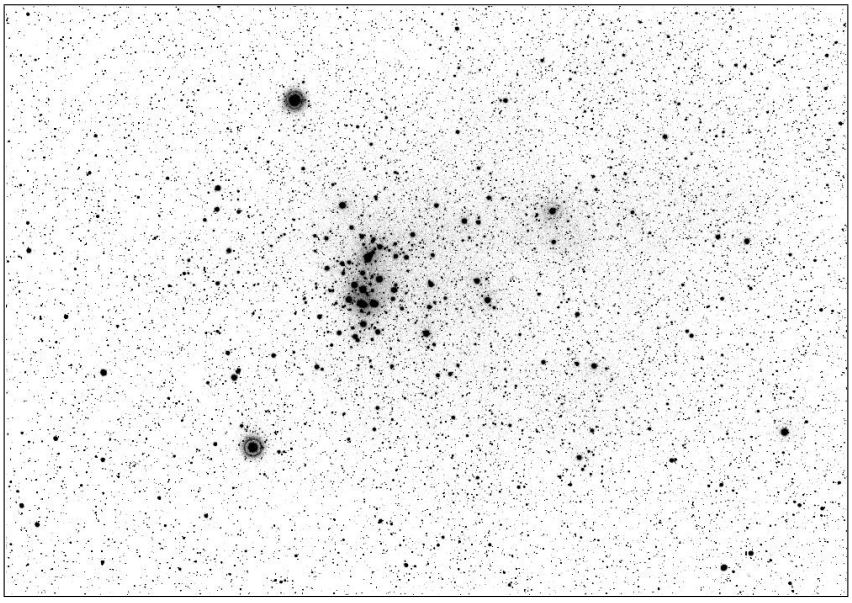


Figure 2.25: A sample DASCH scan of part of one of the photographic glass plates of the Harvard Observatory archives. The initial astrometric calibration of these plates is being computed by *Astrometry.net*. Praesepe, also known as the Beehive cluster or Messier 44, appears in this image. Image credit: DASCH team; Harvard College Observatory.

required to add calibration information to the digitized images. DASCH has been using the *Astrometry.net* system for the past year to create an initial astrometric calibration which is then refined by WCSTools [51].

2.76 The DeepSky project [55] is reprocessing the data taken as part of the Palomar-QUEST sky survey and Nearby Supernova Factory [20, 2]. Since many of the images have incorrect astrometric meta-data, they are using *Astrometry.net* to do the astrometric calibration. Over 14 million images have been successfully processed thus far (Peter Nugent, personal communication).

2.77 We have also had excellent success using the same system for calibrating a wide class of other astronomical images, including amateur telescope shots, photographs from consumer digital SLR cameras, some of which span tens of degrees. We have also calibrated videos posted to YouTube. These images have very different exposure properties, capture light in the optical, infrared and ultraviolet bands, and often have significant distortions away from the pure tangent-plane projection of an ideal camera. Part of the remarkable robustness of our algorithm, which allows it to calibrate all such images using the same parameter settings, comes from the fact that the hash function is scale invariant so that even if the center of an image and the edges have a significantly different pixel scale (solid angle per pixel), quads in both locations will match properly into the index (although our verification criterion may conservatively decide that a true solution with substantial distortion is not correct). Furthermore, no individual quad or star, either in the query or the index is essential to success. If we miss some evidence in one part of the image or the sky we have many more chances to find it elsewhere.

2.3.5 False positives

2.78 Although we set our operating thresholds to be very conservative in order to avoid false positive matches, images that do not conform to the assumptions in our model can yield false positive matches at much higher rates than predicted by our analysis. In

particular, we have found that images containing linear features that result in lines of detected sources are often matched to linear flaws in the USNO-B reference catalog. We removed linear flaws resulting from diffraction spikes [7], but many other linear flaws remain. An example is shown in figures 2.26 and 2.27.

2.4 Discussion

2.79 We have described a system that performs astrometric calibration—determination of imaging pointing, orientation, and plate scale—without any prior information beyond the data in the image pixels. This system works by using indexed asterisms to generate hypotheses, followed by quantitative verification of those hypotheses. The system makes it possible to vet or restore the astrometric calibration information for astronomical images of unknown provenance, and images for which the astrometric calibration is lost, unknown, or untrustworthy.

2.80 There are several sources of astronomical imagery that could be very useful to researchers—especially those studying the time domain—if they were consistently and correctly calibrated. These include photographic archives, amateur astronomers, and a significant fraction of professional imagery which has incorrect or no astrometric calibration meta-data. The photographic archives extend the time baseline up to a century, while amateur astronomers—some of whom are highly-skilled and well-equipped—can provide a dense sampling of the time domain and can dedicate a large amount of observing time to individual targets. Our system allows these sources of data to be made available to researchers by creating trustworthy astometric calibration meta-data.

2.81 The issue of trust is key to the success of efforts such as the Virtual Observatory to publish large, heterogeneous collections of data produced by many groups and individuals. Without trusted meta-data, data are useless for most purposes. Our system allows existing astrometric calibration meta-data to be verified, or new trustworthy meta-data to

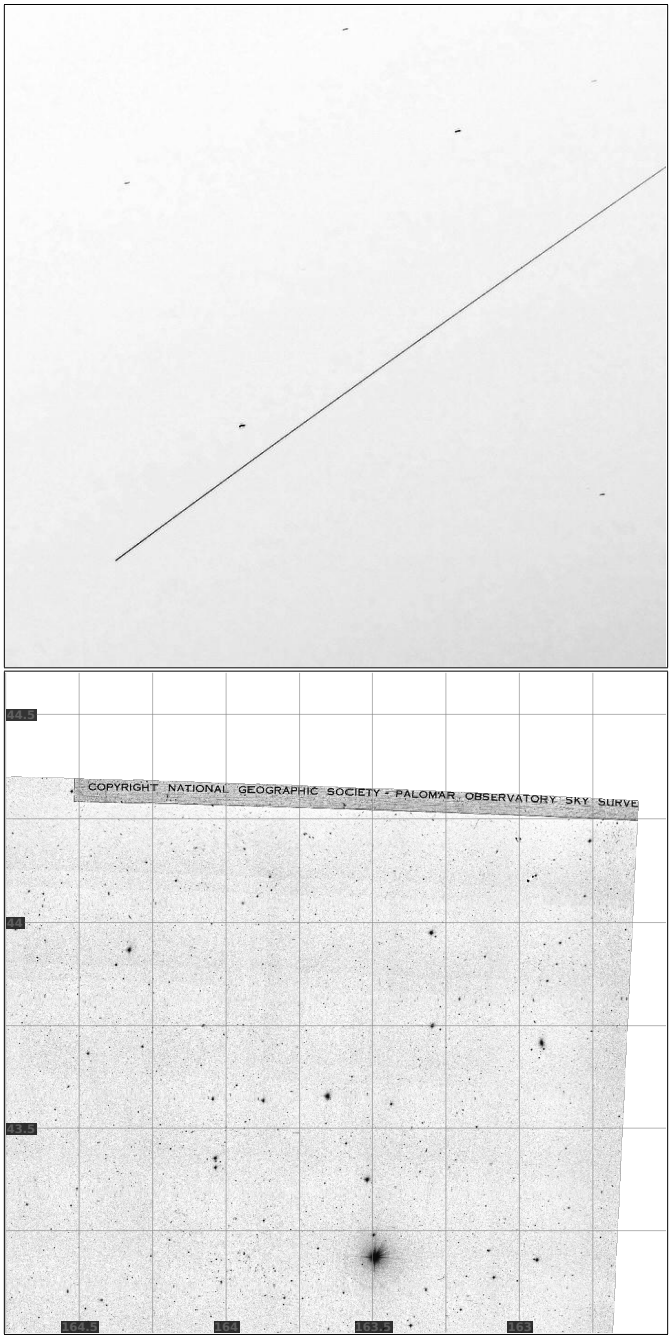


Figure 2.26: A false-positive match. **Top:** The input image contains a linear feature: the International Space Station streaked across the image. Image credit: copyright Massimo Matassi. **Bottom:** The USNO-B scanned photographic plate has writing on the corner. Image credit: copyright Palomar Observatory, National Geographic Society, and California Institute of Technology; courtesy of USNO Image and Catalogue Archive.

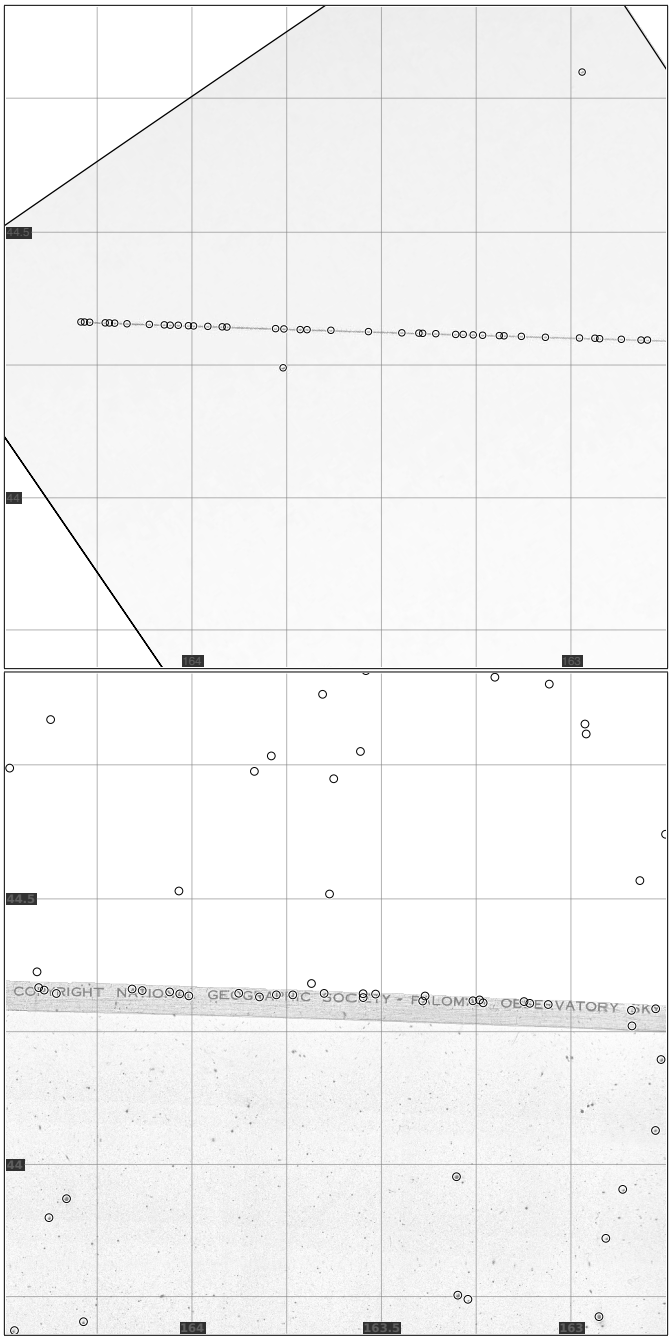


Figure 2.27: A false-positive match (continued). **Top:** The image, rotated to the alignment that our system found. The circles show the sources that were detected. The linear feature becomes a line of false sources. **Bottom:** The corresponding region of the USNO-B plate. The USNO-B source detection algorithm identifies many false sources in the region of text. The lines of sources in the two images are aligned in this (false) match.

be created, from the data. Furthermore, applying a principled and consistent calibration procedure to heterogeneous collections of images enables the kinds of large-scale statistical studies that are made possible by the Virtual Observatory.

2.82 Our experiments on Sloan Digital Sky Survey, Galaxy Evolution Explorer, and Hubble Space Telescope images have demonstrated the capabilities and limitations of our system. The best performance, in terms of the fraction of images that are recognized and the computation effort required, is achieved when the index of known asterisms is well-matched to the images to be recognized. Differences in the bandpasses of the index and images lead to a small drop in performance across the near-infrared to near-ultraviolet range. By creating multiple indices across the spectrum we could overcome this limitation, if suitable reference catalogs were available. The image quality has some effect on the performance, though our experiments using the quality ratings assigned by the SDSS image-processing pipeline do not fully explore this space since the quality ratings are in terms of the very high standards of the survey: even the "bad" images are reasonable and over 99 % of them can be recognized by our system. More experiments on images with poorly-localized sources would better characterize the behavior of the system on low-quality images.

2.83 In order to handle images across a wide range of scales, we build a series of indices, each of which is specialized to a narrow range of scales. Each index is able to recognize images within, and extending somewhat outside, the range to which it is tuned, but the drop-off in performance is quite fast: the index we used in the majority of our experiments works very well on 13×9 arcmin^2 SDSS images and sub-images down to 8×8 arcmin^2 but suffers a serious drop in performance on 7×7 arcmin^2 images. Similarly, the computational effort required is sharply reduced if the system is given hints about the scale of the images to be recognized. This is driven by three main factors. First, any index that contains quads that cannot possibly be found in an image of the given range of scales need not be examined. Second, only quad features of a given range of scales in

the image need be tested. Finally, every quad in the image that is matched to a quad in the index implies an image scale, and any scale outside the allowed range can be rejected without running the verification procedure.

2.84 Using a set of indices, each of which is tuned to a range of scales, is related to the idea of *co-visibility constraints* in computer vision systems: "closely located objects are likely to be seen simultaneously more often than distant objects" [68]. Each of our indices contains the brightest stars within grid cells of a particular size, and contains quad features at a similar scale, so quads of large angular extent can only be built from bright stars, while small quads can be built from quite faint stars. This captures the practical fact that the angular scale of an image largely determines the brightnesses of the stars it contains. Distant pairs of faint stars are very unlikely to appear in the same image, and we take advantage of this constraint by only using faint stars to build quads of small angular size.

2.85 The index used in most of our experiments covers the sky in a dense blanket of quads. This means that in any image, we have many chances of finding a matching quad, even if some stars are missing from the image or index. This comes at the cost of increasing the number of features packed into our code feature space, and therefore the number of false matches that are found for any given quad in a test image. Reducing the number of quads means that each query will be faster, but more queries will typically be required before a match is found.

2.86 Our experiment using indices built from triangles and quintuples of stars shows that, for SDSS images, our geometric features built from quadruples of stars make a good tradeoff between being distinctive enough that the feature space is not packed too tightly, yet having few enough stars that the probability of finding all four stars in both the image and index is high. We expect that for images much larger in angular size than SDSS images, a triangle-based index might perform better, and for images smaller than SDSS images, a quint-based index might be superior.

2.87 Similarly, we found that using a voting scheme—requiring two or more hypotheses to agree before running the relatively expensive verification step—was slower than simply running the verification process on each hypothesis, when using a quad-based index and SDSS-sized images. In other domains (such as triangle-based indices), a voting scheme could be beneficial.

2.88 Although we have focused on the idea of a system that can recognize images of any scale from any part of the sky, our experiments on Hubble Space Telescope images demonstrate that by building a specialized index that covers only a tiny part of the sky, we can recognize tiny images that contain only a few stars that appear in even the deepest all-sky reference catalogs.

2.89 Our system, built on the idea of geometric hashing—generating promising hypotheses using a small number of stars and checking the hypotheses using all the stars—allows fast and robust recognition and astrometric calibration of a wide variety of astronomical images. The recognition rate is above 99.9 % for high-quality images, with no false positives. Other researchers have begun using *Astrometry.net* to bring otherwise "hidden" data to light, and we hope to continue our mission "to help organize, annotate and make searchable all the world's astronomical information."

2.90 All of the code for the *Astrometry.net* system is available under an open-source license, and we are also operating a web service. See `http://astrometry.net` for details.

Acknowledgements

2.91 We thank Jon Barron, Doug Finkbeiner, Chris Kochanek, Robert Lupton, Phil Marshall, John Moustakas, Peter Nugent, and Christopher Stumm for comments on and contributions to the prototype version of the online service *Astrometry.net*. It is a pleasure to thank also our large alpha-testing team. We thank Leslie Groer and the University of Toronto Physics Department for use of their Bigmac computer cluster, and Dave Monet for maintaining and helping us with the USNO-B Catalog. Lang, Mierle and Roweis were

funded in part by NSERC and CRC. Hogg was funded in part by the National Aeronautics and Space Administration (NASA grants NAG5-11669 and NNX08AJ48G), the National Science Foundation (grants AST-0428465 and AST-0908357), and the Alexander von Humboldt Foundation. Hogg and Blanton were funded in part by the NASA *Spitzer Space Telescope* (grants 30842 and 50568).

2.92 This project made use of public SDSS data. Funding for the SDSS and SDSS-II has been provided by the Alfred P. Sloan Foundation, the Participating Institutions, the National Science Foundation, the U.S. Department of Energy, the National Aeronautics and Space Administration, the Japanese Monbukagakusho, the Max Planck Society, and the Higher Education Funding Council for England. The SDSS Web Site is http://www.sdss.org/.

2.93 The SDSS is managed by the Astrophysical Research Consortium for the Participating Institutions. The Participating Institutions are the American Museum of Natural History, Astrophysical Institute Potsdam, University of Basel, University of Cambridge, Case Western Reserve University, University of Chicago, Drexel University, Fermilab, the Institute for Advanced Study, the Japan Participation Group, Johns Hopkins University, the Joint Institute for Nuclear Astrophysics, the Kavli Institute for Particle Astrophysics and Cosmology, the Korean Scientist Group, the Chinese Academy of Sciences (LAMOST), Los Alamos National Laboratory, the Max-Planck-Institute for Astronomy (MPIA), the Max-Planck-Institute for Astrophysics (MPA), New Mexico State University, Ohio State University, University of Pittsburgh, University of Portsmouth, Princeton University, the United States Naval Observatory, and the University of Washington.

2.94 This publication makes use of data products from the Two Micron All Sky Survey, which is a joint project of the University of Massachusetts and the Infrared Processing and Analysis Center/California Institute of Technology, funded by the National Aeronautics and Space Administration and the National Science Foundation.

2.95 Based on observations made with the NASA/ESA Hubble Space Telescope, and ob-

tained from the Hubble Legacy Archive, which is a collaboration between the Space Telescope Science Institute (STScI/NASA), the Space Telescope European Coordinating Facility (ST-ECF/ESA) and the Canadian Astronomy Data Centre (CADC/NRC/CSA).

2.96 This research made use of the NASA Astrophysics Data System.

2.97 This research has made use of the USNOFS Image and Catalogue Archive operated by the United States Naval Observatory, Flagstaff Station (http://www.nofs.navy.mil/data/fchpix/).

Chapter 3

Verifying a proposed alignment between astronomical images

3.1 Introduction

3.1 Suppose we are given two lists of astronomical sources ("stars", hereafter) and a hypothesized astrometric alignment between them. We wish to produce a probabilistic assessment of whether the alignment is true or false, where a false alignment might nonetheless show some coincidentally matching stars. For example, one list might be an astrometric reference catalog and the other a list of stars detected in an image, and the hypothesized alignment could be a World Coordinate System (WCS) transformation that maps pixel coordinates in the image to celestial coordinates in the reference catalog. We then wish to assess whether applying the given transformation to the stars detected in the image causes them to align with the stars in the reference catalog.

3.2 We assume that the lists of stars include the positions, positional uncertainties, and an approximate brightness ordering. We assume that the positional errors are drawn from known probability distributions that are correctly described by the positional uncertainties. In this chapter we will assume the positional errors are drawn from Gaussian

distributions whose variances are given by the positional uncertainties.

3.3 In true alignments, some fraction of the stars in each list will have no counterpart in the other list. This could be due to occlusions in the images (from which the lists of stars are produced), artifacts or errors in detecting stars, or differences in the spectral bandpass or sensitivity of the two images. True stars can be missing from, and false stars added to, either list. The lists may be of quite different lengths due to different sensitivities and error rates.

3.4 Since our goal is to decide whether to accept or reject a proposed alignment, and we assume a probabilistic model, we can frame the problem in terms of Bayesian decision theory. This is briefly reviewed in the following section. We then proceed in section 3.3 to develop a simple model, point out the ways in which it fails and modify the model to handle these concerns. This problem is central to the *Astrometry.net* system, which uses geometric hashing to generate a large number of hypotheses which must be checked. We explore the specifics of the *Astrometry.net* application in section 3.4.

3.2 Bayesian decision-making

3.5 We wish to decide whether to accept or reject a proposed alignment. In effect, we are choosing between two models: a "foreground" model, in which the alignment is true, and a "background" model, in which the alignment is false. In Bayesian decision-making, three factors contribute to this decision: the relative abilities of the models to explain the observations, the relative proportions of true and false alignments we expect to see, and the relative costs or utilities of the outcomes resulting from our decision.

3.6 The *Bayes factor* K is a quantitative assessment of the relative abilities of the two models—the foreground model F and the background model B—to produce or explain the observations. In this case the observations, or data, D, are the two lists of stars. The

Bayes factor

$$K = \frac{p(D \mid F)}{p(D \mid B)} \tag{3.1}$$

is the ratio of the marginal likelihoods. We must also include in our decision-making the prior $p(F)/p(B)$, which is our *a priori* belief, expressed as a ratio of probabilities, that a proposed alignment is correct. In the *Astrometry.net* case, for example, we typically examine many more false alignments than true alignments, so this ratio will be small.

3.7 The final component of Bayesian decision theory is the *utility* table, which expresses the subjective value of each outcome. In this problem, there are four possible outcomes: If the alignment we are considering is true and we decide to accept it, the outcome is a true positive ("TP"), while if we reject it we generate a false negative ("FN"). If the alignment is false and we accept it, we produce a false positive outcome ("FP"), while if we reject it we get a true negative ("TN"). The utility table, as shown below, assigns a value $u(\cdot)$ to each of these outcomes.

		True Alignment?	
		Yes	No
Decision	Accept	$u(\mathrm{TP})$	$u(\mathrm{FP})$
	Reject	$u(\mathrm{FN})$	$u(\mathrm{TN})$

3.8 We make our decision to accept or reject the proposal by computing the expected utility $\mathbb{E}\,[\,u\,]$ of each decision. The expected utility of accepting the alignment is:

$$\mathbb{E}\,[\,u \mid \mathrm{Accept}, D\,] = u(\mathrm{TP})\,p(\mathrm{TP} \mid D) + u(\mathrm{FP})\,p(\mathrm{FP} \mid D) \tag{3.2}$$

$$= u(\mathrm{TP})\,p(F \mid D) + u(\mathrm{FP})\,p(B \mid D) \tag{3.3}$$

while the expected utility of rejecting the alignment is:

$$\mathbb{E}\,[\,u \mid \mathrm{Reject}, D\,] = u(\mathrm{FN})\,p(\mathrm{FN} \mid D) + u(\mathrm{TN})\,p(\mathrm{TN} \mid D) \tag{3.4}$$

$$= u(\mathrm{FN})\,p(F \mid D) + u(\mathrm{TN})\,p(B \mid D) \quad . \tag{3.5}$$

We should accept the hypothesized alignment if:

$$\mathbb{E}\,[\,u\,|\,\mathrm{Accept}, D\,] > \mathbb{E}\,[\,u\,|\,\mathrm{Reject}, D\,] \tag{3.6}$$

$$u(\mathrm{TP})\,p(F\,|\,D) + u(\mathrm{FP})\,p(B\,|\,D) > u(\mathrm{FN})\,p(F\,|\,D) + u(\mathrm{TN})\,p(B\,|\,D) \tag{3.7}$$

$$\frac{p(F\,|\,D)}{p(B\,|\,D)} > \frac{u(\mathrm{TN}) - u(\mathrm{FP})}{u(\mathrm{TP}) - u(\mathrm{FN})} \tag{3.8}$$

$$K > \frac{p(B)}{p(F)}\;\frac{u(\mathrm{TN}) - u(\mathrm{FP})}{u(\mathrm{TP}) - u(\mathrm{FN})} \tag{3.9}$$

where we have applied Bayes' theorem to get

$$\frac{p(F\,|\,D)}{p(B\,|\,D)} = K\;\frac{p(F)}{p(B)} \quad . \tag{3.10}$$

3.9 That is, we accept or reject a proposed alignment by thresholding the Bayes factor of the foreground model to the background model. The threshold is set based on our desired operating point for the particular application. Increasing the threshold will cause us to reject more proposals, thus changing some false positives to true negatives (increasing the utility), while changing some true positives to false negatives (decreasing the utility).

3.10 There is an arbitrary overall scaling of the utility values, since in equation 3.9 only the ratios of differences of utilities appear. When setting utility values, it may be convenient to set $u(\mathrm{TP}) = 1$ and choose the other utilities relative to that. In that case, the $u(\mathrm{FP})$ and $u(\mathrm{FN})$ values may be negative. In the *Astrometry.net* case, for example, we want strongly to avoid false positives, so the ratio of utility differences is dominated by $u(\mathrm{FP}) \ll 0$, and combined with the small prior $p(F)/p(B)$ mentioned earlier, we find that K must be very large: we demand that the foreground model be an overwhelmingly better explanation of the data before accepting the hypothesis.

3.3 A simple independence model

3.11 In this section we consider one of the lists to be a fixed "reference" list, and develop foreground and background models in which each star in the other "test" list is treated

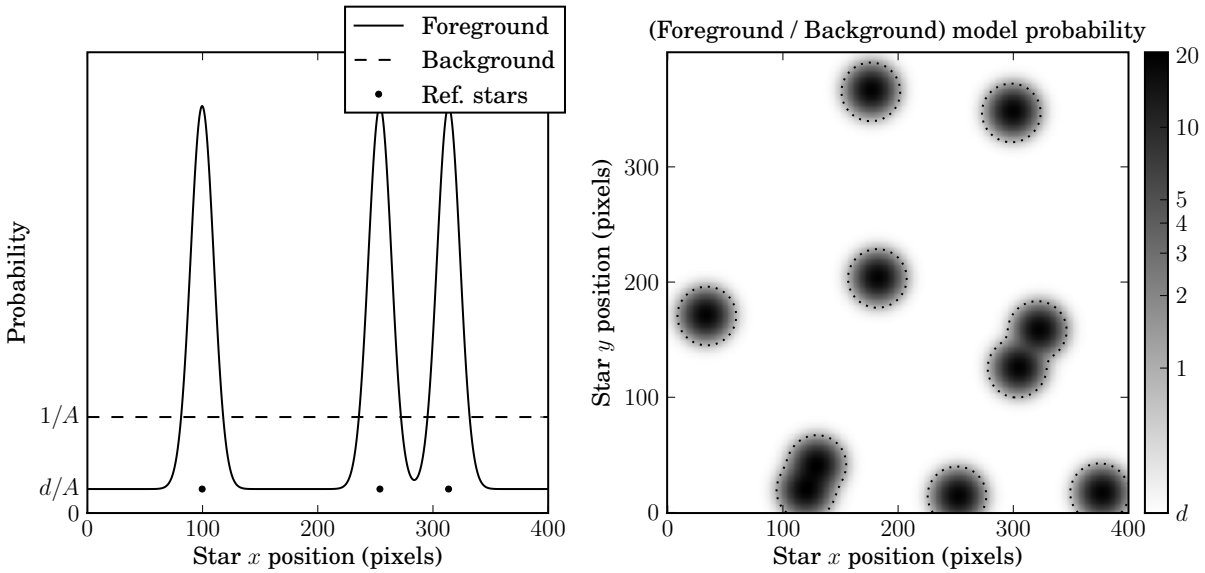


Figure 3.1: **Left:** one-dimensional slice of the simple foreground and background probability distributions F_1 and B_1. The background model is uniform, while the foreground model has a uniform floor plus Gaussian peaks around each reference star. **Right:** two-dimensional probability density ratio of the foreground to background models—the Bayes factor terrain for a single test star—on a log scale. The contours indicate where the models are equal. In these plots, the distractor fraction $d = 0.25$, $\sigma_{i,j}^2 = 10$, and the image size is 400×400 pixels. This value of $\sigma_{i,j}^2$ is larger than is typical in astronomical images in order to make the figures more clear.

as an independent sample from the model. In this case, the Bayes factor (equation 3.1) becomes:

$$K_1 = \frac{p(D \mid F)}{p(D \mid B)} = \prod_{i=1}^{N_t} \frac{p(t_i \mid F)}{p(t_i \mid B)} \tag{3.11}$$

where t_i is the ith star in the test list, and the reference list is used to define the foreground model. We have added the subscript K_1 to identify this definition of the Bayes factor: we will use different definitions in later sections.

3.12 In the foreground model, each test star is generated either by a reference star, or by an object that does not appear in the reference list: possibly a real star, possibly

an artifact. If a test star is generated by a reference star, then its observed position is distributed according to the positional variances of the reference and test stars. If a test star is not generated by a reference star, we assume it will be observed with uniform probability anywhere in the image. We call test stars that are not generated by reference stars "distractors." Distractor stars can be generated if the test star corresponds to a real star that does not appear in the reference list, or if the test star is actually an artificial satellite, a planet, a cosmic ray, a false positive in the star detection process, or some other kind of artifact. The distractor component of the model is a catch-all for artifacts and errors in both the test and reference lists. In general we expect stars to be both added and removed from both lists, but since it is difficult to determine whether a distractor is due to a star being added to the test list or removed from the reference list, we lump these cases together.

3.13 This foreground probability model, F_1, is thus a mixture of a uniform distribution of distractor stars, plus a Gaussian distribution around each reference star:

$$p(t_i \mid F_1) = \frac{d}{A} + \frac{1-d}{N_r} \sum_{j=1}^{N_r} \mathcal{N}(t_i \mid r_j, \sigma_{i,j}^2) \tag{3.12}$$

where d is the fraction of distractor stars, A is the area of the image, and $\{r_j\}$ are the N_r reference stars. $\mathcal{N}(x \mid \mu, \sigma^2)$ is the probability of drawing value x from the Gaussian distribution with mean μ and variance σ^2. Note that t_i and r_j are both two-dimensional vectors (positions in the image plane). We have written the combined positional variance of test star i and reference star j as $\sigma_{i,j}^2$.

3.14 In the background model, test stars are drawn without regard for the positions of the reference stars. The simplest model, B_1, is that the test stars are distributed uniformly across the image:

$$p(t_i \mid B_1) = \frac{1}{A} \tag{3.13}$$

where, as before, A is the area of the image. These models are illustrated in figure 3.1.

3.15 Observe that the background model probability density sums to unity over the area

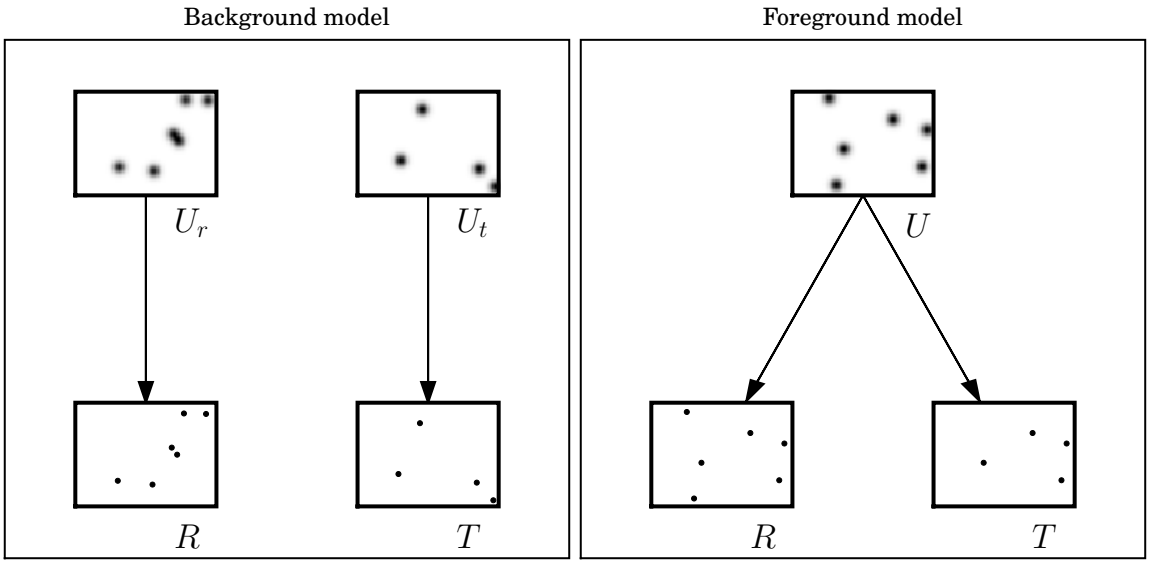


Figure 3.2: Left: "background" model: the reference star list R and test star list T are unrelated: each is generated by sampling from its own underlying probability density of stars (U_r and U_t, respectively). Right: "foreground" model: star lists R and T are two samples from a common underlying probability distribution of stars, U.

of the image, while the foreground model yields a value very slightly less than unity since the Gaussians around each reference star have infinite support but the image is finite. Since the positional variances in typical astronomical images are small relative to the image size, this effect is negligible.

3.3.1 Issues with this model

3.16 We will now present a number of different problems with this simple setting, adjusting the models to address each concern. Some of the scenarios we describe may seem contrived, but in *Astrometry.net* we have encountered most of them in real data.

3.3.1.1 Asymmetric model structure

3.17 One might ask why we have chosen a setting that treats reference and test stars

differently, as opposed to a setting such as that pictured in figure 3.2, in which reference and test stars are treated symmetrically. Instead of asking whether the reference stars predict the test stars, we could select between two models: in the background model, the reference and test lists are derived from independent underlying lists (*i.e.*, they are images of different parts of the sky), while in the foreground model they are derived from a single shared list (*i.e.*, they are images of the same part of the sky).

3.18 In this symmetric setting, the foreground and background model probabilities become

$$p(D \mid F_s) = \int \prod_{i=1}^{N_t} p(t_i \mid U, F_s) \prod_{j=1}^{N_r} p(r_j \mid U, F_s)\ p(U)\ \mathrm{d}U \tag{3.14}$$

$$p(D \mid B_s) = \int \prod_{i=1}^{N_t} p(t_i \mid U_t, B_s)\ p(U_t)\ \mathrm{d}U_t \int \prod_{j=1}^{N_r} p(r_j \mid U_r, B_s)\ p(U_r)\ \mathrm{d}U_r \tag{3.15}$$

where we must marginalize over the shared parameters U in the foreground model and the independent parameters U_r and U_t in the background model. The parameters U, U_r and U_t are the positions of stars in the underlying star lists.

3.19 Although it seems considerably more complex, this setting has similar behavior to the simpler asymmetric setting presented above. Assuming uniform priors over the positions of stars, we must have

$$p(U) = \prod_{i=1}^{N_u} p(u_i) = \prod_{i=1}^{N_u} \frac{1}{A} = \left(\frac{1}{A}\right)^{N_u} \tag{3.16}$$

and similarly for $p(U_r)$ and $p(U_t)$.

3.20 The background model should have one star in the underlying lists for each star in the reference and test lists, thus we would expect it to be able to explain each observation perfectly:

$$\int p(t_i \mid U_t, B_s)\ \mathrm{d}U_t = 1 \tag{3.17}$$

$$\int p(r_j \mid U_r, B_s)\ \mathrm{d}U_r = 1 \tag{3.18}$$

though if we are not careful, it is possible for one star in the underlying list to explain more than one observed star; we will discuss this in section 3.3.1.4.

3.21 The foreground model can be structured in different ways: one option is to include the positions of all the reference stars in U, then allow each test star to be described as either a new star seen only in the test list (requiring another star position to be added to the parameter set U, with a prior probability of $1/A$), or as one of the reference stars offset by some positional error:

$$\int p(t_i \,|\, U, F_s)\; p(U_i)\; \mathrm{d}U_i = \frac{d}{A} + \frac{1-d}{N_r} \sum_{j=1}^{N_r} \mathcal{N}(t_i \,|\, r_j,\, \sigma_{i,j}^2) \tag{3.19}$$

where U_i are the parameters that are added to U to explain test star t_i. Note the strong similarity to the simple asymmetric foreground model probability (equation 3.12).

3.22 We thus find that this symmetric setting leads to results that parallel the simpler asymmetric setting presented above. In the remainder of this chapter we will use the simple asymmetric setting.

3.3.1.2 Uniform background distribution

3.23 Non-uniform distributions of stars occur fairly often. At large angular scales, the galactic equator has much higher star density than the galactic poles. At smaller scales, astronomical sources of non-uniformity include star clusters, binary systems, galaxies and galaxy clusters, while dust features and nebulosity can cause non-uniformities in star detection.

3.24 Non-uniformity can result in overestimation of the Bayes factor (in favor of the foreground model), possibly resulting in a false positive. To see this, consider a reference star list that contains the stars in one globular cluster, and a test star lists that contains stars in a different globular cluster, where the cluster is centered in the image in both lists. Both lists will contain a concentration of stars near the center. If the number of reference stars or the positional variance is large, the individual peaks in the foreground probability model blend together, with the result that the foreground probability terrain has a flat background (the distractor component) plus a smooth peak near the center of

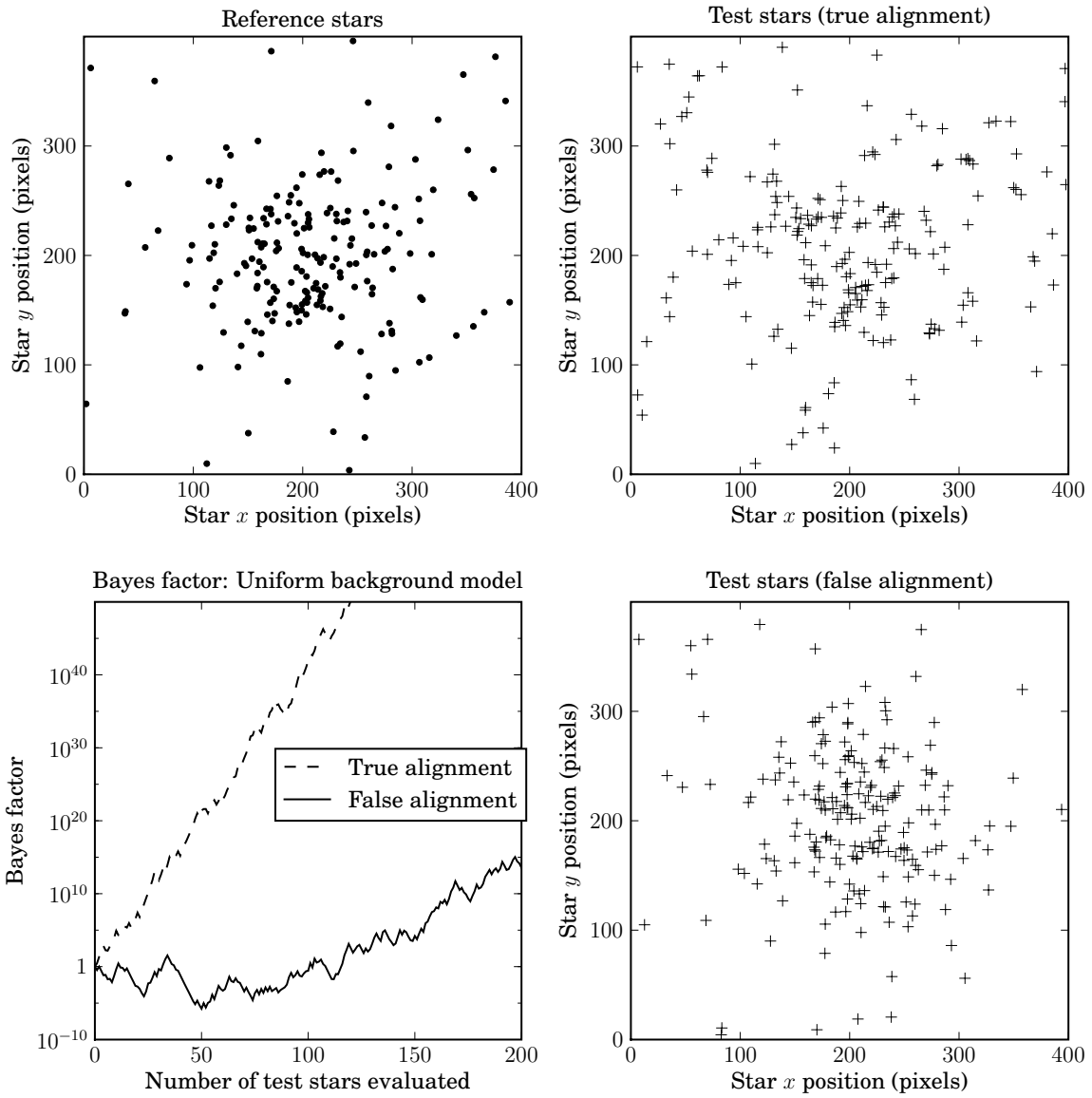


Figure 3.3: Globular cluster example. **Top-left:** Reference stars drawn from a simulated globular cluster: a mixture of 20% uniform plus 80% Gaussian clustered stars. **Top-right:** Test stars sampled from the reference stars (*i.e.*, a true alignment). **Bottom-right:** A different draw from the globular cluster distribution (*i.e.*, not a true alignment). **Bottom-left:** The Bayes factor K_1, using the uniform background model, as we evaluate test stars. The true alignment reaches a huge Bayes factor, but the false alignment reaches a very large Bayes factor—above 10^{15}—resulting in a false positive outcome.

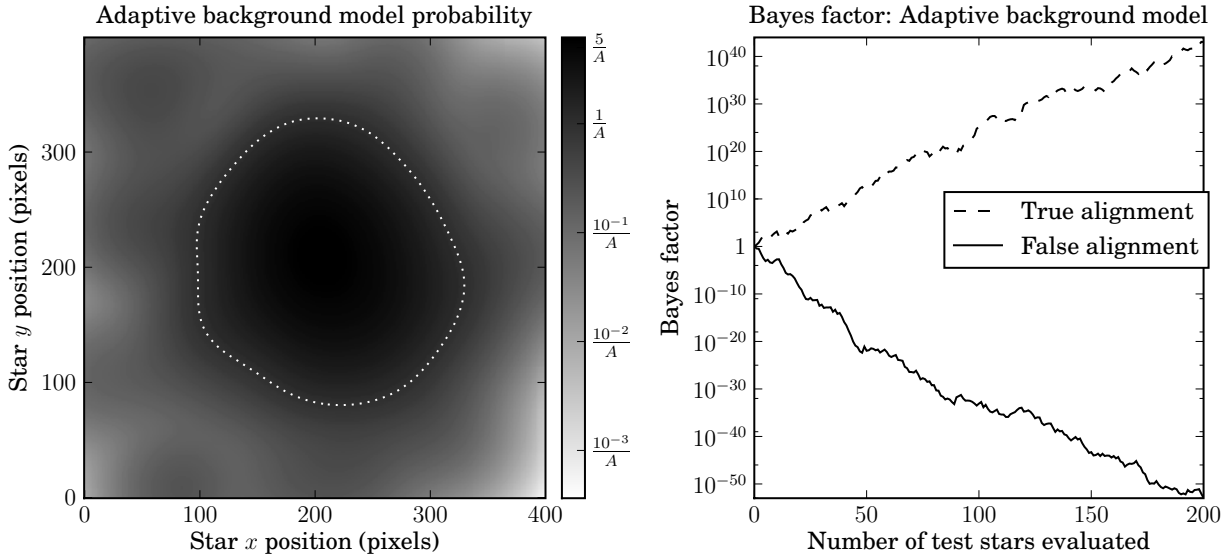


Figure 3.4: Globular cluster example, continued. **Left:** The adaptive background probability model. The contour line shows where the adaptive model is equal to the uniform model: the adaptive model concentrates its probability where the test star density is high, near the center of the image. **Right:** The Bayes factor, using the adaptive background model, as we evaluate test stars. With the adaptive background model, the true alignment is correctly accepted and the false alignment is correctly rejected.

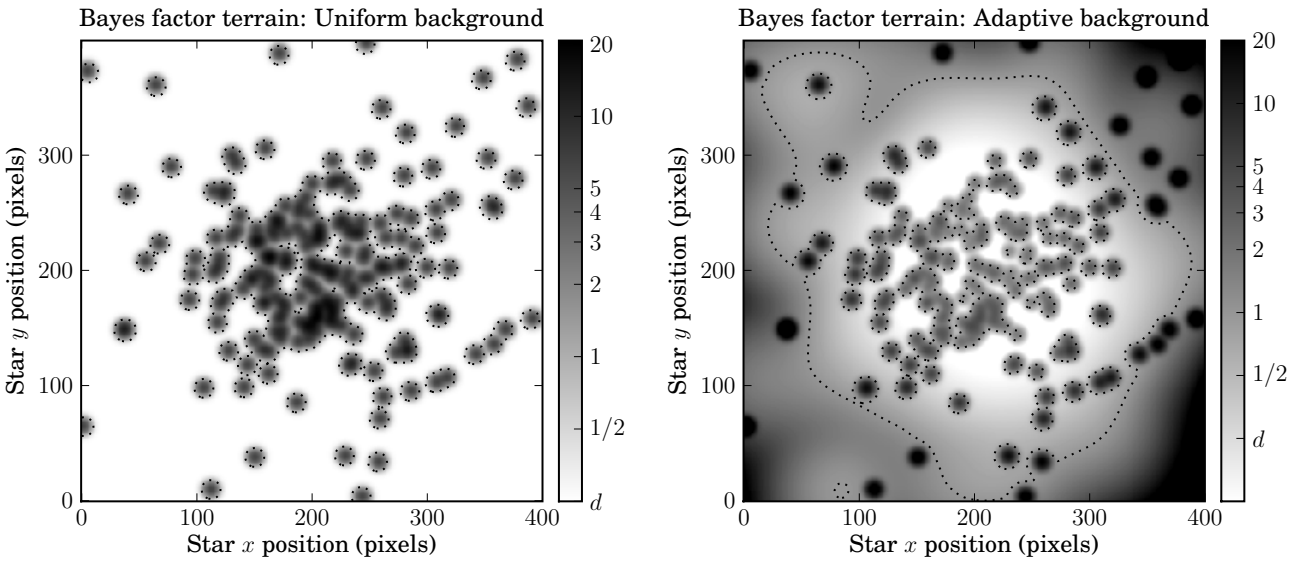


Figure 3.5: Globular cluster example, continued. **Left:** The Bayes factor terrain using the uniform background model (*i.e.*, the ratio of the foreground model to the uniform background model probability). Equality is indicated by contour lines. Nearly the entire central clustered region yields an increase in the Bayes factor in favor of the foreground model. Any test image that contains many stars in the central region is likely to be assigned a large Bayes factor, resulting in a false positive outcome. **Right:** The Bayes factor terrain using the adaptive background model. The adaptive background model concentrates its probability mass near the central dense region, so the foreground model must more precisely predict the positions of stars to achieve a gain in Bayes factor in that region, and even a perfect prediction yields only a modest gain. Using this model, the true alignment is correctly accepted and the false alignment is correctly rejected.

the image. This is quite a good model for the test stars—much better, in fact, than the uniform background model! Thus each test star will on average prefer the foreground model, and given enough test stars, the Bayes factor will slowly grow in favor of the foreground model. This is shown in figure 3.3.

3.25 This problem can be remedied by using a better background model. One approach is to use *kernel density estimation*—estimate the distribution of test points using the test points themselves. To be fair, the background model for test star i should not include any information about test star i, but it can use all test stars other than i, because they are assumed to be drawn independently.

3.26 For example, a flexible model for test star i is a sum of Gaussians around the other test stars, with variance proportional to the area of the image divided by the number of test stars:

$$p(t_i \mid B_a) = \frac{1}{N_t - 1} \sum_{\substack{j=1 \\ j \neq i}}^{N_t} \mathcal{N}(t_i \mid t_j,\, S^2) \tag{3.20}$$

$$S = 2\sqrt{\frac{A}{N_t \pi}} \tag{3.21}$$

so that stars are approximately distance S from their neighbors.

3.27 The result of using this adaptive background model in the globular cluster example is shown in figures 3.4 and 3.5. The adaptive background model can capture the non-uniformity of the test stars, so the foreground model must make significantly better predictions in areas of high stellar density in order to be preferred. Using the adaptive model, the false positive outcome demonstrated in figure 3.3 is eliminated.

3.3.1.3 Number of reference and test stars

3.28 The simple foreground model treats the reference stars as fixed and test stars as independent random draws from the model. The behavior of the model changes with the number of reference and test stars. Figures 3.6 and 3.7 show the different effects.

A reference list with few stars will have a strongly peaked probability terrain (because the total probability mass is split between fewer stars), thus each correctly matched test star results in a significant gain in Bayes factor. A reference list with many stars makes "softer" predictions and yields a more slowly-growing Bayes factor given a true alignment. If the test star list contains stars that cannot appear in the reference star list (for example, if the test list is a deeper exposure), then as more and more test stars are evaluated, they will most likely be assigned probability close to the distractor floor, and the Bayes factor will drop steadily. This suggests that we should select a reasonable number of reference stars (so that the foreground probability terrain is reasonably informative), and a reasonable number of test stars (so that we are not asking the foreground model to make predictions that it cannot be expected to answer correctly). We could instead marginalize over the number of stars in each list, but for our application this is too computationally expensive.

3.29 By introducing some approximations we can compute an estimate of the expected Bayes factor of the foreground model over the background model as we change the number of reference stars included in the model. That is, we can ask how strongly we will prefer the foreground model, in expectation, if we are given test stars that are truly drawn from the foreground model. In this section we use the uniform background model for ease of computation. The expected (log-) gain $\langle g \rangle$ is

$$\langle g \rangle = \mathbb{E}_{p(t \mid F)} \left[\log p(t \mid F) - \log p(t \mid B) \right] \tag{3.22}$$

$$= \int_{\mathcal{A}} p(t \mid F) \left[\log p(t \mid F) - \log p(t \mid B) \right] \mathrm{d}t \tag{3.23}$$

where $\mathcal{A}$ indicates that the integral is over the whole image area. This is equal to the Kullback–Leibler divergence [41]:

$$\langle g \rangle = D_{\mathrm{KL}}\Big(p(t \mid F) \,\Big\|\, p(t \mid B) \Big) \tag{3.24}$$

and, if we assume the uniform background model, is equal to a constant plus the entropy

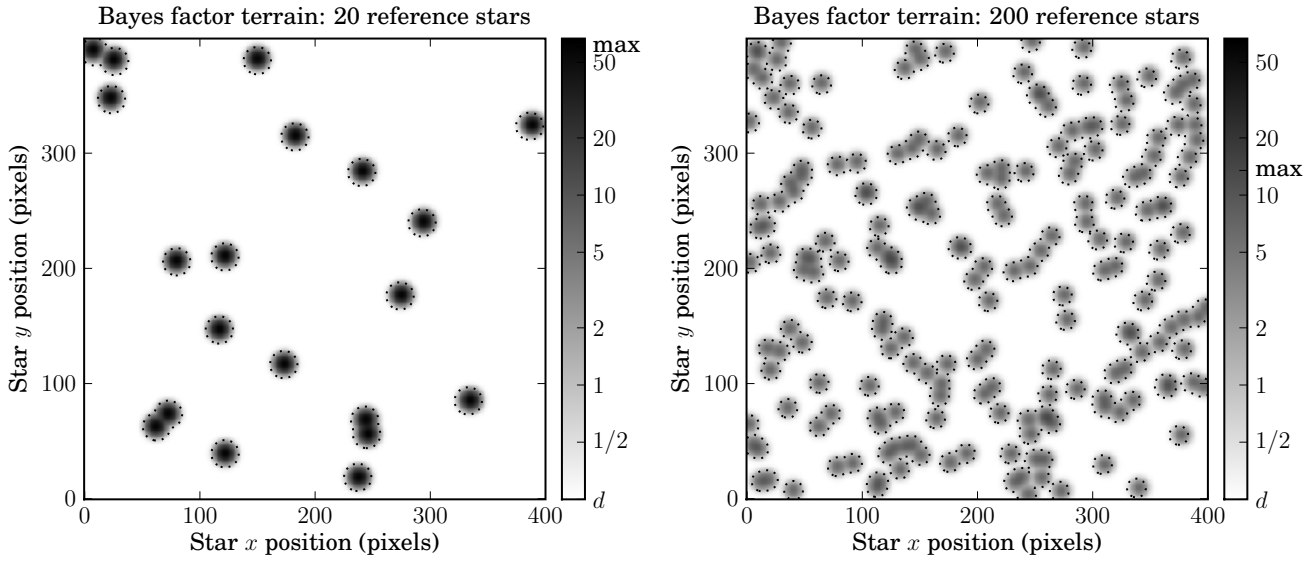


Figure 3.6: Informativeness of the foreground model as the number of reference stars changes. **Left:** Bayes factor terrain (ratio of foreground to background model probabilities) with 20 reference stars. **Right:** Bayes factor terrain with 200 reference stars. The same scale is used in both figures, and the maximum values attained in each terrain is marked. The model with 200 reference stars has a much smaller peak value: it makes "softer" predictions because it is forced to spread its probability mass among more peaks.

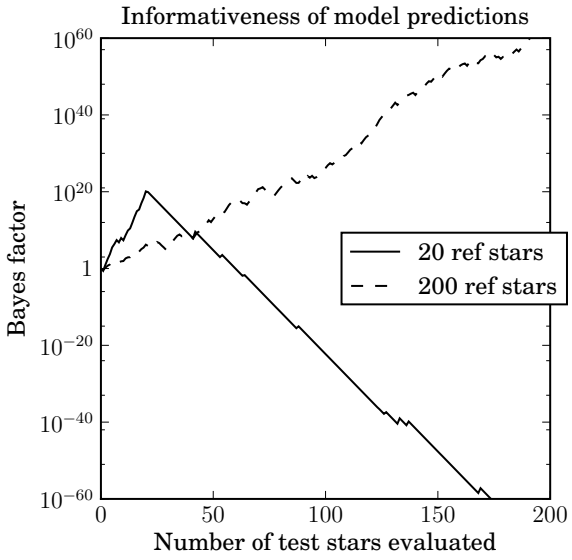


Figure 3.7: Informativeness of the foreground model as the number of reference stars changes (continued). Plotted is the Bayes factor as test stars drawn from the model are evaluated, for the two foreground models. The model with 20 reference stars yields a much faster-increasing Bayes factor (since the model's predictions are stronger), but of course it is unable to explain test stars past 20 and the Bayes factor drops. The model with 200 reference stars makes softer predictions and thus yields a more slowly-growing Bayes factor.

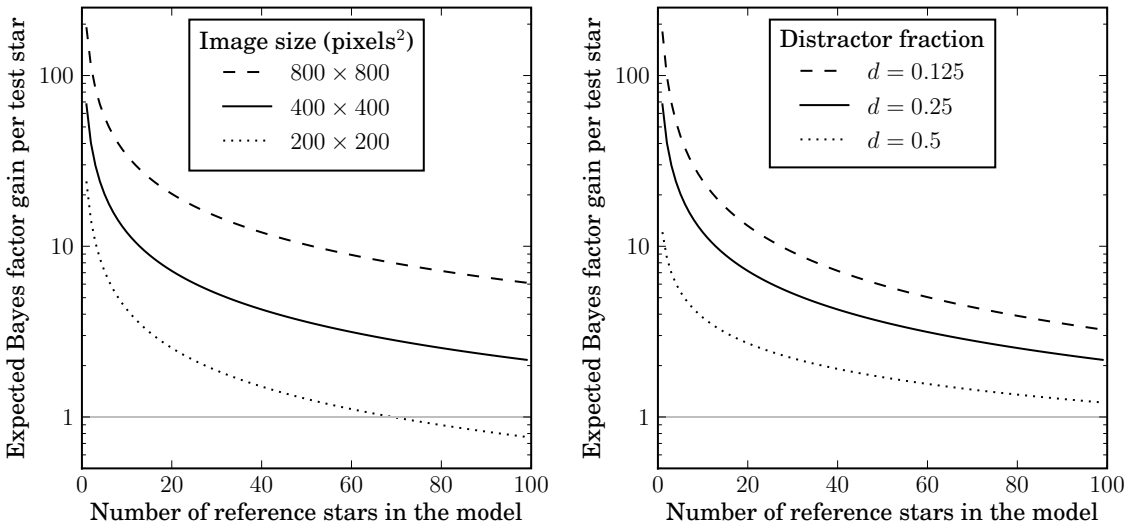


Figure 3.8: Informativeness of the foreground model. **Left:** Expected increase in the Bayes factor in favor of the foreground model, per test star ($\langle g \rangle$ in the text, equation 3.30). The middle curve shows the fiducial parameters used in the previous examples: distractor fraction $d = 0.25$, image size $A = 400 \times 400$ pixels, positional error $\sigma = 4$ pixels. The other curves show the effect of changing the image size by a factor of two in each dimension. Changing the positional uncertainty by a factor of two in the opposite direction has exactly the same effect, since this effectively just changes the units in which we work. The fact that these curves drop below unity (rather than asymptoting to unity) may indicate that the approximations used to derive them are violated in that regime. **Right:** Same, but changing the distractor fraction by factors of two.

of the foreground model distribution $H\big(p(t \mid F)\big)$:

$$\langle g \rangle = \int_{\mathcal{A}} p(t \mid F) \log p(t \mid F) \, \mathrm{d}t - \int_{\mathcal{A}} p(t \mid F) \log p(t \mid B_1) \, \mathrm{d}t \tag{3.25}$$

$$= H\big(p(t \mid F)\big) + \log A \tag{3.26}$$

which simply formalizes our notion of the "sharpness" or strength of the predictions made by the foreground model.

3.30 In the regime where the number of reference stars N_r is small and the reference stars

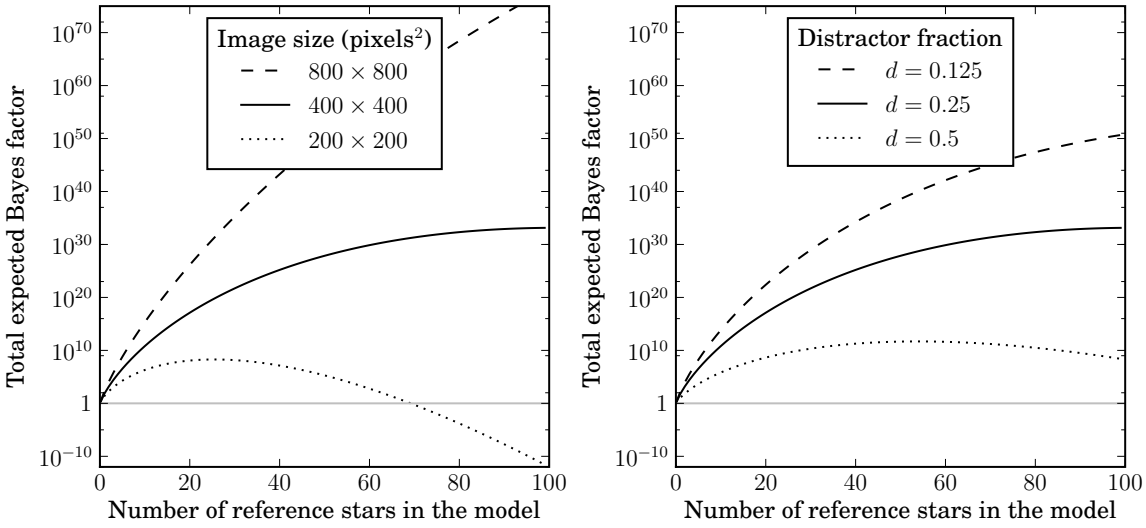


Figure 3.9: Informativeness of the foreground model. Plotted is the expected total Bayes factor after evaluating as many test stars as there are reference stars in the model (*i.e.*, $N_r\langle g \rangle$). **Left:** Increasing the size of the image makes the foreground model relatively more informative, because the background model is forced to spread its probability mass over a larger area. **Right:** Increasing the distractor fraction causes the foreground model to become less informative, because the model spends more of its probability mass "hedging its bets" in the distractor component of the mixture.

are far apart relative to the positional variance σ, the terms in the foreground model approximately decouple:

$$\langle g \rangle = \mathbb{E}_{p(t\,|\,F_1)} \left[\log p(t\,|\,F_1) - \log p(t\,|\,B_1) \right] \tag{3.27}$$

$$= \int_{\mathcal{A}} \left(\frac{d}{A} + \frac{1-d}{N_r} \sum_{j=1}^{N_r} \mathcal{N}(t\,|\,r_j,\,\sigma^2) \right) \log \left(\frac{d}{A} + \frac{1-d}{N_r} \sum_{j=1}^{N_r} \mathcal{N}(t\,|\,r_j,\,\sigma^2) \right) \mathrm{d}t$$

$$+ \log A \tag{3.28}$$

$$\approx \log A + \int_{\mathcal{A}} \frac{d}{A} \log \frac{d}{A} \, \mathrm{d}t$$

$$+ \int_{\mathcal{A}} \left(\frac{1-d}{N_r} \sum_{j=1}^{N_r} \mathcal{N}(t\,|\,r_j,\,\sigma^2) \right) \log \left(\frac{1-d}{N_r} \sum_{j=1}^{N_r} \mathcal{N}(t\,|\,r_j,\,\sigma^2) \right) \mathrm{d}t \tag{3.29}$$

$$\langle g \rangle \approx d \log d + (1-d) \left(\log \frac{A(1-d)}{2\pi\sigma^2 N_r} - 1 \right) \quad . \tag{3.30}$$

3.31 Since we expect that a model with N_r reference stars should be able to predict about N_r test stars, we can choose N_r to maximize the total expected Bayes factor after adding $N_t = N_r$ test stars:

$$N_r{}^* = \operatorname*{argmax}_{N_r} N_r \langle g \rangle \tag{3.31}$$

$$N_r{}^* \approx \frac{A(1-d)}{2\pi\sigma^2} \exp \left(\frac{d \log d}{1-d} - 2 \right) \tag{3.32}$$

which is plotted for various parameter settings in Figures 3.8 and 3.9.

3.32 This analysis suggests that we should use a rather large number of reference stars, yielding a modest gain in Bayes factor for each test star evaluated but a huge total Bayes factor after evaluating many stars. However, in astronomical images, we typically do not have the suggested number of reference and test stars. For example, for Sloan Digital Sky Survey images it suggests we should use over $30,000$ stars, but typically fewer than 1000 are available. How should we choose the number of stars to use when relatively few are available?

3.33 If the reference and test lists had exactly the same brightness ordering, and the simple constant distractor rate model were correct, we would simply choose to keep the number

of stars in the smaller of the two lists. However, neither of these conditions hold in practice: even if the reference and test list are derived from images taken in identical conditions—with the same telescope, camera, filter, and exposure time—we still expect that random measurement noise may lead to different brightness orderings, and different artifacts and errors in detecting stars may lead to different stars being added and removed. When the two lists are derived from imaging taken in different wavelength bandpasses, the true brightness orderings will likely be different, since astronomical objects have a wide range of spectral energy distributions.

3.34 In practice, we might wish to keep the number of stars in the smaller of the two lists and add a margin to account for the differences in brightness ordering and distractor stars. Alternatively, we could marginalize over the numbers of stars. Since the ultimate goal is a *decision* rather than a precise assessment of the Bayes factor, we have some flexibility.

3.3.1.4 Independent samples

3.35 In the simple model setting, nothing prevents multiple test stars from matching to a single reference star, or vice versa. That is, a test star can be found in a region of the image where the probability peaks of multiple reference stars overlap, or multiple test stars can be found within the probability peak of a single reference star. This can be problematic when one or both of the lists contains structured artifacts. We call this the "lucky donut" effect, since we first saw it when we were handling out-of-focus images where each star appeared as a ring in the image. Our source extraction procedure produced a "donut" of detections, and in some cases these donuts of test stars happened to occur near a reference star. When this happened, the simple foreground model, which assumes independent draws, could be strongly preferred when the alignment was false, resulting in a false positive outcome. See figures 3.10 and 3.11.

3.36 In effect, these donuts increase the variance of our Bayes factor estimation. Each

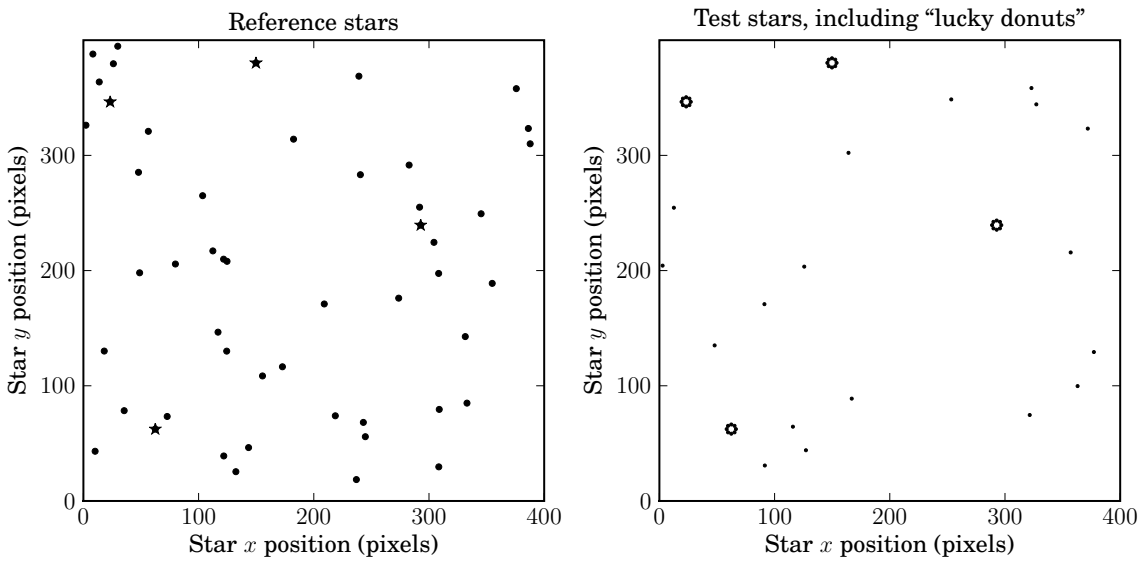


Figure 3.10: "Lucky donut" example. **Left:** Reference stars (with donut centers marked with star symbols). **Right:** Test stars. The reference stars are drawn from a uniform distribution. The test stars include 4 donuts of 8 stars each, each centered on a reference star and with radius 4 pixels; the remainder of the stars are drawn uniformly. Each list contains 50 stars total. The results are shown in the next figure.

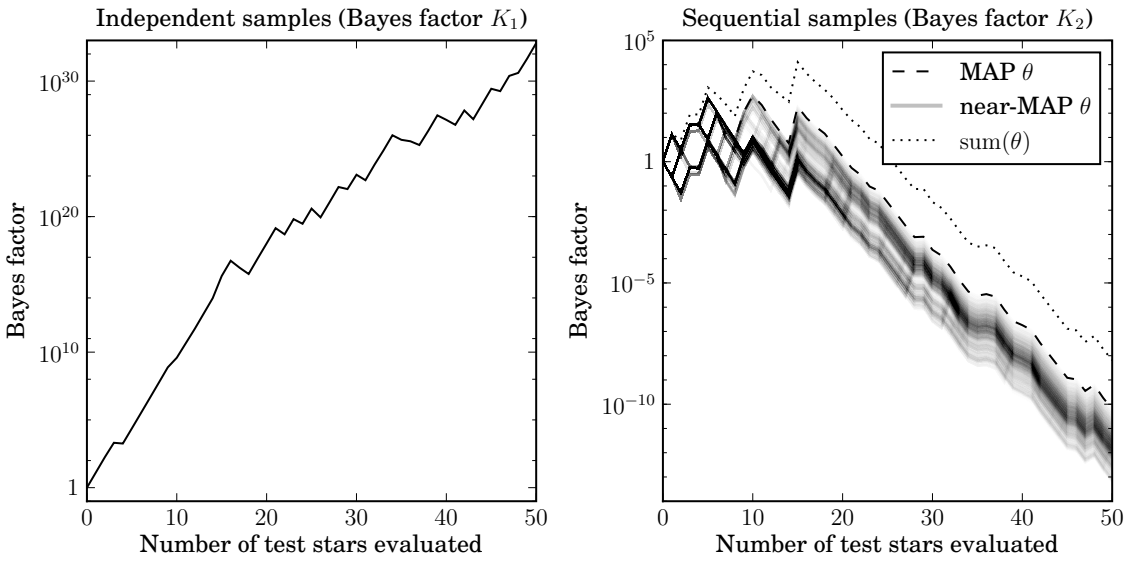


Figure 3.11: "Lucky donut" example, continued. **Left:** The Bayes factor in the simple independent setting, K_1, using the adaptive background model. Since each test star that is part of a donut is very close to a reference star, the Bayes factor rises quickly to a large value, though no stars other than the lucky donuts are aligned with the reference stars. **Right:** the Bayes factor in the sequential draw setting, K_2. The maximum *a posteriori* (MAP) setting of the parameter θ is shown, along with a collection of alternate settings of θ (with probabilities near that of the MAP) and their sum. We kept only those paths that were within a factor of about 1000 of the best path. Since only one of the stars in each "donut" is allowed to match to the reference star, the sequential setting correctly produces a tiny Bayes factor and rejects this false alignment. In this example, the sum of the probabilities given different values of the parameter θ is significantly larger than the MAP setting, because for each donut the model can choose any one of the test stars to be assigned to the reference star, and each of these assignments has approximately equal probability. In most other cases, the MAP comprises a much larger fraction of the sum.

donut draws many samples within a small region of the image, thus the Bayes factor contribution of each donut is effectively a single sample from the Bayes factor terrain raised to the power of the number of stars in the donut. Although any false alignment can have a "lucky" test star that happens to be near a reference star, this yields only a moderate increase in the Bayes factor, and we are unlikely to see many lucky stars in a single image. A lucky donut that appears near a reference star, on the other hand, can yield a very large increase in the Bayes factor, since it receives a moderate increase in Bayes factor taken to a large power. As a result, our simple model setting will produce false positives at a much higher rate than predicted.

3.37 We can eliminate this problem by requiring that each reference star match to at most one test star. Doing this requires reframing the probabilistic question we are asking. Instead of assessing the likelihood of each test star independently, we assess the likelihood of a single draw of the set of test stars. Equivalently, we can think of this as a sequential process: we evaluate the likelihood of a test star *given* the test stars that have already been seen, and with the constraint that multiple test stars cannot match a single reference star.

3.38 In this setting, we can parametrize the foreground model with a vector $\theta_{1:N_t}$ with one element per test star. Element θ_i contains either the identity of the reference star to which test star i corresponds, or indicates that the test star is a distractor. Given a parameter vector θ, this new foreground model, F_2, has probability:

$$p(t \mid \theta, F_2) = \prod_{i=1}^{N_t} \begin{cases} \frac{1}{A} & \text{if } \theta_i \text{ says } t_i \text{ is a distractor} \\ \mathcal{N}(t_i \mid r_{\theta_i}, \sigma_i^2) & \text{otherwise} \end{cases} \quad . \tag{3.33}$$

3.39 The Bayes factor in this setting, K_2, is

$$K_2 = \frac{\sum_\theta p(t \mid \theta, F_2) p(\theta \mid F_2)}{p(t \mid B)} \tag{3.34}$$

where we must ensure that the prior is normalized:

$$\sum_\theta p(\theta \mid F_2) = 1 \quad . \tag{3.35}$$

The constraint that each reference star be matched to at most one test star can be expressed by assigning zero probability to any setting of the parameter θ that has repeated elements ($\theta_i = \theta_j$ for $i \neq j$), ignoring distractors.

3.40 The simple foreground model, F_1, can be placed in this framework by setting the prior to

$$p(\theta \mid F_1) = \prod_{i=1}^{N_t} \begin{cases} d & \text{if } \theta_i \text{ says } t_i \text{ is a distractor} \\ 0 & \text{if } \theta_i = \theta_j \text{ for any } j \neq i \\ \dfrac{1-d}{N_r} & \text{otherwise} \end{cases}, \tag{3.36}$$

but the zeroed-out terms (the constraint that repeated elements are not permitted) causes the sum of this prior to be less than one. Knowing that repeated terms will be eliminated, we can reportion the prior mass:

$$p(\theta \mid F_2) = \prod_{i=1}^{N_t} \begin{cases} d + (1-d)\,\dfrac{\mu_i}{N_r} & \text{if } \theta_i \text{ says } t_i \text{ is a distractor} \\ 0 & \text{if } \theta_i = \theta_j \text{ for any } j \neq i \\ \dfrac{1-d}{N_r} & \text{otherwise} \end{cases} \tag{3.37}$$

where μ_i is defined to be the number of matches (non-distractors) preceding element i. This prior has the behavior that if many of the test stars are matched to reference stars, the distractor probability increases: the foreground model is not penalized strongly if the last few test stars are labelled as distractors.

3.41 Unfortunately, computing exactly the Bayes factor K_2 requires evaluating the foreground model over an exponential number of settings of the parameter θ. However, nearly all of these terms are negligible: if test star t_i is far from reference star r_{θ_i} then the Gaussian probability will be extremely small and any θ with that setting of θ_i will contribute negligibly to the total foreground model probability. Indeed, for many images we find that a handful of terms, or even the single maximum *a posteriori* (MAP) term, contributes nearly all the probability. See figure 3.11, for example.

3.4 The *Astrometry.net* case

3.42 The *Astrometry.net* system generates many hypothesized matches, then runs a verification test to reject false positives. The proposed matches are generated using a "geometric hashing" technique: the relative positions of a small number of test stars is converted into a hash code, and by searching for similar codes in a large index, we find sets of reference stars that may match the test stars. We call the small set of stars a "quad", because traditionally we used sets of four stars. The *Astrometry.net* system proposes that a quad of stars in the test image matches a quad of stars in the reference set, then looks up *other* stars in the index that we would expect to find in the image if the proposed match were correct. In the current implementation of the *Astrometry.net* system, we use the simple uniform background model, and the sequential foreground model of section 3.3.1.4. The current section discusses the modifications and extensions of these models that are required in the *Astrometry.net* setting.

3.43 The *Astrometry.net* system design goals include very low false positive rate, sensitivity when the test image contains few stars, robustness in the face of artifacts in the test image, and speed. The low false positive rate, combined with the fact that we usually evaluate many false hypotheses before encountering a true hypothesis, means that we require the Bayes factor to be very large. For example, if our utility function is:

	True Alignment	False Alignment
Accept	$u(\mathrm{TP}) = +1$	$u(\mathrm{FP}) = -2000$
Reject	$u(\mathrm{FN}) = -1$	$u(\mathrm{TN}) = +1$

and our prior (the expected number of false alignments we see before a true alignment) is $p(F)/p(B) = 10^{-6}$, then we demand that the Bayes factor be $\gtrsim 10^9$ in order to accept the proposal.

3.44 Fortunately, we typically find that true alignments yield very large Bayes factors, so we are willing to make some approximations in order to make the process faster, as long

as the approximations are "conservative": we strongly want to avoid overestimating the Bayes factor, but we are willing to underestimate it slightly.

3.45 In the *Astrometry.net* case, we are testing a proposed alignment *given* that a quad of reference and test stars have similar relative arrangements. Therefore, we should not be surprised that the reference and test stars comprising the quad are much closer to each other than expected by chance. We are not comparing the foreground model against a purely random background model, but rather the background model *given* that the quad matches. We take this into account by simply removing the reference and test stars that comprise the quad. This is an approximation—it causes the foreground model to ignore the degree to which the quad stars match—but correctly marginalizing the background model given that the quad matched is not trivial.

3.46 The fact that we generate hypotheses by aligning a reference quad with a test quad has another effect: positional errors in the stars comprising the quad cause the whole proposed alignment to be translated, rotated, scaled or sheared. This error in the alignment matrix affects all reference stars (if we think of applying the alignment matrix to project reference stars into the test image coordinate system). The rotation, scaling, and shear terms result in larger positional errors for stars far from the center of the matched quad. See figure 3.12 for an example.

3.47 There are two obvious ways of handling this effect. We could add parameters to the foreground model to correct for errors in the transformation matrix, and attempt to marginalize over these extra parameters. This approach has the advantage that proposed alignments suffering from transformation errors will be assigned only a slightly reduced Bayes factor compared to an alignment with no transformation error. In addition, the corrections to the proposed alignment that we infer can be propagated to later stages of processing. We would like to implement this strategy in future versions of *Astrometry.net*. The much simpler approach we use currently is to increase the effective positional variance of reference stars according to their distance from the quad center. We set the variance

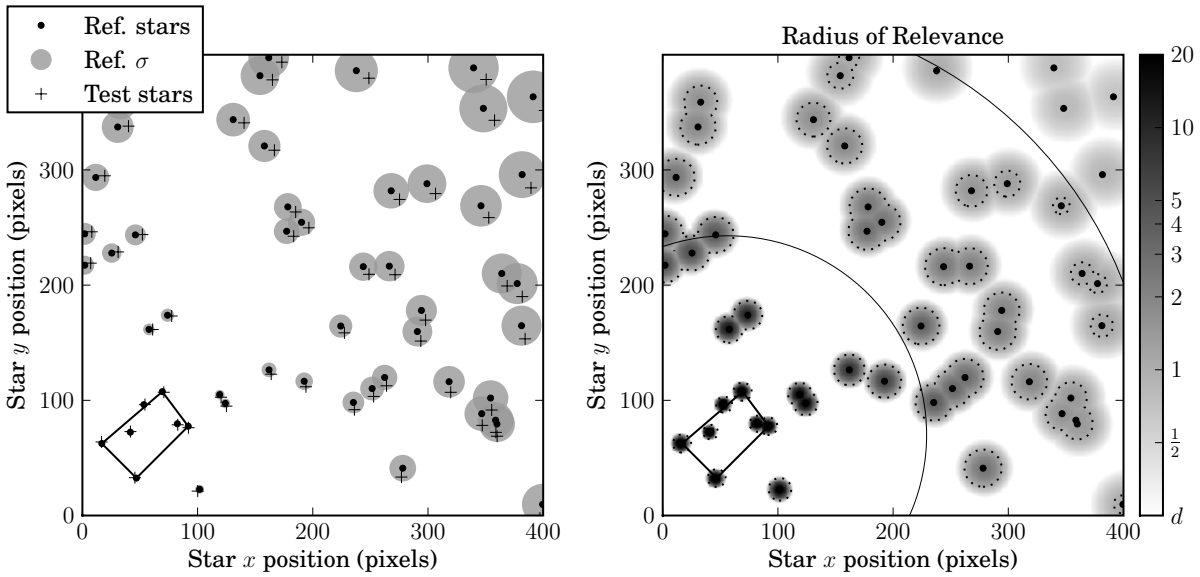


Figure 3.12: **Left:** The effect of errors in the matched quad on the rest of the stars. A small rotation of the test stars in a quad that matches a reference quad leads to errors in the predicted positions of reference stars in the image. The size of the errors is large for stars that are far from the center of the quad. We compensate for this effect by increasing the positional error estimate according to the distance from the quad. The disks show the size of the positional error σ that we assign to the reference stars (equation 3.38). **Right:** The Bayes factor terrain that results from increasing the positional variance for stars far from the quad. The outer circle shows where the peak of the Gaussian around each reference star is equal to the uniform background level. Outside this radius, the Bayes factor can never increase. The inner circle shows the "radius of relevance" (equation 3.40), where the expected Bayes-factor gain for stars drawn from the foreground model is zero. Outside this radius, test stars drawn from the foreground model will tend to result in a decreasing Bayes factor. We therefore select only stars within the radius of relevance.

for reference star r_j to σ_j^2 based on its distance from the quad center q and the effective radius of the quad, Q:

$$\sigma_j^2 = \sigma^2 \left(1 + \frac{(r_j - q)^2}{Q^2}\right) \quad . \tag{3.38}$$

This approach has the advantage of simplicity and speed, but results in underestimates of the Bayes factor for true alignments with transformation matrix errors, since in effect we are treating the larger-than-expected observed positional errors as independent improbable events.

3.48 The effective positional variance σ^2 grows for reference stars that are far from the quad center. As a result, the "sharpness" or "informativeness" of the foreground model—which we can express as the expected gain in Bayes factor given stars that are drawn from the true foreground model—also decreases for reference stars far from the quad center. Indeed, at some radius, the expected gain is zero! This is the radius at which the Gaussian peak around a reference star is broad enough that its expected value is equal to the background model. We call this distance the "radius of relevance", because reference and test stars outside this radius cannot, in this approximation, contribute to discriminating between the foreground and background models. The radius of relevance R can be found by setting the expected Bayes-factor gain (equation 3.30) to zero:

$$\langle g \rangle \approx d \log d + (1-d)\left(\log \frac{A(1-d)}{2\pi\sigma^2(R)N_r} - 1\right) = 0 \tag{3.39}$$

$$R = Q\sqrt{\frac{A(1-d)\exp\left(\frac{d\log d}{1-d} - 1\right)}{2\pi\sigma^2 N_r} - 1} \quad . \tag{3.40}$$

Before evaluating test stars, we drop any reference and test stars that lie outside the radius of relevance. This reduces both the effective area of the image and the number of reference stars N_r. In effect, we find a subimage in which the quad is informative enough to distinguish between the foreground and background models and consider only that subimage. See figure 3.12.

3.49 The reference stars in the *Astrometry.net* system are designed to have several properties that we must take into account in the verification test. First, they have been selected

to be spatially uniform and bright in a particular optical bandpass. The spatial uniformity is achieved by placing a HEALPix [26] grid over the sky and selecting the brightest stars in each grid cell. In addition, we attempt to eliminate spurious stars (false stars introduced by errors in the imaging or processing steps) by removing any star that is closer than some minimum distance to a brighter star. We call this "deduplication." In effect, deduplication and uniformization force the mean inter-star separation to lie within a reasonable range. In order for the reference stars to be a good model of the test stars, we must either adjust the foreground model to take into account the processing that has been applied to the reference list, or apply the same processing to the test star list. We take the latter approach: before applying the verification procedure, we uniformize the test stars at the same scale as the reference stars. We place a grid over the test stars and reorder them by selecting first the brightest star in each grid cell, then the second-brightest, and so on. We also perform deduplication of the test stars at the same scale as the reference stars. As a result of the uniformization and deduplication of reference stars, the simple uniform background model becomes reasonable again, since stars are not allowed to be close enough for the problem demonstrated in the globular cluster example (figure 3.3) to arise.

3.50 In order to make the uniformization of test stars faster, we use a rectangular grid rather than the HEALPix grid used for the reference stars. The grid size is chosen so that the cells are equal-area and have approximately the same area as the reference grid. Next, during radius of relevance filtering, we compute the distance to the grid cell centers rather than to each star individually. This also allows us to compute the effective area by simply counting the number of grid cells within the radius of relevance.

3.51 Another effect of the deduplication of reference and test stars is that it is rare to find multiple test stars near a reference star, or vice versa. As a result, it is rare for there to be more than one setting of the foreground model parameter vector θ to have significant probability. We have found that it is reasonable to approximate the sum over all θ values

by the single maximum *a posteriori* element.

3.5 Discussion

3.52 We have developed, and explored the subtleties of, a simple model for assessing the probability that a proposed match between two sets of stars is true, using the framework of Bayesian decision theory. This model is key to the success of the *Astrometry.net* system, which is built around a heuristic that generates hypotheses that need to be checked in a robust and efficient manner.

3.53 We have kept the model simple for several reasons. Since the *Astrometry.net* system runs the verification check a large number of times, we want it to be computationally inexpensive. But perhaps more importantly, the simplicity of the model and the few assumptions it makes about the data make it quite robust to the very wide variety of images we have encountered during several years of operating the *Astrometry.net* web service.

3.54 One problem in the *Astrometry.net* context that we feel would be remedied by a more flexible model is that small positional errors in the matched quad are multiplied for stars that are far from the matched quad. While running the verification procedure we should be able to detect when the errors in the test star positions are predominantly in directions that could be explained by small changes to the stars that form the matched quad. We could then suggest a correction to the hypothesis. This could allow fewer correct hypotheses to be rejected at relatively little computational cost.

3.55 A related idea is to update the hypothesis based on all the stars that are labelled as matches by the verification procedure. This leads naturally to an iterative routine similar to the Expectation-Maximization algorithm [18] for mixture models: we would alternate steps of optimizing the foreground model parameters (the alignment hypothesis) based on the stars "assigned" to the foreground model, and re-assigning stars to the foreground

and background models based on the new model parameters. The resulting hypothesis would be a (locally) maximum-likelihood solution. This EM-like solution is just one of several optimization algorithms that could be applied to improve the hypothesized match.

3.56 We have argued that uniformization and deduplication make the uniform background model sufficient, but using the adaptive background model could still be beneficial. Since the adaptive model is more flexible, using it will make us more conservative: hypotheses that are only marginally accepted when using the uniform background model may be rejected when using the more powerful adaptive model. One way of alleviating the higher computational cost of the adaptive background model would be to use the uniform background model to pre-filter hypotheses, then use the adaptive background model on hypotheses that have sufficiently high Bayes-factor thresholds.

3.57 We have assumed that the fraction of "distractor" stars is known. In the experiments here we simply fix it at $d = 0.25$. The model is not particularly sensitive to this parameter, but in some cases it could be benificial to optimize or marginalize over its value. For example, about 20% of the photographic plates in the Harvard College Observatory archives contain multiple exposures [27]. Since each hypothesized match pertains to only one exposure, the stars from all other exposures will appear as distractors. For such images, it would be benificial to allow the foreground model to adjust the distractor fraction.

Chapter 4

Efficient implementation of the KD-tree data structure[1]

[1]This chapter was originally prepared as a manuscript by me, Keir Mierle, and Aeron Buchanan.

4.1 Introduction

4.1 The kd-tree is a simple yet versatile data structure for speeding up searches in k-dimensional data sets. It applies to nearest-neighbour queries, which arise in many computer vision applications such as vector quantization and template matching. It also applies to a family of related problems such as range search (finding all points within a given radius of a query), K-nearest neighbours, and approximate kernel density estimation. Over its long history the kd-tree has accumulated many extensions and variations making the literature difficult to comprehend. This chapter presents a modern view of the kd-tree. It also points out a number of optimizations that can increase the performance by an order of magnitude over existing publicly available implementations while also reducing the memory requirements—or increasing the size of the largest kd-tree that be constructed in a given amount of memory.

4.2 The standard kd-tree

4.2 Kd-trees have accelerated spatial queries for over thirty years. Kd-trees speed up N-body simulations, search, object recognition, ray tracing, template matching, kernel density estimation, and more.

4.3 Imagine you have a classification problem. You have a million labelled 2-dimensional data points ("feature vectors") in the unit square. You also have a set of a million unlabelled data points, and a deadline. You decide to run nearest-neighbour classification. After a moment of thought you realize that if you compare every labelled feature to every unlabelled feature you will have to do 10^{12} comparisons. Your deadline looms. Upon further reflection, you decide to split the labelled points in half: the ones with $x \leq \frac{1}{2}$ and the ones with $x > \frac{1}{2}$. When you want to find the nearest neighbour of an unlabelled point that has $x \leq \frac{1}{2}$, you look in the first set of labelled points. Most of the time, you find that the distance between the unlabelled point and its nearest labelled neighbour is

less than the distance to the $x = \frac{1}{2}$ boundary. You don't even need to look at the second set of labelled points!

4.4 While waiting for your algorithm to finish, you remember recursion. You decide to split the labelled points in half again. And again. And again. And again. You just invented kd-trees. If it was 1975 you would be the first, and win a prize [9].

4.5 You used a kd-tree to speed up nearest neighbour search (find the closest data point to the query point). They are also useful for *range search* (find all data points within some radius of a query point) and more complex queries such as *approximate kernel density estimation*.

4.6 A kd-tree is a type of *Binary Space Partition* tree. Each node in the tree represents a portion of space. In kd-trees the portion of space is an axis-aligned bounding box.[2] Each node has two children which split the node's space in two. This split is usually chosen so that half the data points go into each child. A node's axis-aligned bounding box is either explicitly stored, or implicitly defined by its ancestors' splitting planes.

4.7 Kd-trees work well in low-dimensional spaces (opinions vary, but say up to 15). In high dimensions, other data structures may work better [13].

4.8 Below, we first describe the core structure of kd-trees, but note that the foundational implementation is not the most efficient. We then move to the comprehensive series of implementation "tricks" that must be considered for optimal performance. Finally, we compare our implementation, `libkd` (incorporating the efficiencies described), with two previously released kd-tree implementations: `ANN` [4] and `Simple kd-trees` [54] (abbreviated to `simkd`) to show the considerable level of speed-ups that can be attained.

[2] In the example above, the first node you created had the bounding box $x \in [0, \frac{1}{2}]$ and $y \in [0, 1]$.

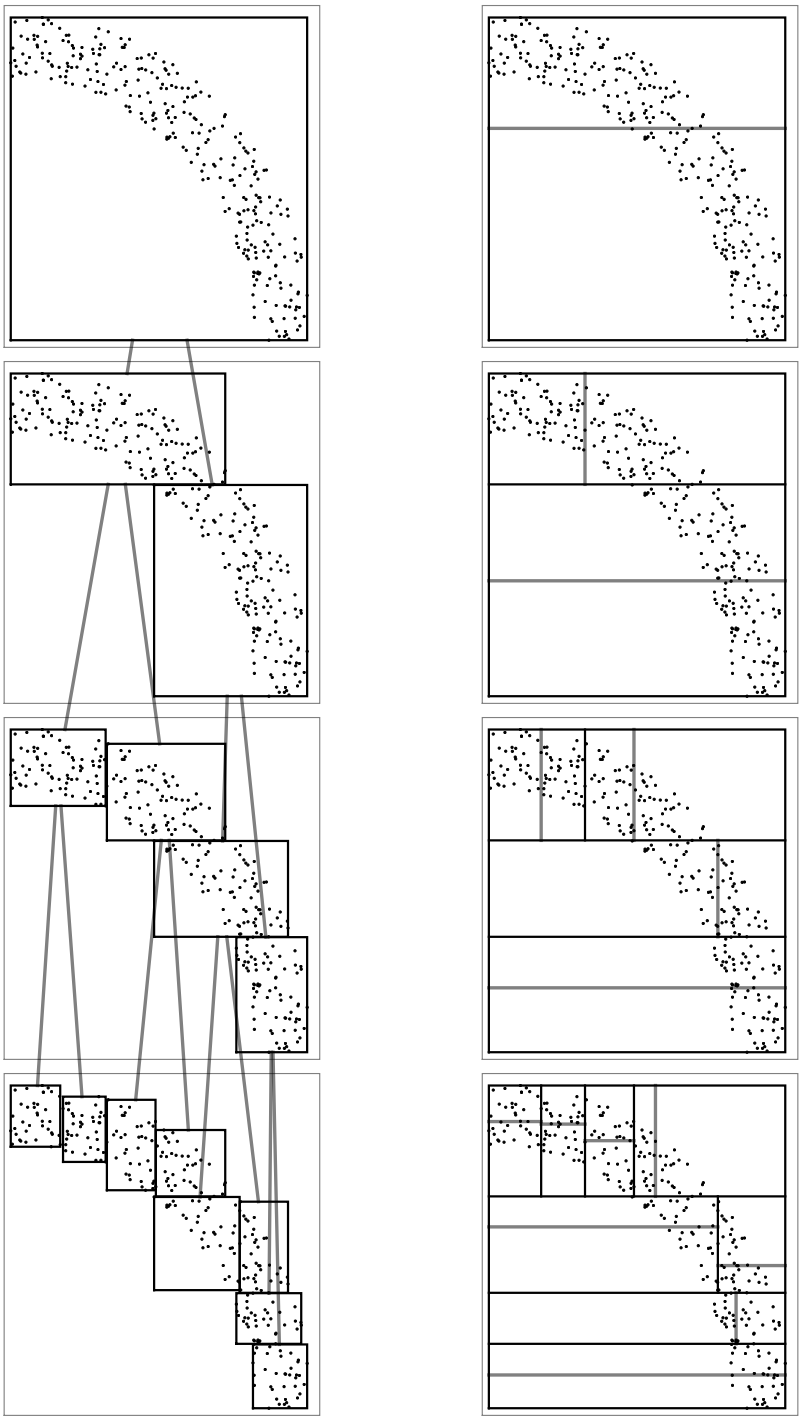


Figure 4.1: Two varieties of kd-tree. **Left:** A kd-tree with bounding boxes. Parent-child relationships are shown by gray lines. **Right:** A kd-tree with splitting planes. The splitting plane at each node is shown by a gray line. The top box is the root node. Parent-child relationships are not shown in order to avoid cluttering the figure.

4.3 Kd-tree implementation

4.3.1 Data structures

4.9 Figure 4.2 shows pseudocode for kd-tree data structures. There are two types of kd-tree nodes listed, `SplittingNode` and `BoundingBoxNode`. In this chapter, a kd-tree contains nodes of only one type, but of course it is possible to store both the splitting plane and bounding box of each node if required by the application. `BoundingBoxNode`s contain two points which define the maximum and minimum corners of the bounding box. `SplittingNode`s do not contain an explicit representation of their spatial extent. Instead, each node contains a split dimension and position. A `SplittingNode`'s bounding box is implicitly described by the splits of its ancestors.

4.10 The two representations have different performance characteristics on different data distributions. For example, the benchmark in section 4.5 favours splitting planes. Structured data sets where the dimensions are correlated and the data point distribution is clumpy work well with bounding box nodes.

4.11 Please note that these data structures are only for explaining the kd-tree; for practical implementation details see section 4.4.

4.3.2 Construction

4.12 Kd-tree construction starts with a set of points. Construction begins by creating a root node which contains all of the data points. The root node is split into two halves by choosing a splitting dimension (usually the dimension with the largest spread) and a splitting position (usually the median value of the points along that dimension). Then, the set of points whose position along the splitting dimension is less than the splitting position are put in the left child. The remaining points are put in the right child. Construction proceeds recursively until the nodes at the lowest level of the tree contain some small number of points (perhaps 10). See figure 4.3 for pseudocode.

```
class KDTree:
   Node root

class Node:
   # empty superclass for leaf and internal nodes

class InternalNode extends Node:
   # superclass of bounding box
   # and splitting plane nodes
   Node left_child
   Node right_child

class BoundingBoxNode extends InternalNode:
   # lower and upper corners of the bounding box
   point lower
   point upper

class SplittingNode extends InternalNode:
   # splitting plane dimension and value
   int split_dimension
   real split_position

class LeafNode extends Node:
   point[] points_owned
```

Figure 4.2: Data structures for kd-tree nodes. Each `LeafNode` contains a set of D-dimensional `point`s. All nodes in a kd-tree are either of type `BoundingBoxNode` or `SplittingNode`. In practice, more efficient structures should be used; see section 4.4.

4.3.3 Distance bounds

4.13 Fundamentally, kd-tree algorithms are about pruning nodes that don't need to be examined. The primary tool to achieve this is the `mindist` function, which provides a lower bound on the distance between a point and any point that lives in the region owned by a node. Intuitively, this is the distance from a point to the closest corner or closest edge of the node.

4.14 The form of the `mindist` function depends on the representation of a kd-tree node. For kd-trees that have bounding boxes, one computes the `mindist` to an individual node. For kd-trees that only store the splitting plane, it makes more sense to compute the `mindist` to each of the node's two children. An illustration is given in figure 4.4 and pseudocode in figure 4.5.

```
def build_kdtree(point[] p, int nleaf, bool b_boxes):
   t = new KDTree()
   t.root = build_node(p, nleaf, b_boxes)
   return t

# Build a kd-tree node from a set of points.
def build_node(point[] p, int nleaf, bool boxes):
   # if the set of points is small enough...
   if p.size() <= nleaf:
      # create a leaf node.
      return new LeafNode(p)
   # else, choose a splitting dimension...
   split_dim = choose_split_dimension(p)
   # and split the points along that dimension.
   (p_left, p_right, split_pos) = partition(p, split_dim)
   if boxes:
      n = new BoundingBoxNode()
      n.lower = min(p)
      n.upper = max(p)
   else:
      n = new SplittingNode()
      n.split_dimension = split_dim
      n.split_position = split_pos
   # recurse on the two point sets
   # to create the child nodes.
   n.left_child  = build_node(p_left,  nleaf, boxes)
   n.right_child = build_node(p_right, nleaf, boxes)
   return n

# Typical:  split the dimension with largest range.
def choose_split_dimension(points):
   return argmax(max(points) - min(points))

# Typical:  split at the median value.
def partition(points, dim):
   # find the median value in the splitting dimension
   split_val = median(points, dim)
   # grab points on each side of the partition.
   p_left  = [p in points where p[dim] <= split_val]
   p_right = [p in points where p[dim] >  split_val]
   return (p_left, p_right, split_val)
```

Figure 4.3: Pseudocode for kd-tree construction.

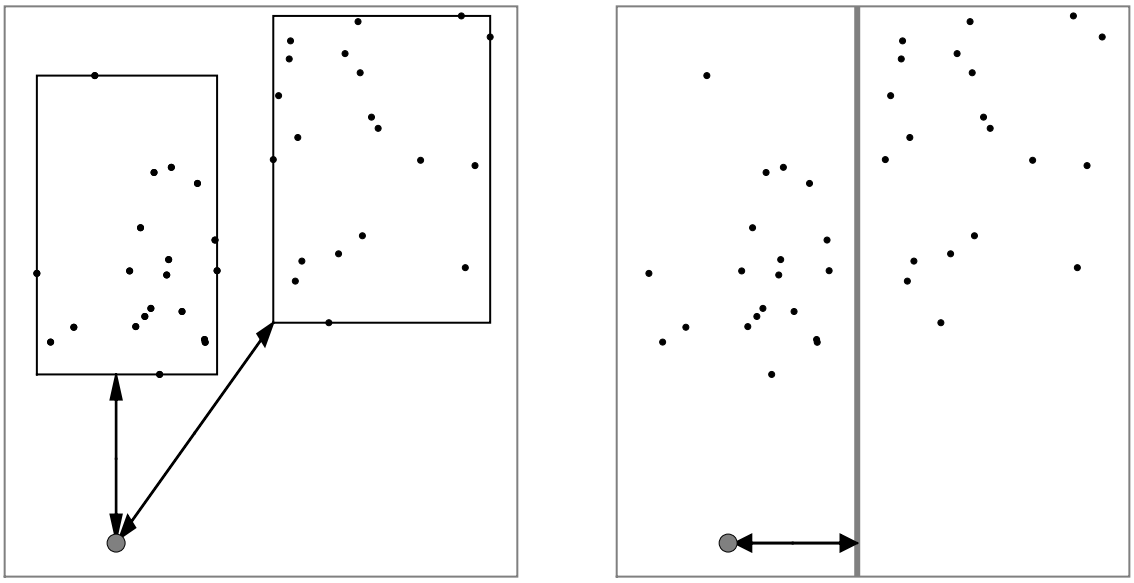


Figure 4.4: **Left:** `mindist` for a kd-tree node with bounding boxes. The query point is shown as a gray dot, and the `mindist` s to the two boundig-boxes are shown by the arrows. **Right:** `mindist` for a kd-tree node with only the splitting dimension and value. The `mindist` is zero for the child on the same side of the splitting plane as the query point; for the other child the `mindist` is simply the distance in the splitting dimension from the query to the splitting plane.

```
def mindist_to_box(point p, BoundingBoxNode n):
   mindist = 0
   for each dimension d:
      if p[d] > n.upper[d]:
         # p is above the bounding
         # box in this dimension
         diff = n.upper[d] - p[d]
      else if p[d] < n.lower[d]:
         # p is below the bounding
         # box in this dimension
         diff = p[d] - n.lower[d]
      else:
         # p is within the bounding
         # box in this dimension
         diff = 0
   mindist += diff * diff
   return sqrt(mindist)

def mindist_to_left_child(point p, SplittingNode n):
   d = n.split_dimension
   if p[d] > n.split_position:
      # point is on the right side
      # of node's splitting plane
      return p[d] - n.split_position
   return 0

def mindist_to_right_child(point p, SplittingNode n):
   d = n.split_dimension
   if p[d] < n.split_position:
      # point is on the left side
      # of node's splitting plane.
      return n.split_position - p[d]
   return 0
```

Figure 4.5: Pseudocode for `mindist`. **Top:** for kd-trees with bounding boxes. **Bottom:** for kd-trees with splitting planes.

4.3.4 Nearest neighbour

4.15 Given a set of data points and a query point, solving the nearest neighbour problem consists of finding the data point that is closest to the query. Specifically, given query point $\mathbf{q}$, find

$$\mathbf{x}_{NN} = \underset{i}{\operatorname{argmin}} \|\mathbf{x}_i - \mathbf{q}\|_2 \quad . \tag{4.1}$$

The algorithm is of the *branch and bound* variety [43]: the tree is traversed, maintaining a *pruning distance*—the distance to the nearest data point found so far. In addition, the set of nodes that might contain the solution are stored in a stack or priority queue ordered by `mindist`. If a node is encountered whose `mindist` to the query point is larger than the pruning distance, it is discarded because it cannot contain the solution. When a leaf node is encountered, its data points are examined individually. If any data point is closer than the pruning distance, that point becomes the nearest neighbour found so far, and the pruning distance is set to the point's distance to the query point. The algorithm begins by setting the pruning distance to $+\infty$ and placing the root node on the stack. See the pseudocode in figure 4.6.

4.3.5 Rangesearch

4.16 In the rangesearch problem, the goal is to find all points that are within a certain radius of the query point. Specifically, given query point $\mathbf{q}$, radius r, and a set of points $\mathbb{X}$, find the set of points

$$\left\{ \mathbf{x} \;\middle|\; \|\mathbf{x} - \mathbf{q}\|_2 \leq r,\ \mathbf{x} \in \mathbb{X} \right\} \quad . \tag{4.2}$$

Range search is simpler than nearest neighbour search because the pruning distance is fixed. This implies that the list of nodes that must be inspected does not depend on the order of tree traversal. At each node the `mindist` to the query point is computed. If it is larger than the search radius, it is pruned because it cannot possibly contain any data points within the radius. See the pseudocode in figure 4.7.

```
# Input:  root of a kd-tree, and a query point q
# Output:  (x, distance(x, q)) such that x is the
#  nearest neighbour of q in the kd-tree
def nearest_neighbour(KDTree t, point q):
   nodestack.push(t.root)
   closest_so_far = +inf
   nearest_neighbour = null
   while nodestack is not empty:
      n = nodestack.pop()
      if mindist(n, q) >= closest_so_far:
         # this node can be pruned.
         continue
      if n.is_leaf():
         # leaf node:  examine individual points
         for p in n.points_owned:
            d = distance(p, q)
            if d < closest_so_far:
               # nearest neighbour found so far
               closest_so_far = d
               nearest_neighbour = p
      else:
         # find the mindists to the child nodes
         mindist_L = mindist_to_left_child(n, q)
         mindist_R = mindist_to_right_child(n, q)
         # visit the closest child first
         # (by stacking the farther child first)
         if mindist_L > mindist_R:
            nodestack.push(n.left_child)
            nodestack.push(n.right_child)
         else:
            nodestack.push(n.right_child)
            nodestack.push(n.left_child)
   return (nearest_neighbour, closest_so_far)
```

Figure 4.6: Pseudocode for nearest neighbour search. The algorithm maintains a stack of nodes to explore and a pruning distance (`closest_so_far`), which is the distance to the nearest data point found so far. Any node whose `mindist` is greater than the pruning distance is pruned.

```
# Input:  A kd-tree, a query point q, and a range
# Output:  The set [(x, distance(x, q)), ...]
#  such that distance(x, q) < range
#  for all x in the kd-tree
def range_search(KDTree t, point q, real range):
   results = [] # empty list
   range_search_helper(t.root, q, range, results)
   return results

def range_search_helper(Node n, point q,
                        real range, list res):
   if is_leaf(n):
      for each p in n.points_owned:
         d = distance(p,q)
         if d < range:
            res.append( (p, d) )
      return
   if mindist_to_left_child(n, q) < range:
      range_search_helper(n.left_child,  q, range, res)
   if mindist_to_right_child(n, q) < range:
      range_search_helper(n.right_child, q, range, res)
```

Figure 4.7: Pseudocode for range search.

4.3.6 Approximations

4.17 Kd-trees are applicable to approximate as well as exact algorithms. A prominent example is the *Best Bin First* (BBF) algorithm used by many SIFT-based vision systems [8]. BBF is essentially kd-tree nearest-neighbour search where the tree traversal is ordered by a priority queue. By terminating the search after exploring some fixed number of leaf nodes, an approximate nearest neighbour is produced quickly.

4.18 There is an elegant solution to approximate kernel density estimation which looks similar to the nearest-neighbour algorithm. It refines upper and lower bounds on the kernel density with a series of `mindist` and `maxdist` calculations. The bounds start loose and tighten as the tree is traversed. The algorithm terminates when the interval is small enough.

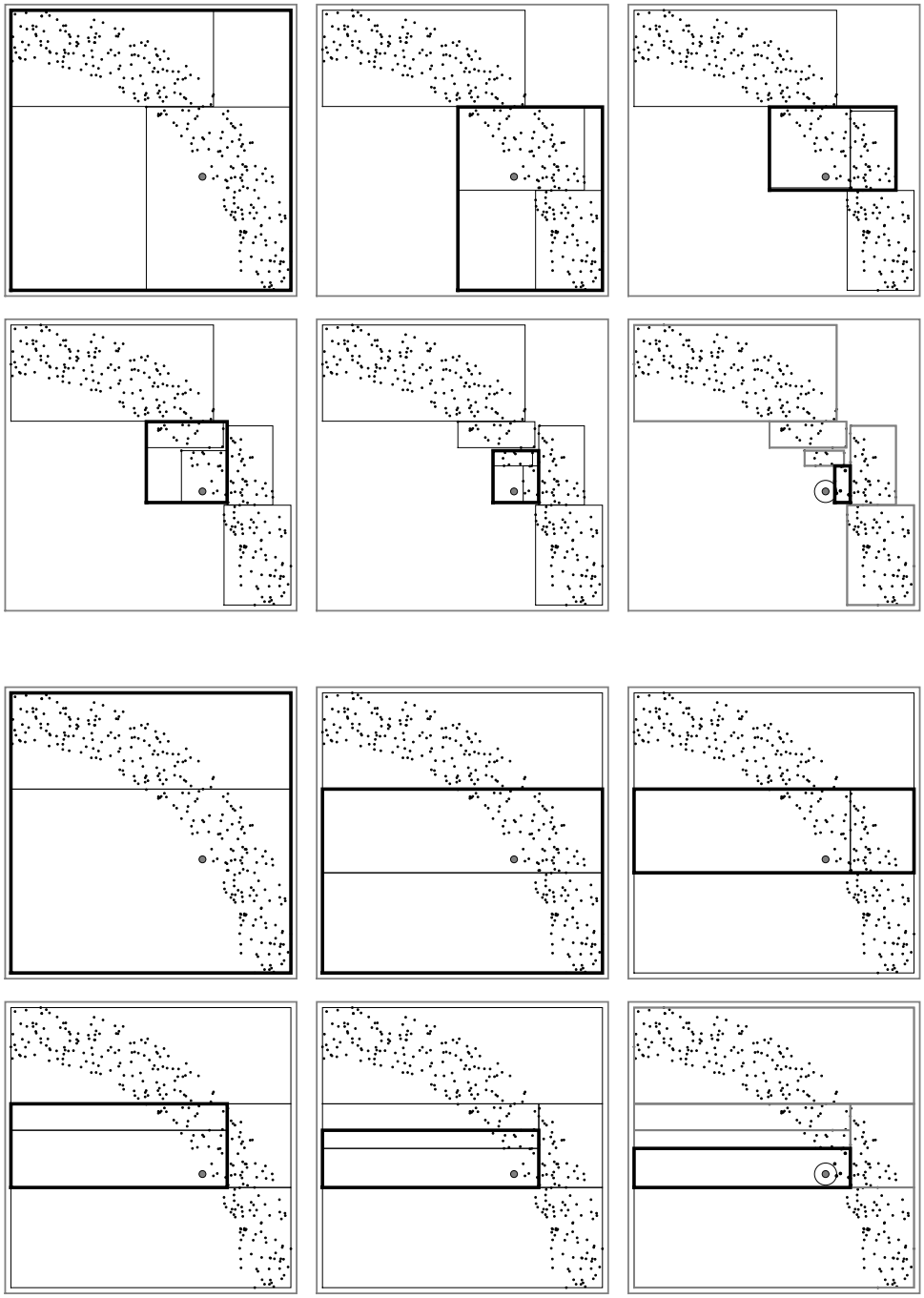


Figure 4.8: Nearest-neighbour search. **Top:** With bounding boxes. **Bottom:** With splitting planes. The node currently being examined is shown in bold. Nodes that are queued to be examined are shown with thin black borders. Pruned nodes are shown with thin gray borders. In this example, the first leaf node explored contains the nearest neighbour which is close enough that all the queued nodes can be pruned.

4.4 Efficient Implementation Tricks

4.19 We consider the common case in which the kd-tree is built from a static data set and then queried millions of times. Although there are some applications in which the data set must be updated online, a number of opportunities arise when points are not added or removed after the kd-tree is built.

4.20 This section presents a series of optimization "tricks" taken from the battle-hardened kd-tree implementation at the heart of *Astrometry.net*. These tricks yield an order of magnitude speed increase and vastly reduce the memory required compared to the alternatives available.

4.21 In this section, the dimension of the data points is D, the number of data points is N, and the number of nodes in the kd-tree is M. All trees are binary and complete. N is large relative to D; for example in the authors' application D is 3 or 4, and N is 10 to 100 million. For us, the maximum tree size that fits in a 32-bit address space is an important design parameter.

4.22 Not all of these implementation "tricks" apply in all cases, but each one helps. Although most of the tricks are concerned with reducing the memory requirements of the kd-tree, this often has the effect of increasing the speed as well.

4.4.1 Store the data points as a flat array.

4.23 Use a one-dimensional array of size ND, where coordinate j of data point i is stored at location $iD + j$. For example, in three dimensions the array will contain

$$(x_0, y_0, z_0, x_1, y_1, z_1, \ldots).$$

4.24 It may be tempting to store the data points as an $N \times D$ two-dimensional array. This incurs a memory overhead of N pointers, and at runtime requires a pointer dereference. To store one million points in 3 dimensions ($N = 10^6, D = 3$), our `libkd` requires

an overhead of 4 bytes, while the existing implementations `ANN` and `simkd` use over 4 megabytes (8 megabytes on 64-bit) and over 20 megabytes (40 on 64-bit), respectively.

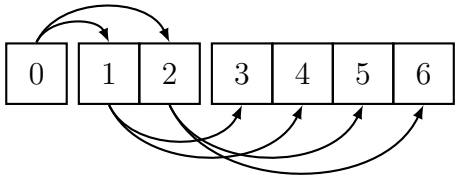
4.4.2 Create a complete tree.

4.25 Instead of splitting nodes until each leaf node contains some maximum number of points, fix the number of tree levels and create a complete tree. The huge advantage of this trick is that the number of nodes is known in advance, which allows the next trick to be used without wasting any memory.

4.26 There are disadvantages to this trick. First, the number of data points in the leaf nodes is only adjustable by factors of two. Second, if nodes are not split at the median value, then some leaf nodes will contain more data points than others.

4.4.3 Don't use pointers to connect nodes.

4.27 Instead, put the nodes in a single array. Use the heap indexing strategy shown below instead of explicit pointers. The root is node 0. The left child of node i is node $2i + 1$, and the right child is node $2i + 2$.



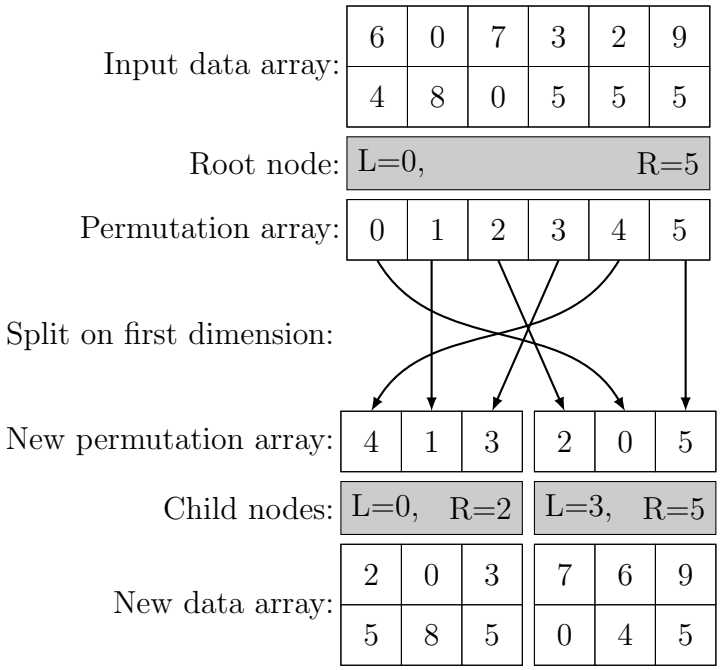
4.28 For a tree with M nodes, this saves about M pointers, and places sibling nodes—which are likely to be accessed at the same time—next to each other in memory. This locality of reference improves cache performance.

4.29 A convenient side-effect of using this trick is that the kd-tree is position-independent: the array of nodes can be moved in memory without changing any of the contents. Indeed, it can be written directly to disk in 'live' form and restored at a later date. By using the

`mmap()` system call, the live on-disk representation is immediately available for queries; the time required to read it from disk will be amortized over the subsequent queries. Regions of space that are never queried may never be read from disk.

4.4.4 Pivot the data points while building the tree.

4.30 When building the tree, pivot the data along the splitting dimension so that the data points owned by the child nodes are contiguous in memory. Represent the set of data points owned by a node as leftmost and rightmost offsets into the data array (L and R). If necessary, keep a permutation array to map from the final array indices back to the original indices.



4.31 Once this has been done, the data points owned by leaf nodes are contiguous in memory. In addition, leaf nodes that are nearby in space are likely to have their data points stored nearby in memory. This increases locality of reference and therefore performance.

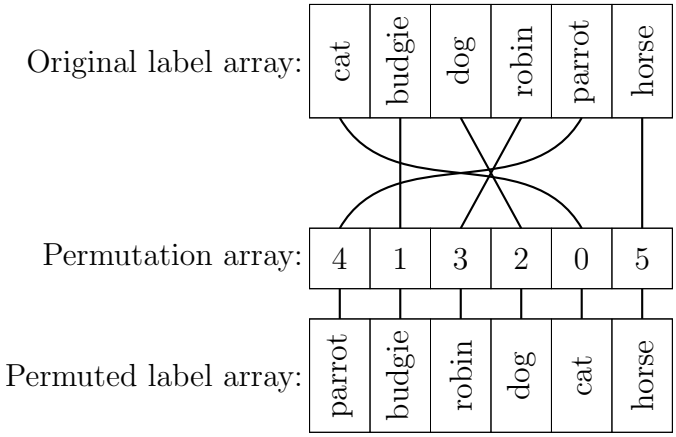
4.4.5 Don't use `C++` virtual functions.

4.32 It may be tempting to define a class hierarchy with an abstract base class `Node` and

classes `InternalNode` and `LeafNode` which inherit from it (as in section 4.3.1). Compilers implement this by adding a `vtable` pointer to each `InternalNode` and `LeafNode` object. This enlarges each node by the size of a pointer. This wastes 32 (or 64) bits of memory per node, since the node type is completely determined by its position in the tree. Using the node layout shown above, node i is a leaf if $i \geq M/2-1$ (where M is the total number of nodes). Using inheritance also incurs an indirect function call for each virtual function, which itself costs time and prevents the compiler from performing inlining optimizations.

4.4.6 Consider discarding the permutation array.

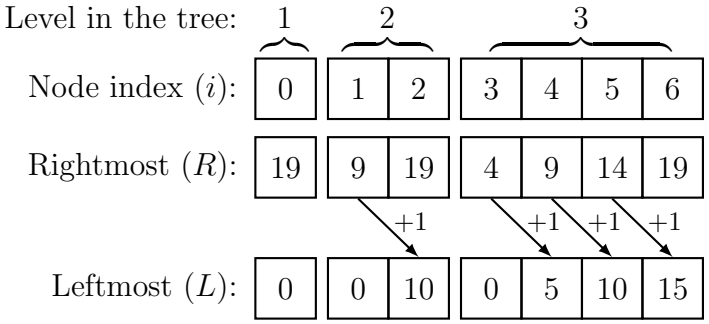
4.33 Apply the permutation array to auxiliary data to make the kd-tree's data order the canonical order. For example, you might have a set of data points where each point has a class label such as "cat" or "dog". After creating a kd-tree from the data points, the points are permuted so that the ith data point no longer corresponds to label i. However, if you apply the inverse of the kd-tree's permutation array to the labels, they will again correspond. You can then discard the kd-tree's permutation array. This eliminates a level of indirection and a significant amount of memory (N integers).



4.4.7 Store only the rightmost offset of points owned by a node.

4.34 The L and R offsets describe the range of data points owned by a node. Within a

level of the kd-tree, the L offset of a node is simply one greater than the R offset of the node just to the left, or zero if there is no node to the left. Since the L value can be trivially computed from the R value, there is no need to store both. This saves M integers.



4.4.8 Don't store the R offsets of internal nodes.

4.35 Only leaf nodes need to store the R offsets. The leftmost data point L of a non-leaf node is just the L value of its leftmost leaf. Similarly, the rightmost data point R of a non-leaf node is R of its rightmost leaf. Try saying that ten times fast! This saves $M/2$ integers.

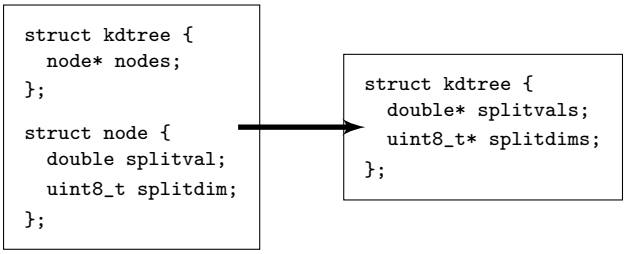
4.4.9 With median splits, don't store the R offsets.

4.36 Compute them instead. If the kd-tree is built by splitting at the median value (the common case), then the number of data points owned by each node in the tree is a function of the total number of data points, independent of the values of the data points. Computing the offsets costs $\mathcal{O}(\log M)$[3] if exact median splits are used and a fixed rounding direction is chosen (*i.e.*, the right subtree gets the extra data point when the number of data points is odd). However, by choosing the rounding direction carefully—by moving the splitting position by one place—the data points can be distributed so that the computation is $\mathcal{O}(1)$.

[3]There *may* be an $\mathcal{O}(1)$ algorithm, though the authors haven't found it.

4.4.10 Consider transposing the data structures.

4.37 Instead of creating an array of `node` structures, pull the node contents directly into the `kdtree` structure, creating a separate array for each element.



4.38 When a compiler encounters a structure such as `node` above, it pads the structure so that its total size is a multiple of the size of the largest element. For `node`, the structure will be padded to 16 bytes, even though only 9 bytes are used. It is possible to force the compiler to pack the structure tightly, but this results in unaligned memory accesses (many of the `double` values will not start on a multiple of eight bytes), which incurs significant overhead.

4.39 The advantage of transposing the data structures is that the memory required can be minimized without destroying the structure alignment. The disadvantage is a loss of locality of reference: typically the members of a structure are accessed at the same time, and transposing them results in the members being dispersed in memory.

4.4.11 Consider using a smaller data type.

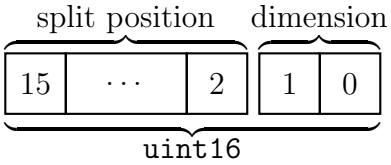
4.40 For most applications, using `double` values to represent points in $[0, 1]$ is not an effective use of bits. Consider transforming the data to a smaller integer representation. If the data points live in a small bounded space, then converting `double` values to 32-bit integers saves a factor of two of space with very little loss of precision. Converting to 16- or 8-bit integers saves more memory but the effect of introducing this approximation should be considered in the context of the problem at hand.

4.41 The boundary between the kd-tree software library and the application is an ideal

place to do this transformation: the application need not know that the kd-tree library represents the data points in a smaller format. The kd-tree library converts query values into the smaller format on the way into the library, and converts results back to the original format on the way out. The only user-visible change should be that the data points occupy much less memory, queries are faster, and there may be small approximation errors.

4.4.12 Consider bit-packing the splitting value and dimension.

4.42 Since kd-trees are only suitable for low-dimensional data, the splitting dimension only requires a few bits. Instead of storing it in a separate array, store it in the bottom few bits of the splitting value. For example, with four-dimensional data points, the splitting dimension can be stored in the bottom two bits of a 16-bit integer, leaving 14 bits to store the splitting position.



4.43 In general, this trick conflicts with trick of not storing the R offsets (section 4.4.9), since that trick requires precise control over the splitting plane location, while this trick involves sacrificing some precision to save space. For some data point sets, this conflict may not arise, and both tricks may be used simultaneously.

4.4.13 Consider throwing away the data.

4.44 If your data points are associated with some other information (such as labels), and your application can tolerate some false positive results, consider the tradeoffs of discarding the data and keeping the kd-tree. By using the other tricks in this section, the memory requirements of a kd-tree can be reduced to a small fraction of the size of the

data points themselves. In some applications, the benefit of being able to search more data points might outweigh the cost of not being able to say exactly where the data points were.

4.45 Given N data points, we can build a kd-tree with $N-1$ total splitting plane nodes. Each node contains a splitting value and possibly the splitting dimension: it is at most slightly larger than a single data value. The entire tree requires about $1/D$ as much memory as the data points themselves.

4.46 The kd-tree search algorithms then produce bounded approximations. For nearest-neighbour search, the algorithm can produce the index of the data point that lives in the same box as the query point, an upper bound on the distance to that point, and a lower bound to the distance to the true nearest neighbour. It is also possible to produce a small list of points that is guaranteed to contain the nearest neighbour.

4.5 Speed Comparison

4.47 This section gives empirical evidence that the tricks presented above can yield significant improvements. We compare our implementation, `libkd`, with two previously released kd-tree implementations: `ANN` [4] and `simkd` (`Simple kd-trees`) [54]. We used these packages 'out of the box', using the default compiler settings and choices about how to build the kd-tree.

4.48 We chose a very simple benchmark test: the data points are 5 million samples drawn uniformly from the unit cube ($N = 5 \times 10^6, D = 3$). The number of points owned by a leaf node was set to 14 for `ANN` and `simkd` in order to make the number of nodes in the three implementations approximately equal. The query points are one million samples from the same distribution. We tested the one-nearest neighbour algorithm. We would have preferred to use larger N, since this is the regime in which our *Astrometry.net* project operates, but the other libraries were not able to build such large trees. We

Implementation	Speed (k q/sec)		Memory data + tree = total (Mbytes)	
	32-bit	64-bit	32-bit	64-bit
`simkd`	47	39	120 + 250 = 370	120 + 366 = 486
`ANN`	71	90	120 + 67 = 187	120 + 101 = 221
`libkd-d-box`	127	144	120 + 52 = 172	
`libkd-d-split`	231	284	120 + 6 = 126	
`libkd-d-noR`	239	293	120 + 5 = 125	
`libkd-i-split`	242	311	60 + 4 = 64	
`libkd-i-noR`	240	326	60 + 3 = 63	
`libkd-s-split`	328	386	30 + 3 = 33	
`libkd-s-noR`	307	396	30 + 2 = 32	

Table 4.1: Benchmark results. Speed is measured in thousands of queries per second (k q/sec), memory in megabytes. The memory values include the memory required to store the data points themselves, which is 120 MB when `double`s are used. For the `libkd` entries, `d` indicates that the data points are stored as `double`, `i` indicates 32-bit integers, and `s` indicates 16-bit integers (using trick 4.4.11). `box` means that bounding-boxes are created; otherwise splitting-planes are used. The `noR` entries indicate that the we avoid storing the offset arrays (trick 4.4.9). We have also assumed that trick 4.4.6 is applicable, so the permutation arrays are not stored.

present results for a 32-bit machine[4] and a 64-bit machine[5] to show how the memory requirements differ.

4.49 Table 4.1 and figure 4.9 show our results. Our implementation, `libkd`, always takes less memory and produces faster results. The memory requirements of `ANN` and `simkd`

[4]Intel Xeon (SL72G) at 3.06 GHz with 4 GB of RAM.
[5]AMD Opteron 8220 SE at 2.8 GHz with 32 GB of RAM.

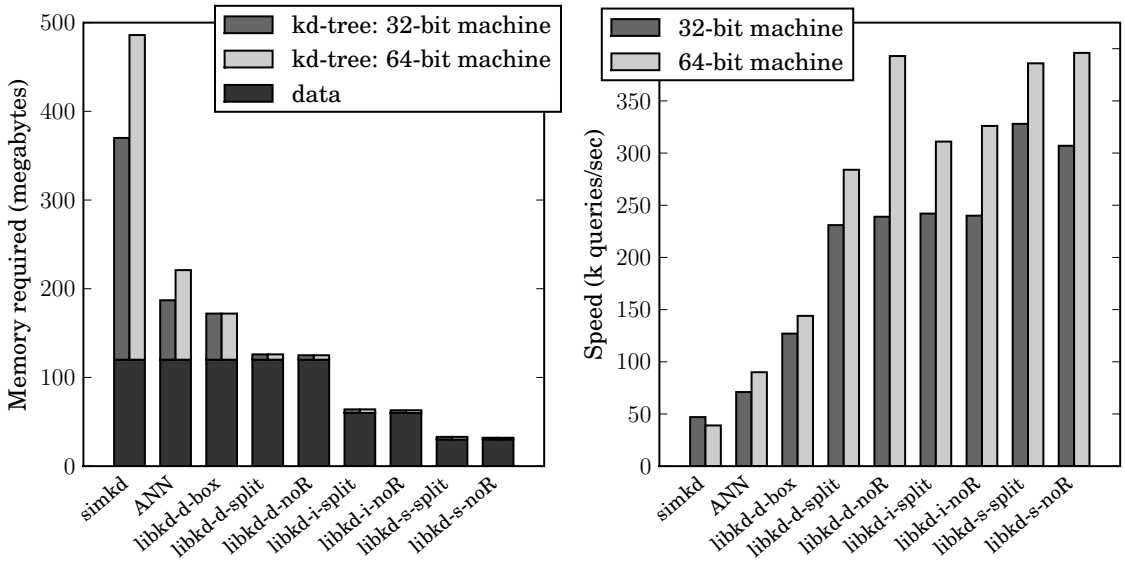


Figure 4.9: Benchmark results. **Left:** Memory requirements of the various implementations. The `simkd` and `ANN` methods use different amounts of memory on 32- and 64-bit machines because the size of pointers is larger in 64-bit machines, and these methods use many pointers. `libkd` uses no pointers internally so uses the same amount of memory on 32-bit and 64-bit machines. The memory usage of `libkd` is significantly smaller than the competing implementations. When using 32-bit or 16-bit integers (`i` and `s` variants, respectively) to store the data, rather than `doubles`, the memory requirements are reduced even further, though at the expense of some approximation error. **Right:** Speed (in thousands of queries per second) of the various implementations. `libkd` is always faster than the competition. The splitting-plane variant is significantly faster than the bounding-box variant. The `noR` trick seems to help on the 64-bit machine but have little effect on the 32-bit machine. Note that the 64-bit CPU is an AMD while the 32-bit CPU is an Intel, so these differences are likely due to the many differences in the chips rather than the word size *per se*. Given these differences, practitioners are urged to perform benchmarks on their own hardware and their own data.

increase when using a 64-bit machine, while `libkd` stays fixed because we use no extraneous pointers. Note that the data points themselves require 120 MB of space, so the memory overhead of a `libkd` tree is only a few percent, compared to 50 to 300 percent for `ANN` and `simkd`.

4.50 In this test, splitting planes are much faster than bounding boxes. Using trick 4.4.9 to avoid storing the offset arrays both reduces the memory required and slightly increases the speed. Using trick 4.4.11 to use smaller data types (32- and 16-bit integers) reduces the memory footprint by a factor of two or four, with a corresponding increase in speed. It also incurs some approximation error. In this test, we found that when using 32-bit integers, we still always produced the correct nearest neighbour, and the absolute error in the distance estimates was never more than 10^{-9}. When using 16-bit integers, we produced the correct nearest neighbour 99.5% of the time, and the absolute distance error was less than 10^{-4}.

4.51 In summary, these results show that, on this benchmark, by using the tricks presented `libkd` is three times faster and incurs one-fifteenth the memory overhead compared to the closest competitor. While benchmarks are not listed among the three canonical types of lies ("lies, damned lies, and statistics"), we urge practitioners to test these implementations on their own data. Our testing and benchmarking software is available for download.

4.6 Conclusion

4.52 Kd-trees are a fun and easy data structure for searching in k-dimensional space. A careful implementation can be orders of magnitude faster than a naïve one. Readers wishing to reap the benefits of such a careful implementation, without building it themselves, are encouraged to download the GPL'd `libkd`. Patches are welcome!

Chapter 5

Conclusion

5.1 Contributions

5.1 The *Astrometry.net* system applies the framework of geometric hashing to the astronomical problem of automated astrometric calibration of images. This can be seen as an instance of object recognition in which the individual objects to be recognized—stars and galaxies—are almost completely indistinctive at the resolution of typical images. The geometric relationships between the objects, however, can be used to build very distinctive features. The problem is made easier by the fact that the stars are very distant, so the viewpoint is essentially fixed, and while the stars do move, their motions are small enough that their geometric relationships change very little. The problem is difficult largely for its sheer scale: typical images cover one-millionth of the surface area of the sky or less, and errors of various types mean that stars are lost and gained in both the image and the reference catalog of stars.

5.2 Chapters 2, 3 and 4 highlight the main aspects of the system: a geometric feature-indexing method that is able to generate hypothesized matches; a robust probabilistic scheme for testing these hypotheses; and a data structure implementation that allows the whole system to operate at a speed that is acceptable for it to be used as a practical tool.

5.3 The specific contributions include the following:

- The design of our "quad" geometric hash code, including the constraints that the hashed stars be within the circle defined by the "framing" stars, and the constraints that break the symmetries of the hash code;
- The idea of using a kd-tree rather than a hash table to store the geometric hash codes;
- A justified probabilistic framework for the verification of the hypotheses generated by the geometric hashing system, including the use of Bayesian decision theory to set thresholds in a rational way;

- A method for constructing indices that tiles the sky uniformly with geometric features but avoids over-reliance on any individual star;

- Extensive evaluation of the *Astrometry.net* system, including a large-scale study using the Sloan Digital Sky Survey, and studies using a variety of other imagery to explore edge cases;

- In particular, an evaluation of the performance of indices using triangles, quadruples, and quintuples of stars; and

- A time- and space-efficient implementation of the kd-tree data structure.

5.2 Future work

5.2.1 Tuning of the *Astrometry.net* system

5.4 While the *Astrometry.net* system as presented here is successful at recognizing a wide variety of astronomical images, and is reasonably fast, there are several aspects of the system that could be tuned to improve overall performance in practice.

5.5 As detailed in chapter 2, when attempting to recognize an image we simply build each valid quad (or triangle or quintuple) using stars in the image, starting with the brightest stars. Different strategies for ordering the quads we test could lead to improved performance. When an image contains a bright "distractor"—an object that looks like a star but isn't a real star—the system will typically build a very large number of quads that include the distractor, none of which can possibly produce a correct result. Instead of building quads strictly based on brightness ordering, we could choose to build only a limited number of quads from any star or pair of stars (perhaps queueing these "overused" stars for further examination later) in order to examine more faint stars sooner. Alternatively, we could sample quads from the image in a probabilistic manner, taking

into account the brightness of each star and perhaps the number of times it has already been sampled.

5.6 Typically, the *Astrometry.net* system has several indices in which it must search for matches to each quad in the image. Currently, we simply proceed in lock-step: for each quad, we search each index in series. This is an embarassingly parallel problem, so distributing the indices across multiple processor cores (and multiple machines) would lead to speedups. More importantly, however, we suspect that it would be beneficial not to proceed in lock-step but to split the indices by angular scale and let the wide-angle indices examine more quads. Since searching in wide-angle indices is fast, this would tend to balance the amount of computational effort (rather than the number of quads) applied to each scale. More generally, the computational effort could be distributed based on the (expected or measured) properties of the images to be recognized, with the computation being ordered so that the most successful indices are checked first. Indeed, we could build relative sparse indices that could be checked quickly to recognize "easy" images, and denser, slower indices to recognize "hard" images.

5.7 Our experiments in section 2.3.1.7 showed that for Sloan Digital Sky Survey images, a quad-based index was faster than either a triangle- or quintuple-based index. We expect that the relative speeds will changes as a function of image scale, and that at some large rangular scale a triangle-based index will be faster. We have not measured this, partly because we lack a large uniform test set of wide-angle images. Similarly, we found that for SDSS images and a quad-based index, using a voting scheme—waiting for multiple agreeing hypotheses to accumulate before running the verification process—was slower and very memory-intensive (since our implementation stored every hypothesis). With a triangle-based index, a voting scheme could be beneficial, because many more false hypotheses are generated, and a voting scheme is supposed to reduce the number of times the relatively expensive verification procedure is run. In order to decrease the memory requirements, we could use a light-weight voting scheme that stores only some

summary information such as the *healpix* containing the image center; we would then run the verification procedure on any new hypothesis whose image center healpix (or any of its healpix neighbours) had already been hit by a previous hypothesis.

5.8 In principle, decisions about which indices to use (triangles, quads, or quints; or density of features), whether or not to use voting, the number of votes required before running the verification procedure, and other search parameters, could be chosen at run-time based on their relative costs (in terms of CPU time or memory) and the properties of the images to be recognized and the desired operating characteristics (speed versus recognition rate, for example). We suspect that the optimal structural aspects of the search (which indices to use, and whether or not to use voting) are determined almost exclusively by the angular scale of the images to be recognized, and can thus be optimized off-line. Trading speed for recognition rate, in contrast, can be achieved by, for example, using indices containing fewer features, or decreasing the feature-space matching distance. Using a sparser index simply decreases the number of potential matches for each image, but increases the search speed. Decreasing the feature-space matching distance means that some true matches will not be found because positional noise of the stars in the image or index moves the feature further than the distance threshold in feature space. We have not investigated how to trade off between these approaches so as to optimize the speed of the system at a given target recognition rate, but it could be done by simulating the positional errors of stars in the image and the differences in brightness ordering between the image and index.

5.2.2 Additions to the *Astrometry.net* system for practical recognition of astronomical images

5.9 The *Astrometry.net* system described here is intended to be equally able to recognize images from any part of the sky and of any scale, using only the information contained in the image pixels. In many settings, however, the images to be recognized are not

uniformly distributed, and other information is available. This section mentions a few extensions to the system that could improve its speed or recognition rate in such real-world settings.

5.10 We have focused on building a system that is equally able to recognize images from any part of the sky, but the coverage of astronomical imaging is highly non-uniform. Among both amateur astrophotographers and professional astronomers, images of specific astronomical objects (such as Messier objects and NGC/IC galaxies) are far more common than images of "blank sky". Yet the USNO-B reference catalog often contains errors in regions of the sky containing bright stars, nebulosity, or high stellar density, so an index built from that reference catalog will tend to perform *worse* in the parts of the sky that are most commonly imaged. Building a specialized index (using an appropriate reference catalog) that is able to recognize the most commonly imaged regions of the sky would improve the overall speed and recognition rate of the system.

5.11 A related problem is that the brightest stars in an image are often poorly localized due to saturation, diffraction spikes, halos, and bleeding of the CCD; our brightness-ordering heuristics should take this into account, perhaps by preferring to build quads from stars within a (scale- and location-dependent) range of brightnesses, rather than simply preferring the brightest stars.

5.12 Our approach presumes that individual stars and unresolved galaxies are completely indistinctive and cannot be used individually to recognize images. We therefore ignore the *appearance* of astronomical sources and focus on their relative *geometry*. Yet resolved galaxies and nebulae are commonly imaged, and their appearances can be quite distinctive, as indicated by the evocative names of the Sombrero, Whale, Whirlpool, and Sunflower galaxies and the Horsehead, Running Chicken, Helix, Cat's Paw, and Eagle nebulae. Since resolved galaxies and regions of nebulosity are often problematic both in reference catalogs and for source extraction routines, a pattern-recognition system that could recognize these astronomical objects based on their *appearance* would be a good

complement to the geometry-based approach of *Astrometry.net*.

5.13 Currently, the *Astrometry.net* web service uses a static set of indices. Ideally, however, it would *learn* about the sky and constantly update its reference catalog and indices based on the stars that it finds in the images it encounters. A system that did this would be able to achieve some remarkable feats: it could *patch* the USNO-B reference catalog in regions where the catalog has flaws. It could *deepen* the reference catalog, since many images contain stars that do not appear in the catalog. By building new indices from its deeper reference catalog, it would be able to recognize narrow-field images in regions that had been previously imaged many times. In this way, it could adapt to the non-uniform *interest* of astronomers in different regions of the sky. For example, if the system had access to all astronomical imaging, it would easily be able to recognize the narrow-field images from the Hubble Space Telescope, because every Hubble image is preceded by ground-based imaging. How exactly a system would update and maintain an astrometric reference catalog based on a huge amount of imaging is a question on which we have speculated, in vague terms, in our "theory of everything" paper [33].

5.14 Many of the images submitted to the *Astrometry.net* web service are produced by consumer-grade digital cameras and contain EXIF (exchangeable image file format) tags, which include meta-data about the image such as the camera model, lens, focal length, exposure time, aperture, and date. We currently ignore these meta-data, but they could be quite useful. Given the focal length of the lens and the camera model (thus the sensor geometry), we can compute the pixel scale (in arcseconds per pixel), which makes recognition of the image easier. Perhaps more interestingly, we could build up statistics on the distortion patterns of different lenses, and given a new image, apply the inverse distortion before attempting to recognize the image. Of course, we should never completely trust the information in the EXIF headers, but we could use it as a possibly-correct hint about the image.

5.2.3 Other kinds of calibration

5.15 After recognizing (*i.e.*, astrometrically calibrating) an image, we can match stars in the image with stars in a reference catalog. This enables other kinds of calibration and meta-data creation. For example, we can estimate the photometric calibration and bandpass filter of the image, or estimate the date the image was taken.

5.16 Automated photometric calibration and bandpass estimation proceeds by matching stars in the image to stars in the reference catalog, and finding the best-fitting parameters that allow us to convert between "instrumental" fluxes in the image and magnitudes in one or several of the bands in the reference catalog. If we find that a single band in the reference catalog fits the image well, it is likely that the bandpass filters of the reference catalog and the image are similar; if more than one band in the reference catalog is required to yield a satisfactory fit, then the bandpass of the image is likely different than any of those in the reference catalog.

5.17 Astrometric dating—calibrating the date on which an image was taken—proceeds by computing the positions of reference catalog stars at a given date (using their tabulated proper motions) and finding the date when they best match those of the image stars [6]. Perhaps surprisingly, the distribution of proper motions of stars is such that even over a time baseline of 50 to 100 years, enough of the stars move little enough that we are still able to recognize the images based on their relative positions, yet enough of the stars move enough that their changes in position contain sufficient information to constrain the date on which the image was taken. For typical historical photographic plates, the date on which the image was taken is constrained to within a few years using this method. By adding photometric information about variable stars, and about transients that might appear in the image (planets, comets, satellites), we could fine-tune the date estimate to much greater precision.

5.2.4 Using heterogenous images for science

5.18 The *Astrometry.net* system makes a large amount of heterogeneous data readily available for scientific investigation. How can large collections of images be analyzed to answer scientific questions? This is a surprisingly ill-explored research area in astronomy. We have been doing some early experiments building generative models of astronomical images, and doing probabilistic inference in these models, and we feel this is a very promising avenue for combining the information in images with very different resolutions, sensitivities, and bandpasses. This is an exciting area for future work.

Bibliography

[1] K. N. Abazajian et al. The Seventh Data Release of the Sloan Digital Sky Survey. *Astrophysical Journal Supplement Series*, 182:543–558, June 2009. arXiv:0812.0649.

[2] G. Aldering, G. Adam, P. Antilogus, P. Astier, R. Bacon, S. Bongard, C. Bonnaud, Y. Copin, D. Hardin, F. Henault, D. A. Howell, J.-P. Lemonnier, J.-M. Levy, S. C. Loken, P. E. Nugent, R. Pain, A. Pecontal, E. Pecontal, S. Perlmutter, R. M. Quimby, K. Schahmaneche, G. Smadja, and W. M. Wood-Vasey. Overview of the Nearby Supernova Factory. In J. A. Tyson and S. Wolff, editors, *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 4836, pages 61–72, December 2002.

[3] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations Of Computer Science*, pages 459–468. IEEE Computer Society, 2006.

[4] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[5] D. H. Ballard. Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2):111–122, 1981.

[6] J. T. Barron, D. W. Hogg, D. Lang, and S. Roweis. Blind Date: Using Proper Mo-

tions to Determine the Ages of Historical Images. *Astrophysical Journal*, 136:1490–1501, October 2008. arXiv:0805.0759.

[7] Jonathan T. Barron, Christopher Stumm, David W. Hogg, Dustin Lang, and Sam Roweis. Cleaning the USNO-B catalog through automatic detection of optical artifacts. *The Astronomical Journal*, 135(1):414–422, 2008.

[8] Jeffrey S. Beis and David G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Conference on Computer Vision and Pattern Recognition*, pages 1000–1006, 1997.

[9] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[10] E. Bertin. Automatic astrometric and photometric calibration with SCAMP. In Carlos Gabriel, Christophe Arviset, Daniel Ponz, and Enrique Solano, editors, *Astronomical Data Analysis Software and Systems 15*, ASP Conference Series, Vol. 351, pages 112–, San Francisco, 2006. Astronomical Society of the Pacific.

[11] E. Bertin and S. Arnouts. SExtractor: Software for source extraction. *Astronomy and Astrophysics Supplement*, 117:393–404, June 1996.

[12] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[13] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Survey*, 33(3):322–373, 2001.

[14] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The Bloomier filter: an efficient data structure for static support lookup tables. In *SODA '04: Proceedings*

of the fifteenth annual ACM-SIAM symposium on Discrete algorithms, pages 30–39, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[15] D. S. Clouse and C. W. Padgett. Small field-of-view star identification using Bayesian decision theory. *IEEE Transactions on Aerospace and Electronic Systems*, 36(3):773–783, Jul 2000.

[16] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In Jack Snoeyink and Jean-Daniel Boissonnat, editors, *Symposium on Computational Geometry*, pages 253–262. ACM, 2004.

[17] M. Davis et al. The All-Wavelength Extended Groth Strip International Survey (AEGIS) Data Sets. *Astrophysical Journal Letters*, 660:L1–L6, May 2007. arXiv:astro-ph/0607355.

[18] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[19] Kan Deng and Andrew Moore. Multiresolution instance-based learning. In *Proceedings of the Twelfth International Joint Conference on Artificial Intellingence*, pages 1233–1239, San Francisco, 1995. Morgan Kaufmann.

[20] S. G. Djorgovski, A. Mahabal, A. Drake, C. Donalek, E. Glikman, M. Graham, R. Williams, T. Morton, A. Bauer, C. Baltay, D. Rabinowitz, R. Scalzo, P. Nugent, and Palomar-Quest Survey Team. Exploration of the Time Domain With Palomar-Quest Survey. In *Bulletin of the American Astronomical Society*, volume 41 of *Bulletin of the American Astronomical Society*, pages 258–+, January 2009.

[21] Ying Dong, Fei Xing, and Zheng You. Brightness independent 4-star matching algo-

rithm for lost-in-space 3-axis attitude acquisition. *Tsinghua Science & Technology*, 11(5):543–548, 2006.

[22] R. O. Duda and P. E. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15:11–15, 1972.

[23] H. C. Ford et al. Advanced camera for the Hubble Space Telescope. In P. Y. Bely and J. B. Breckinridge, editors, *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 3356 of *Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*, pages 234–248, August 1998.

[24] Jerome H. Freidman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.

[25] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 518–529. Morgan Kaufmann, 1999.

[26] K. M. Górski, A. J. Banday, E. Hivon, and B. D. Wandelt. HEALPix — a Framework for High Resolution, Fast Analysis on the Sphere. In D. A. Bohlender, D. Durand, and T. H. Handley, editors, *Astronomical Data Analysis Software and Systems XI*, volume 281 of *Astronomical Society of the Pacific Conference Series*, pages 107–+, 2002.

[27] J. Grindlay, S. Tang, R. Simcoe, S. Laycock, E. Los, D. Mink, A. Doane, and D. Champine. DASCH to Measure (and preserve) the Harvard Plates: Opening the 100-year Time Domain Astronomy Window. *Astronomical Society of the Pacific Conference Proceedings*, in press, 2009.

[28] E. J. Groth. A pattern-matching algorithm for two-dimensional coordinate lists. *Astrophysical Journal*, 91:1244–1248, May 1986.

[29] J. E. Gunn et al. The Sloan Digital Sky Survey Photometric Camera. *Astrophysical Journal*, 116:3040–3081, December 1998. arXiv:astro-ph/9809085.

[30] Chris Harvey. New algorithms for automated astrometry. Master's thesis, University of Toronto, 2004.

[31] Gisli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces (survey article). *ACM Trans. Database Syst.*, 28(4):517–580, 2003.

[32] E. Høg et al. The tycho-2 catalogue of the 2.5 million brightest stars. *Astronomy and Astrophysics*, 355:L27–30, 2000.

[33] D. W. Hogg and D. Lang. Astronomical imaging: The theory of everything. In C. A. L. Bailer-Jones, editor, *Classification and Discovery in Large Astronomical Surveys*, volume 1082 of *AIP Conference Proceedings*, pages 331–338, 2008. arXiv:0810.3851.

[34] Daniel P. Huttenlocher and Shimon Ullman. Recognizing solid objects by alignment with an image. *Int. J. Comput. Vision*, 5(2):195–212, 1990.

[35] Z. Ivezic, J. A. Tyson, R. Allsman, J. Andrew, and R. Angel. LSST: from Science Drivers to Reference Design and Anticipated Data Products. arXiv:astro-ph/0805.2366, May 2008.

[36] H. Jenkner, R. E. Doxsey, R. J. Hanisch, S. H. Lubow, W. W. Miller, III, and R. L. White. Concept for the Hubble Legacy Archive. In C. Gabriel, C. Arviset, D. Ponz, and S. Enrique, editors, *Astronomical Data Analysis Software and Systems XV*, volume 351 of *Astronomical Society of the Pacific Conference Series*, pages 406–+, July 2006.

[37] J. L. Junkins, C. C. White, III, and J. D. Turner. Star pattern recognition for real time attitude determination. *Journal of the Astronautical Sciences*, 25:251–270, 1977.

[38] N. Kaiser, G. Wilson, G. Luppino, and H. Dahle. A photometric study of the supercluster MS0302 with the UH8K CCD camera: Image processing and object catalogs. arXiv:astro-ph/9907229v1, 1999.

[39] David G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.

[40] A. Kirsch and M. Mitzenmacher. Distance-sensitive Bloom filters. In *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2006.

[41] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[42] Y. Lamdan, J. T. Schwartz, and H. J. Wolfson. Affine invariant model-based object recognition. *IEEE Transactions on Robotics and Automation*, 6(5):578–589, oct 1990.

[43] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, Jul 1960.

[44] Carl Christian Liebe. Pattern recognition of star constellations for spacecraft applications. *IEEE Aerospace and Electronic Systems Magazine*, 8(1):31–39, 1993.

[45] Carl Christian Liebe, Konstantin Gromov, and David Matthew Meller. Toward a stellar gyroscope for spacecraft attitude determination. *Journal of Guidance, Control, and Dynamics*, 27(1):91–99, 2004.

[46] R. Lupton, J. E. Gunn, Z. Ivezić, G. R. Knapp, and S. Kent. The SDSS Imaging Pipelines. In F. R. Harnden, Jr., F. A. Primini, and H. E. Payne, editors, *Astro-*

nomical Data Analysis Software and Systems X, volume 238 of *Astronomical Society of the Pacific Conference Series*, pages 269–+, 2001.

[47] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166, 1990.

[48] David MacKay and Sam Roweis. Astronomical image recognition. http://www.inference.phy.cam.ac.uk/mackay/presentations/stars.

[49] C. Martin and GALEX Science Team. The Galaxy Evolution Explorer (GALEX). In *Bulletin of the American Astronomical Society*, volume 35 of *Bulletin of the American Astronomical Society*, pages 1363–+, December 2003.

[50] D. Mink, A. Doane, R. Simcoe, E. Los, and J. Grindlay. The Harvard Plate Scanning Project. In M. Tsvetkov, V. Golev, F. Murtagh, and R. Molina, editors, *Virtual Observatory: Plate Content Digitization, Archive Mining and Image Sequence Processing*, pages 54–60, April 2006.

[51] D. J. Mink. WCSTools 4.0: Building Astrometry and Catalogs into Pipelines. In C. Gabriel, C. Arviset, D. Ponz, and Solano E., editors, *Astronomical Data Analysis Software and Systems XV*, volume 15 of *Astronomical Society of the Pacific Conference Series*, pages 204–+, 2006.

[52] D. G. Monet et al. The USNO-B Catalog. *Astrophysical Journal*, 125:984–993, February 2003.

[53] Andrew Moore. The Anchors Hierarchy: Using the triangle inequality to survive high dimensional data. Technical Report CMU-RI-TR-00-05, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, February 2000.

[54] Andrew Moore and John Ostlund. Simple kd-tree source code. http://www.autonlab.org.

[55] P. E. Nugent, Palomar Transient Factory, and DeepSky Project. DeepSky and PTF: A New Era Begins. In *Bulletin of the American Astronomical Society*, volume 41 of *Bulletin of the American Astronomical Society*, pages 419–+, January 2009.

[56] C. Padgett and K. Kreutz-Delgado. A grid algorithm for autonomous star identification. *IEEE Transactions on Aerospace and Electronic Systems*, 33(1):202–213, 1997.

[57] A. Pál and G. Á. Bakos. Astrometry in Wide-Field Surveys. *The Publications of the Astronomical Society of the Pacific*, 118:1474–1483, October 2006.

[58] D. Schade. Astrometry and the Virtual Observatory. In P. K. Seidelmann and A. K. B. Monet, editors, *Astrometry in the Age of the Next Generation of Large Telescopes*, volume 338 of *Astronomical Society of the Pacific Conference Series*, pages 156–+, October 2005.

[59] M. F. Skrutskie et al. The Two Micron All Sky Survey (2MASS). *Astrophysical Journal*, 131:1163, 2006.

[60] Robert F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(1):579–589, 1991.

[61] A. S. Szalay. The National Virtual Observatory. In F. R. Harnden, Jr., F. A. Primini, and H. E. Payne, editors, *Astronomical Data Analysis Software and Systems X*, volume 238 of *Astronomical Society of the Pacific Conference Series*, pages 3–+, 2001.

[62] D. Tody and R. Plante. Simple Image Access Specification Version 1.0, May 2009. `http://ivoa.net/Documents/PR/DAL/PR-SIA-1.0-20090521.html`.

[63] Bill Triggs, P. McLauchlan, Richard Hartley, and A. Fitzgibbon. Bundle adjustment—a modern synthesis. In B. Triggs, A. Zisserman, and R. Szeliski, ed-

itors, *Vision Algorithms: Theory and Practice*, volume 1883 of *Lecture Notes in Computer Science*, pages 298–372. Springer-Verlag, 2000.

[64] J. K. Uhlmann. Metric trees. *Applied Mathematics Letters*, 4(5):61–62, 1991.

[65] F. G. Valdes, L. E. Campusano, J. D. Velasquez, and P. B. Stetson. FOCAS Automatic Catalog Matching Algorithms. *Publications of the Astronomical Society of the Pacific*, 107:1119–, November 1995.

[66] R. E. Williams et al. The Hubble Deep Field: Observations, Data Reduction, and Galaxy Photometry. *Astrophysical Journal*, 112:1335–+, October 1996. arXiv:astro-ph/9607174.

[67] H. J. Wolfson and I. Rigoutsos. Geometric hashing: an overview. *IEEE Computational Science and Engineering*, 4(4):10–21, 1997.

[68] T. Yairi, K. Hirama, and K. Hori. Fast and simple topological map construction based on cooccurrence frequency of landmark observation. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 3, pages 1263–1268 vol.3, 2001.

[69] D. G. York et al. The Sloan Digital Sky Survey: Technical Summary. *Astrophysical Journal*, 120:1579–1587, September 2000. arXiv:astro-ph/0006396.

[70] N. Zacharias, D. G. Monet, S. E. Levine, S. E. Urban, R. Gaume, and G. L. Wycoff. The Naval Observatory Merged Astrometric Dataset (NOMAD). In *Americal Astronomical Society 205th Meeting*, 2005.