Efficient Fastener Comparison

Eugene Brevdo, *Rensselaer Polytechnic Institute*
Aaron Hatch, *Rensselaer Polytechnic Institute*
David Akman, *Rensselaer Polytechnic Institute*

ECSE-4560 Senior Design Project
Project Advisor: Professor Richard Radke
December 16, 2005
Rensselaer Polytechnic Institute

ABSTRACT

This document describes the design and implementation of a system created to generate a distance measurement between any two hardware fasteners using images of the fasteners. This system is designed with three major classes of fasteners in mind: machine screws, wood screws, and nails. The distance metric is designed in such a way that it will return a small distance between objects of the same main classification, but will give a large distance between objects of dissimilar classes.

Our system is implemented in Matlab. Analysis begins by rectifying the image to remove the effects of our non-ideal camera. After this pre-processing stage, we simplify the image to allow easy selection of the component objects (individual fasteners). This simplification is achieved through value based thresholding and morphological operations. Objects are selected based on surface area filtering, and Matlab's regionprops() function. Finally, the objects are rotated into a standard orientation. Support is offered for multiple fasteners per image.

Feature extraction begins with calculation of the contour function of the object under consideration. After a contour is calculated, features such as width, length, threading count, threading density, tip angle, and more, can be extracted. Also, head type can be determined from a "head" view of the object. These features are entered into a feature vector. A metric is used to calculate a meaningful distance between two such feature vectors. Finally, classification is implemented with the aid of Support Vector Machines.

Table of Contents

Figures and Tables

Introduction

This document describes the implementation of a Signal Processing Design senior capstone project. It describes in detail how each requirement outlined in the project proposal was fulfilled and details extensions to the project scope. The members of the team are Eugene Brevdo, Aaron Hatch, and David Akman.

The project under consideration is the task of comparing assorted fasteners from at least three main categories: wood screws, machine screws, and nails. The focus of the project is on the comparison of the objects by a distance metric developed by the team. All information used in the similarity measure and other parts of the project are gathered from image processing. The goal of this distance metric is to produce a similarity measure between any two fasteners.

We divided the project into seven subprojects that we implemented in sequence, each time using data extracted from the previous stages. The first phase was data collection. This phase consisted of gathering a sampling of various fasteners and taking high quality digital pictures of them for the training set. Next, we implemented image preprocessing, which included camera calibration and color space conversions. After image preprocessing, Matlab code was developed to simplify the object using thresholding techniques and morphological operations. Following the development of simplification, in the object selection stage, we developed algorithms to select the features of interest, cut them out of the background, and rotate them to a standard orientation.

After isolating the object and getting it into a standard format in the previous sub-projects, we began to develop feature extraction. We developed feature extraction algorithms to calculate various dimensions and to exact a contour of the object. Based on this contour, we measured properties such as symmetry, sinusoidal patterns, standard deviation, minima and maxima, first and second moments, and other properties that allowed us to determine features such as length and width, the pitch of the threads, threading density, tip angle, etc. From these features, we built a metric that calculates a distance between different fasteners and groups them naturally as a human would.

Finally, for extensions to the original project, we implemented a hard classification system, radial un-distortion of raw data, and processing of multiple fasteners in one image.

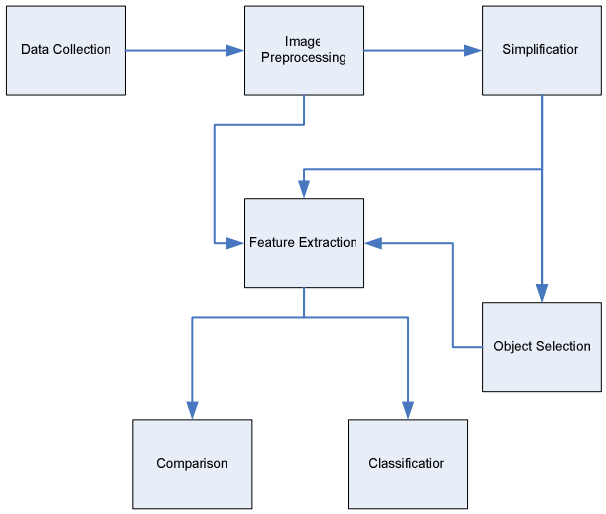

Figure 1 – Data Flow Between Sub-Projects

Design Procedure

Data Collection

We first searched for fastener image databases online and in paper catalogs. Our largest challenge was finding a consistent set of images. In many cases, photorealistic renderings or pictures are available at various hardware catalog sites, but more often than not, a single image is used to represent multiple fasteners. It was impossible to find high-resolution photographs, especially photographs with consistent camera position. Finally, none of these images included objects, such as rulers, that can be used to infer size. We finally abandoned this method.

A standardized data collection procedure was necessary for satisfactory results. We designed and built a camera rig and imaging environment to hold constant a number of variables that can arise when performing data collection. These variables include:

- Digital camera used for data collection
- Camera parameters (including zoom, focus, and position with respect to the imaging plane)
- Background lighting conditions
- Image plane background (color and reflection parameters)
- Static target (fasteners are not moving, e.g., as in a conveyer belt system)

For variation in our data set, and due to financial considerations, we chose not to purchase a new fastener toolkit for our data. Instead, we collected old nails and screws from a variety of basements, backyard sheds, and old toolboxes. As a result, our collection includes a common mixture of fasteners: rusted and shiny, plastic and metal, with a variety of head styles, shapes, and uses. In many cases, we collected both *head* and *front* views: the *head* view shows the fastener head while the *front* view shows the fastener body and threading. The used condition of our fastener assortment more closely simulated the conditions in which a system such as ours is likely to be used and was seen as a benefit to the real world applicability of our design.

Image Preprocessing

Many of the external variables that cannot be fixed during data collection must be accounted for during preprocessing. These include intrinsically variable parameters, such as fastener properties that do not aid in fastener comparison. Specifically, we correct for specular reflection (glare) from polished fastener surfaces, non-uniform lighting conditions, and small camera displacements.

This first set of preprocessing routines includes the following: Conversion from the *RGB* (Red, Green, Blue) color space to the *HSV* (Hue, Saturation, Value) color space, a histogram-based method for background subtraction, and well-chosen cropping borders. Note that in an industrial environment, the camera can be fastened in a manner that allows automatic border cropping.

Oftentimes, our later algorithms assume that images are taken with an ideal camera in an ideal environment. However, real life cameras do not behave in this way, but distort special placement to an extent, especially around the borders of the image. This distortion can cause problems processing and extracting information from the image. This is especially true for images in which there are multiple fasteners with some close to the boundary of the image, or even just a single fastener but which is not centered. Although not an original requirement, we decided to include handling of these situations as an extra feature, making it necessary to correct this distortion.

In the ideal camera model, the fasteners sit on a plane that is flat and parallel to the camera plane. The camera does not distort images in a nonlinear manner. Since our camera is not ideal, software must be employed to remove the effects of these imperfections. We employed a camera calibration package to both approximate the intrinsic (internal) camera distortion parameters and to rectify (undistort) our images. Failure to rectify images will cause fasteners at the edges to appear curved. The curvilinear nature of the unrectified images similarly distorts basic fastener dimensions, such as length and width. These effects are especially apparent at the edges of an image, where straight lines are transformed into arcs with significant curvature.

Image Simplification

After preprocessing, the images in our data set very closely approximate those taken by an ideal camera. This enables the feature extraction algorithms to assume the image represent an ideal image. However, before we begin the feature extraction portion of our system, the image needs to be simplified from an RGB image to an image that more readily distinguishes the targets from the background. We call this the simplification stage.

Before a fastener can be selected, it first has to be differentiated from the background. We chose a color space that enhances the contrast between the fastener and the background. The HSV color space is composed of three components: hue, saturation, or value. Algorithms using only hue were of limited value with our data, since analysis of the hue spectrum of our image revealed that there was a wide range of hues in the background. Saturation also proved to be a poor choice to differentiate, since the image does not have uniform lighting over the entire surface. Value, however, provides an excellent means of differentiating our fasteners from the background, since the background has values within a relatively narrow band that doesn't fully overlap that of the fasteners, and most importantly, is fully distinct from the value range of the outer edges of the flattened image of the fastener.

Using value as the basis for differentiation, simple thresholding and logic operations can be used to remove background noise. The resulting black and white image contains a few solid and mostly connected fields of white representing the fastener and a few scattered bits of noise inside the threshold values. Matlab morphological operations can be used to remove stray noise and completely fill in the fastener outline, thus further simplifying this binary image.

Object Selection

A single frontal view image may contain multiple fasteners, all with different orientations. The "object selection" problem is often referred to as segmentation in Computer Vision literature. We experimented with several well-known and recent image segmentation algorithms. In general, these algorithms are too technically detailed to implement from scratch. Below is a survey of relevant literature and publicly available Matlab code for generalized image segmentation.

Recent work in segmentation has often revolved around three topics: Active Dynamic Contours (snakes), Propagating Level Set methods, and Graph-theoretic methods such as Normalized Cut and Min-Cut. Many of these are described in Forsyth and Ponce [1].

Snake-based methods usually involve minimizing the "energy" of a curve that surrounds an image (to "hone in" on the different segments). This segmented curve must be solved for numerically, the "honing in" is generally an iterative process that takes many steps to convergence. Snake methods will work well if given a properly preprocessed image and allowed to run for a sufficient amount of time. Unfortunately, snake algorithms took too long on images with our dimensions. Level set methods are

similar to energy minimizing dynamic contours. They require an iterative approach similar to snake-based methods and tend to share their slow performance.

As a direct implementation of dynamic contours was not feasible, we modified and experimented with a Matlab implementation by Wasilewski (U. Waterloo) [2]. Wasilewski has implemented both methods: level sets and energy minimization using dynamic contours. As we have already mentioned, these algorithms scaled badly for larger images and we eventually went on to try other alternatives.

We tried several graph-theoretic algorithms for segmentation. Unfortunately, some of the more popular methods, namely the Normalized Cut and the Min-Cut algorithms, require the solution of the eigenvalue problem on large dense matrices. These methods also do not scale well to the image dimensions we worked with.

Our segmentation algorithm is less general than those mentioned above in that it makes several assumptions about the image background and the sizes of the individual fasteners. Specifically, our segmentation relies on good background subtraction, thresholding, and noise removal during the Image Simplification phase. In effect, our algorithm accepts a binary "blob" image, tags all white blobs as possible fasteners, and uses Matlab's regionprops algorithm to efficiently find the bounding box for each of the blobs.

We empirically accept all blobs of size greater than 1% of the image pixel area as fasteners, cut them out from the image, and calculate the orientation of each fastener using our *orient* minimization algorithm. Once we know the fastener's orientation (including the position of the fastener's head), we go back to the original image, cut out a padded version of the fastener, rotate it so the long axis is vertical with the head at the top of the image, and cut out a clean outline of the fastener.

Feature Extraction

Once we have an image in a standard form, along with a well-defined outline of the fastener, it is then possible to analyze its contents. These images contain many vital pieces of information that we can utilize to compare the similarity between two different fasteners. This information varies from the obvious features, such as the length and width of the fastener, to the moments of the contour image.

The features we extract are all necessary for the development of an accurate similarity measure because each feature portrays an independent property of the fastener. To ensure we obtain information about the entire fastener, we extract data from both the front and head views of the fastener. Information such as length, width, thread pitch, and tip angle are obtained from the front view; the head type is an example of a feature extracted from the head view. These properties are stored in a feature vector.

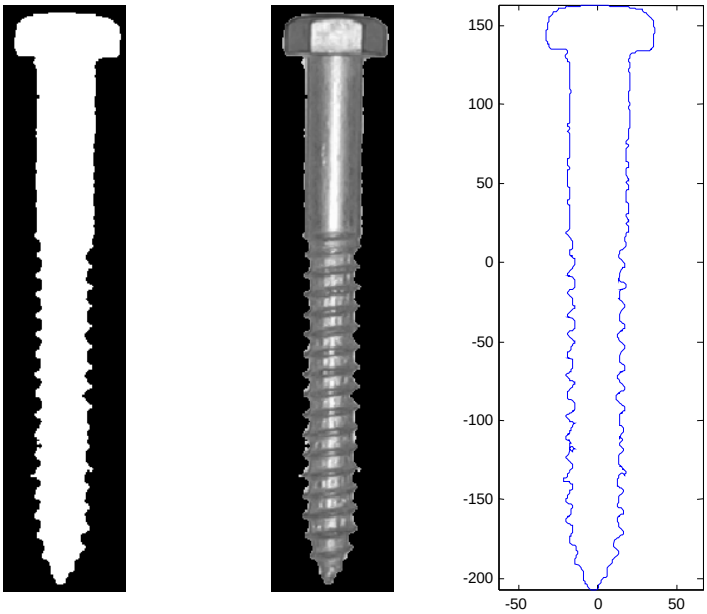

Figure 2 – Fastener Blob and Outline

Design Details

Data collection

Our data collection rig is a wooden structure with a 6mm coarse threaded screw at the top for our camera's tripod mount. We added a support peg to the rig to fix the camera at a second point and thus keep it stationary.

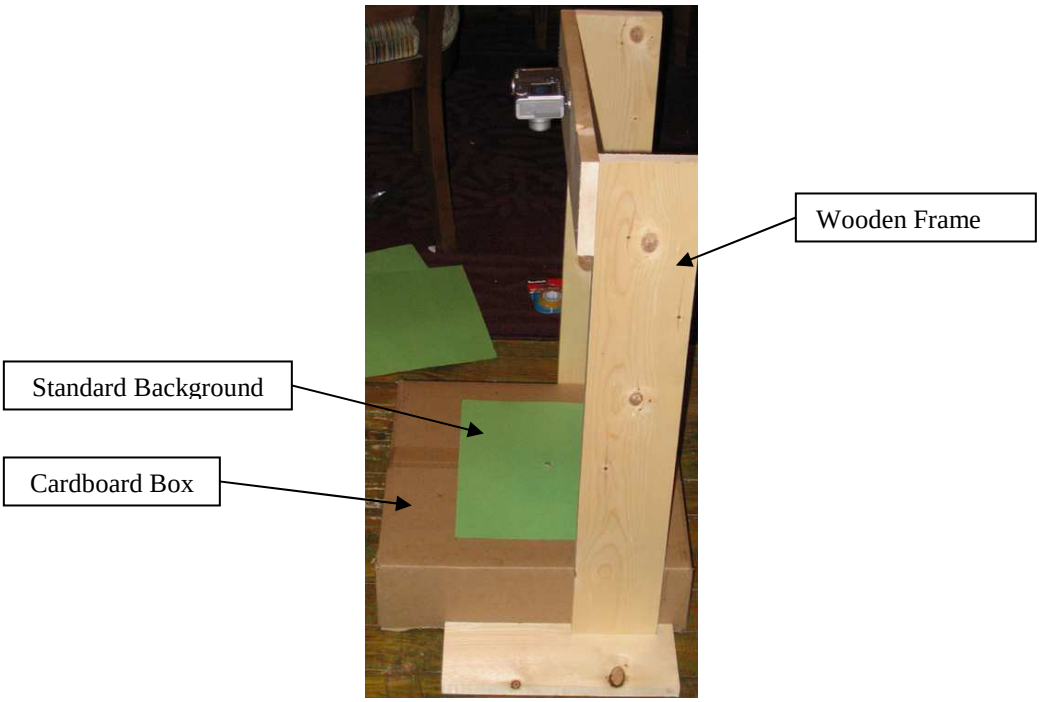

Figure 3 – Side View, Data Collection Rig

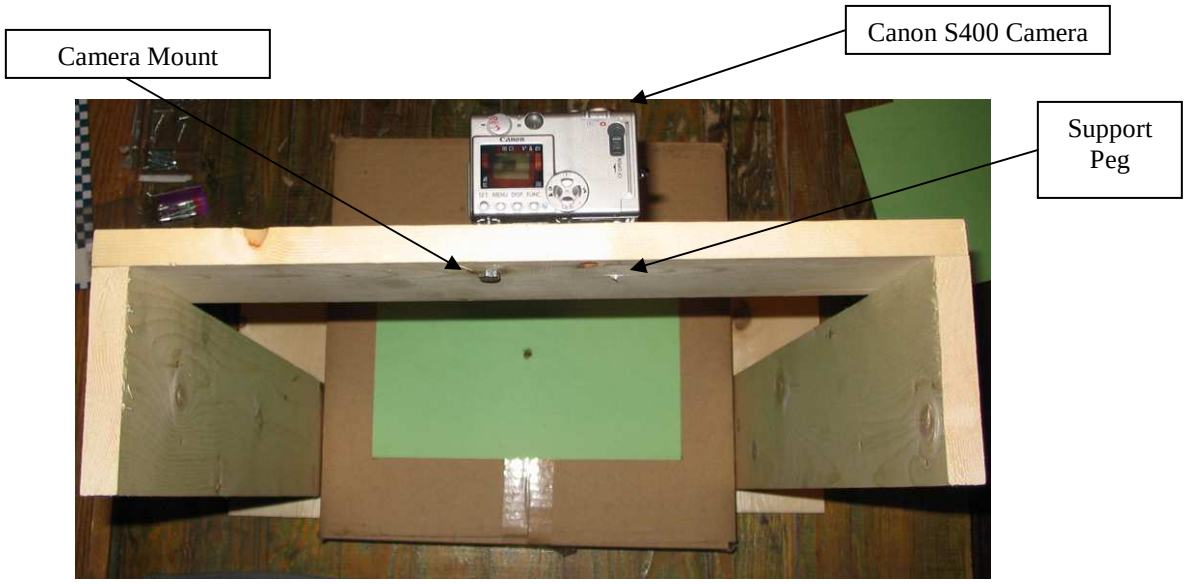

Figure 4 – Front View, Data Collection Rig

We used a Canon S400 camera for our preliminary data collection round. The camera configuration (including external and internal parameters) is provided in Table 1.

Distance from Image Plane	~ 2ft
Orientation	Normal
Image Resolution	Highest (4MP, 2272x1704, 180 px/in)
Image Format	JPEG Ultrafine (Compressed, 100%)
Exposure Time (Shutter Speed)	1/60 sec
F-Stop	2.8
Compressed bits/px	5.0
Apeture	3.0
Exposure Bias	0.0
Max Apeture	5.0
Light Metering mode	Pattern
Flash	On (Standard low light conditions)
Focal Length	7.4mm
Optical Zoom	None (Minimal zoom)

Table 1 – Canon S400 Data Collection Parameters

We extracted the information above from the EXIF field of one of our images. The main parameters we had control over were focus distance (focus was on the imaging plane), camera distance to image plane, flash, and the size and quality of the images.

Image preprocessing

Although not required in the basic proposal, our team decided to implement a radial un-distortion system as a "bell and whistle". This addition increases the accuracy of feature extraction measurements for fasteners along the outer edges of an image. Intrinsic camera parameters were calculated using the Camera Calibration Toolbox for Matlab (Bouguet, Caltech [3]). The toolbox uses ideas from [4],[5] for the intrinsic camera model and calibration methods. We printed a checkerboard pattern available from the following Toolbox help webpage:

http://www.vision.caltech.edu/bouguetj/calib_doc/htmls/own_calib.html

To take our calibration images, we first focused our camera on the background image plane, and then positioned the image. Figure 5 contains two of our 15 calibration images. Note that the width and height of each block was 28mm.

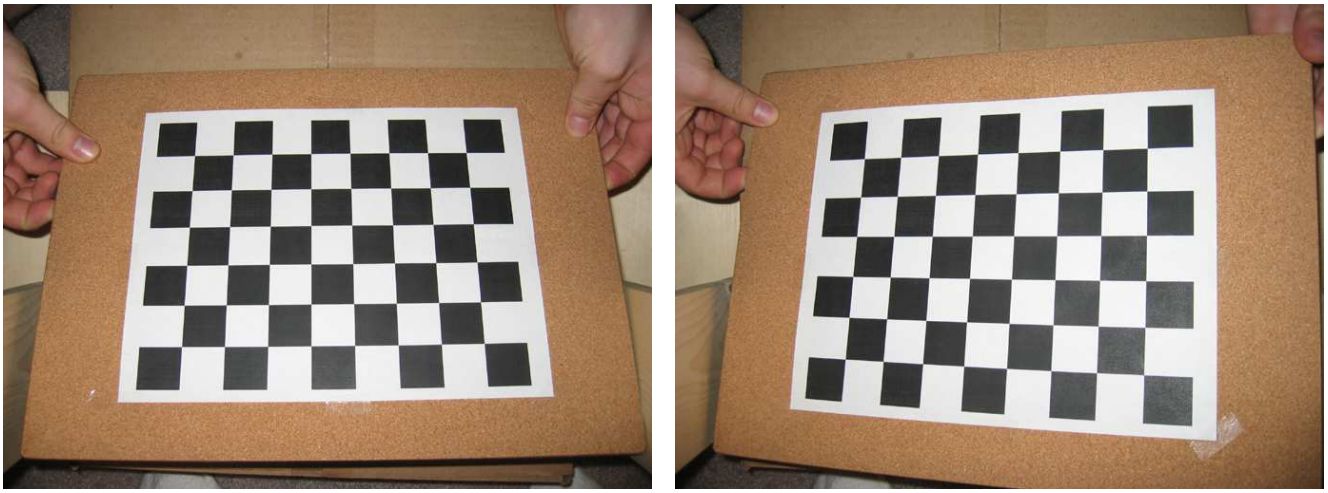

Figure 5 – Image Calibration Checkerboard Patterns

To calibrate, we followed the steps of the Toolbox first calibration tutorial. The intrinsic camera parameters given our specific focus, resolution, zoom, and flash settings are shown in Figure 6, along with a visualization of the camera image distortion for 2272x1704 pixel images.

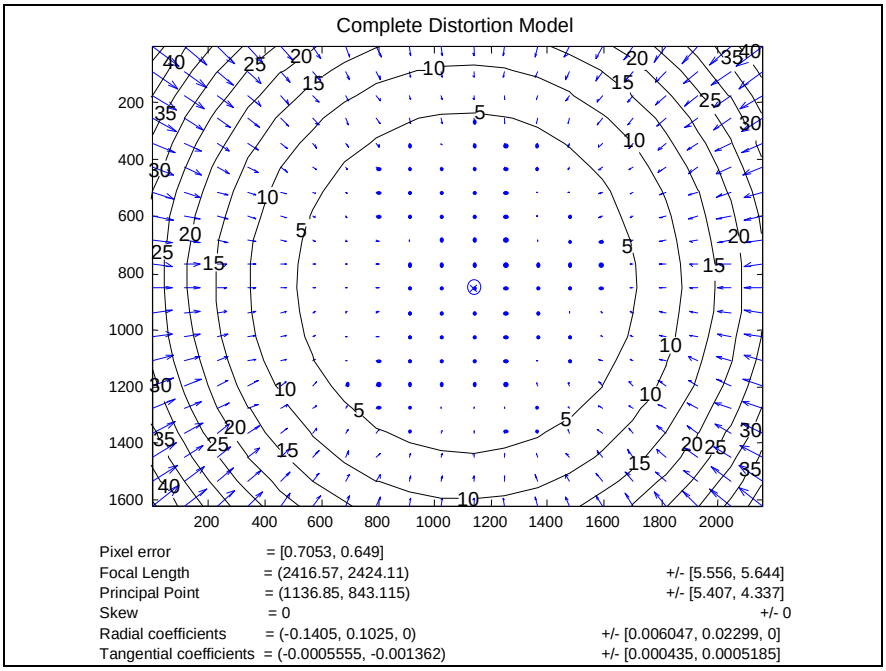

Figure 6 – Canon S400 complete distortion model and intrinsic camera parameters

Fasteners at the center of the image will not be visibly distorted, whereas fasteners near the corners will have substantial nonlinear distortion. Figure 7 contains two images of a fastener, before and after rectification.

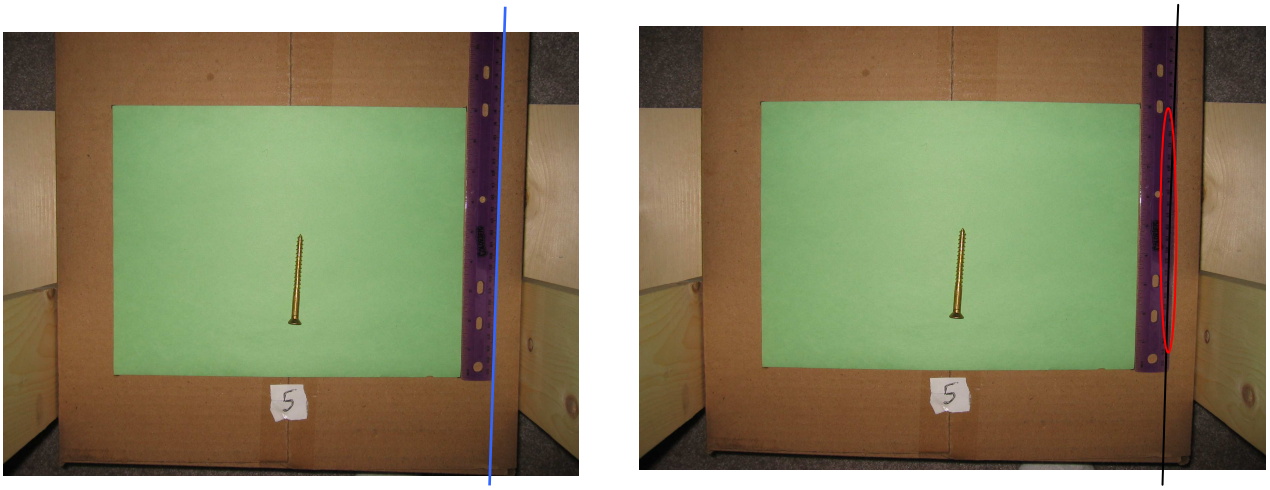

Figure 7 - Before and After Rectification

Next, all of the images must be rectified (undistorted). This step would have been unnecessary if fasteners are always located near the center of each image, since distortion near the center is minimal. However, we designed our project to be able to handle objects located near the corners of the image. This is useful for cases where there are multiple fasteners in the image, and where the fastener is poorly positioned before the snapshot.

We ran into a number of "Out of Memory" errors in Matlab 6.5 when using the Calibration Toolbox function `rect.m` on our 2272x1704 images. Specifically, the rectification function and its support function `apply_distortion.m` both create many temporary variables that cause memory errors on a machine running Windows XP with 712MB physical and 1.5GB virtual memory. We modified these files, added 'clear' statements to reclaim memory from unused temporary variables, and commented out code that created unused temporary variables. Many of the temporary variables take up 7-25MB each. Our changes solved the memory errors.

As each layer (R,G,B) of an image takes several minutes the process, each image takes 7-10 minutes processing time total. In the meantime, no other processes may run on the machine. We wrote a batch script to rectify our initial data set of 50 images; this operation took 7 hours. The process for rectifying a single JPEG image is shown in Figure 8.

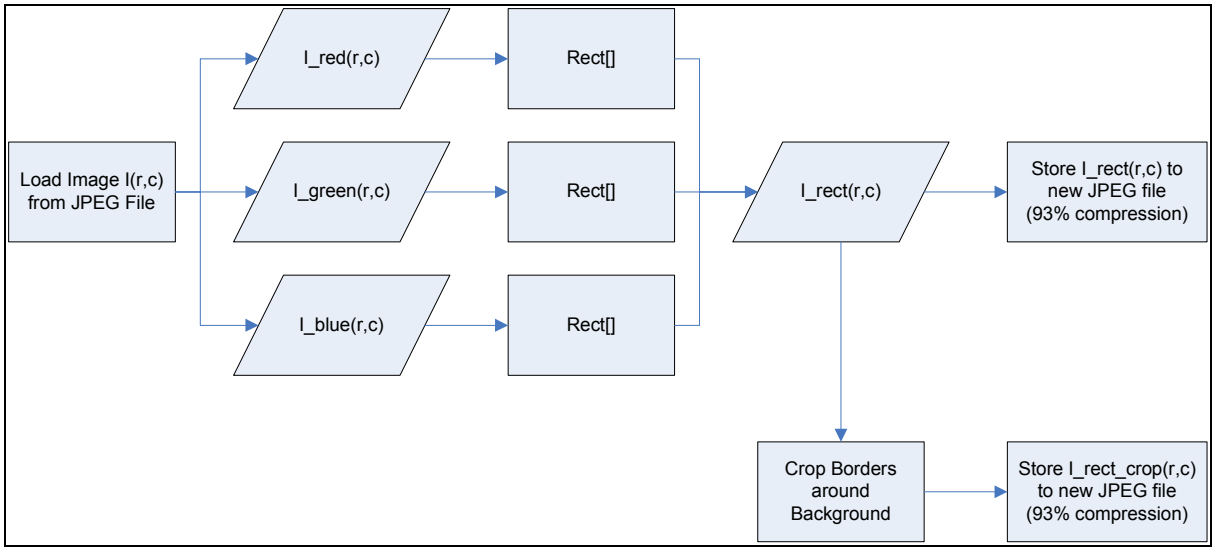

Figure 8 – Image Rectification Diagram

Rectified images allow us to use an *L2*-norm to calculate a pixel-to-distance ratio and an $(r,c) \rightarrow (x,y)$ transform on the image plane.

Figure 9 – Calibration Ruler Image A

Figure 10 – Calibration Ruler Image B

Define a point $p^* = (r^*, c^*)$ that we arbitrarily choose to be the center of an (x,y) coordinate system. Note that the *X,Y* coordinates in Figure 9 and Figure 10 refer to the transpose of Matlab's row/column plane, (e.g, in the images $X = c, Y = r$).

The necessary transformations are:

$$p_{xy} = \begin{bmatrix} R_C & 0 \\ 0 & -R_R \end{bmatrix} (p_{rc} - p^*) = T_{rc}^{xy} p_{rc} \tag{1}$$

Where R_C (m/column-pixels) and R_R (m/row-pixels) are scaling constants determined by the camera and image size. We call this affine transform T_{rc}^{xy} . The inverse transformation is easy to calculate given equation (1). To calculate the camera scaling constants, we use pairs of data points from images A and B, and the ruler distances $D_{1,2}$ as measured (20cm and 29cm for images A and B, respectively). Choosing $p_{rc}^* = (0,0)$ for both images, we use:

$$D_{1,2}^2 = \left\| p_{1,xy} - p_{2,xy} \right\|_2^2 = \left\| T_{rc}^{xy} p_{1,rc} - T_{rc}^{xy} p_{2,rc} \right\|_2^2 \tag{2}$$

e.g., for image A,

$$\begin{aligned} .2 &= R_C^2 (c_1 - c_2)^2 + R_R^2 (r_2 - r_2)^2 \\ &= R_C^2 (1966 - 1940)^2 + R_R^2 (1301 - 415)^2 \end{aligned} \tag{3}$$

A similar calculation for image B provides the second equation. Solving these two equations for $R_C, R_R > 0$, we finally get:

$$R_R = 0.0002256436081 \ \ m/row-px = .2256\, mm/row-px$$

$$R_C = 0.0002172274123 \ \ m/col-px = .2173\, mm/col-px$$

As the rectified images should scale perpendicular distances in equal measure, we expect that $R_C = R_R$. Further, we see a very small difference from our calculations, specifically $\Delta R = R_C - R_R = 8.3\mu m/px$.

Image simplification

After image preprocessing, the data set consists of a series of cropped and rectified images that contain a single green background against which the fasteners are in relief. From this starting point, the image needs to be simplified so that the objects of interest can be extracted later.

Figure 11 – Pre-processed Image

We first decided on a color space in which to do our processing. A number of options were available, the three most feasible of these were: to work in any single RGB layer, or to work in a combination of the RGB layers, or in an HSV color space. We first tried working in the red and blue layers of the RGB image, as the background is green and should not be visible in these layers. However, as can be seen from a high contrast image of the background in Figure 12, the diffuse background actually contains a significant amount of red and blue. This fact can also be seen in the red and blue layer subplots in Figure 13.

Figure 12 - Background High Contrast

We instead chose to work in the HSV space. As shown in Figure 13, the hue layer does not provide enough contrast. The saturation layer has too much noise in the background and similarly lacks contrast. The value layer, on the other hand, provides high contrast between background and subject, very little background noise, and is somewhat resistant to non-uniform lighting. From this point forward all processing was done in the value layer.

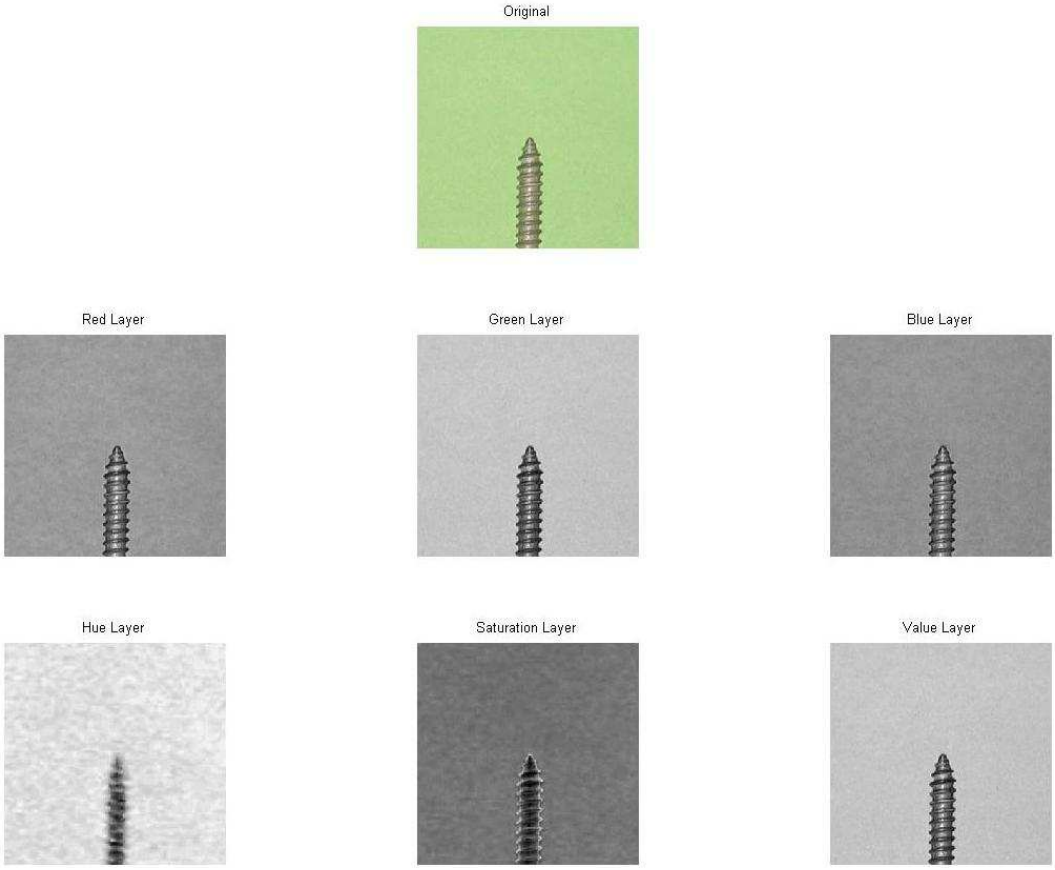

Figure 13 - Layers

To begin simplification, we first subtract the background by setting high and low thresholds that encapsulate the range of values in the background. We then map any pixels above or below these thresholds into a binary image with background pixels in black and the remaining pixels in white.

The calculation of these thresholds is based on two images. First, define σ_{BG} as the standard deviation of the value layer of an image of the green background alone. Then, define M_I as the most common intensity of the value layer of our image (the mode of the binned intensities in our current image). Finally, let the value layer of the fastener image be $I_V(x, y)$. The high and low thresholds are then calculated as:

$$T_{high} = M_I + 3.5\sigma_{BG} \tag{4}$$

$$T_{low} = M_I - 3.5\sigma_{BG} \tag{5}$$

Finally, the new binary image is calculated as:

$$I_{thresh}(x, y) = \begin{cases} 1 & I_V(x, y) < T_{low} \vee I_V(x, y) > T_{high} \\ 0 & o.w. \end{cases} \tag{6}$$

This method works fairly well, as two linear thresholds on the histogram of $I_V(x, y)$ allow us to break up the values into three regions: dark fastener, green background, and white reflection from the fastener.

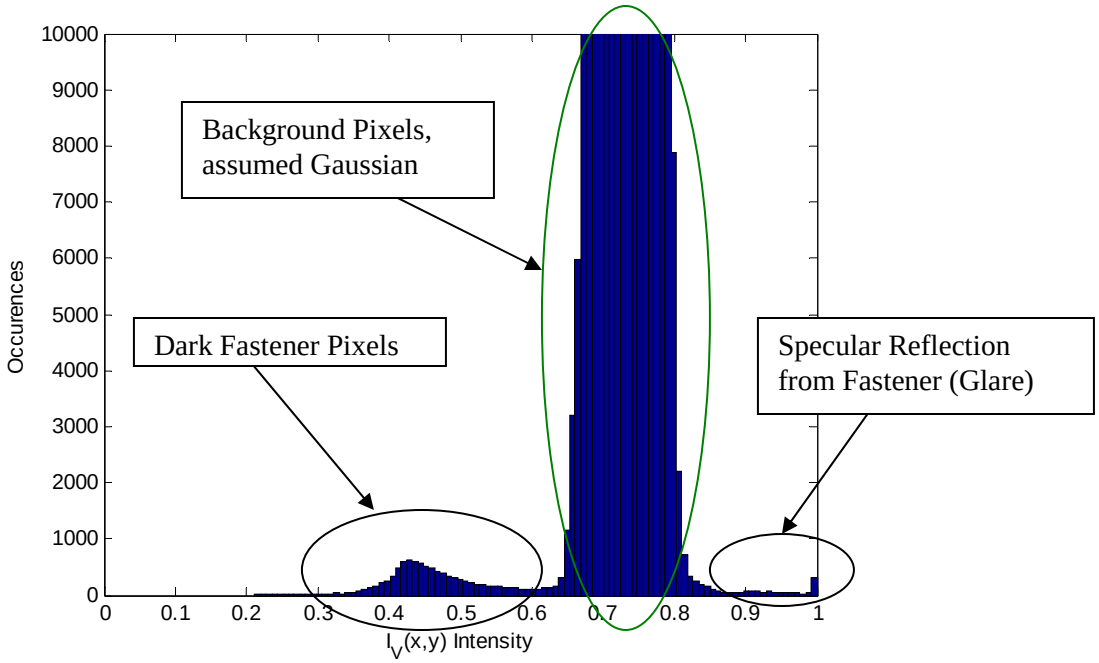

Figure 14 – Histogram of the value layer a fastener frontal view

$I_{thresh}(x, y)$ is shown in Figure 15. Note the background has been removed, the remaining image is a binary image (mapping *r*,*c* to values of True or False).

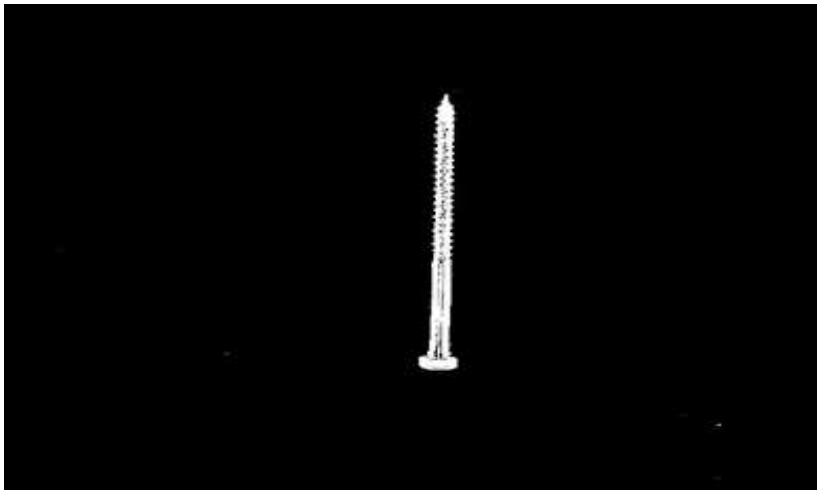

Figure 15 – Threshold Subtracted Image

After this simplification, the fastener object stands out clearly from the background. The fastener is not solid, and there is a moderate amount of scattered noise in the image. Two standard Matlab morphological operations can be used to rectify this situation. A majority operation is performed to eliminate most of the scattered noise, much of which is composed of single pixels. This reduces the amount of processing that needs to be performed later to differentiate the correct object from the noise. After this majority filter, five dilations fill in the area surrounding the fastener into a larger blob. This blob, which fully overlays the fastener, will be used in later processing.

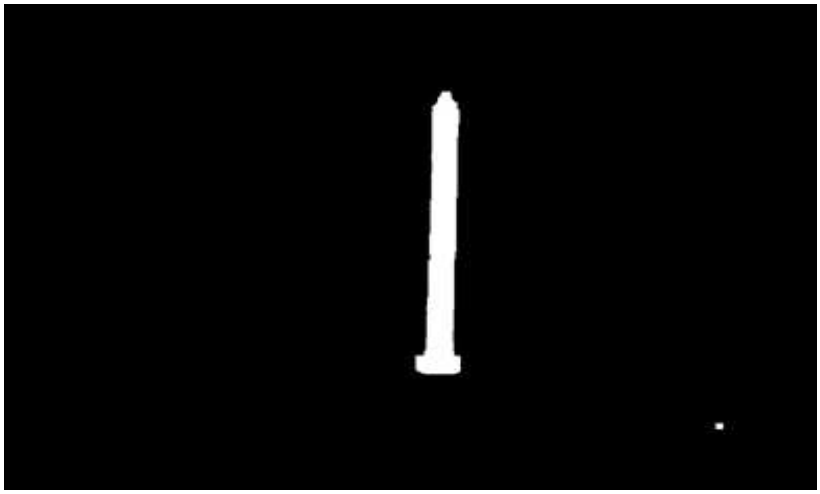

Figure 16 – Simplified Image

Object Selection

After the simplification stage, the binary image contains only a few objects: a number of small "noise blobs" and a number of large blobs representing the fasteners in the original image. The next problem is the selection of desired fastener objects from among the noise.

The first step in object selection is the use of the Matlab *bwlabel* function to return a set of labels that group contiguous pixels into objects. We use the default bwlabel parameters, which group any pixel on an adjacent diagonal, row, or column. The result is an added layer to the image with an integer associated with each pixel and describes the object of which the pixel is a part.

Once we have this labeled image, we use the Matlab *regionprops* function to calculate the area, centroid and bounding box properties of each labeled blob. Suppose I_L is the labeled image, we define

$$regs = regionprops(I_L, 'Area', 'Centroid', 'BoundingBox'); \quad (7)$$

where regs is a list structure containing the returned values for object areas, centroids, and bounding boxes. We sort each blob in order of decreasing area. Random noise blobs only make up a small portion of the total area and are dwarfed by the much larger objects of interest. Screening out objects with small areas eliminates the remaining noise, leaving only the fastener objects. All objects that comprise less than 1% of the total image area are ignored.

After this screening process, we can find the bounding boxes of the remaining objects from their BoundingBox property. It is important that our objects always be presented in the same manner, so a standard orientation must be enforced. We chose the following as the standard orientation for a frontal view: the long axis of the fastener is along the *y* axis, the short axis is along the *x*, and the fastener head is at the top. Our orientation algorithm calculates ϕ, the angular offset between the standard coordinate axis (x, y) and the true coordinate axis of the fastener in the image (x', y'). Thus, if we rotate the image of the fastener by $-\phi$, we align it with our desired axes. We provide a detailed explanation of our *orient* algorithm in Appendix A.

Feature extraction

Once the fastener has been selected and rotated, the next step is to extract its features. Any number of features of the fastener can be measured, however the main features that uniquely describe a fastener are the length of the fastener, the width, the head type, and the pitch. Our distance metric makes heavy use of these 4 pieces of data as a core set of features. We use a variety of algorithms to extract all the necessary components from the image.

First the contour is extracted from the image, and transformed into the metric (*x*,*y*) plane. We can then reparametrize the left and right sides of the contour to simplify many of the required calculations. Specifically, we reparametrize <*x(t),y(t)*> into $x_L(y)$ and $x_R(y)$, the left and right contours respectively. The side contours are now functions of height.

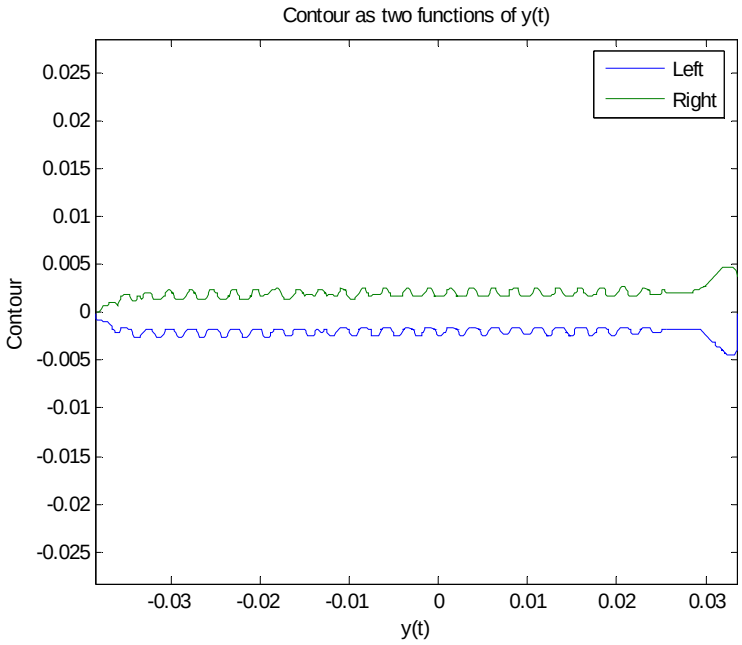

Figure 17 – Fastener contour as $x_L(y)$ and $x_R(y)$

Length and Width

First, we examine the methods used to determine the length and diameter (width) of the fastener. Length can be determined by simply examining the regionprops of the image. We can extract the head width from the contour using the formula $w_{\max} = |x_L(y) - x_R(y)|$. We extract the median width by averaging the medians on the left and right sides: $w_{med} = 1/2\left(med\,|x_L(y)| + med\,|x_R(y)|\right)$. This value most accurately describes the diameter of the shaft.

Threading Count

One of the most important properties of a fastener is whether it has threads or not, and if it does, what the pitch is. After we have simplified our images and extracted a contour, threads are clearly visible. We can count these threads and determine such details as thread count, thread density, thread width, and thread height.

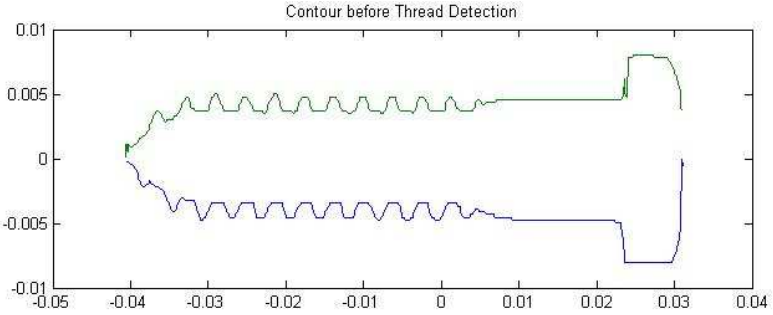

Figure 18 - Contour

The original contour, consisting of two functions describing the right (green) and left (blue) sides can be seen in Figure 18. The threads are clearly visible. The first step in extracting this information is to take the derivative of this contour with respect to *y*. Appendix B contains a formula for calculating the finite differences of a function given nonuniform spacing, and corresponding error analysis. For clarity, we will only display half of the contour throughout the rest of this explanation, the left half (green above). The same operations are performed on both halves and the results are nearly mirrors of each other. The derivative is shown in Figure 19.

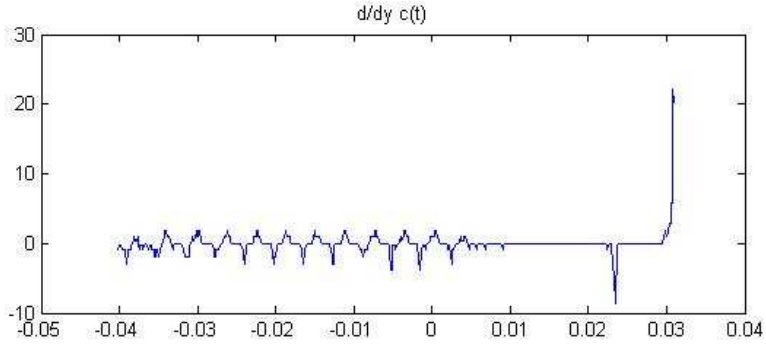

Figure 19 - Derivative of $x_L(y)$

We extract the roots of this derivative; these describe local maxima and minima. The next step is to find the curvature (second derivative) of $x_L(y)$. This is accomplished through our *findcurv2* function. We then match all the zeros of the derivative with their corresponding curvature. Zeros with a positive curvature are local minima, while those with negative curvature are local maxima. Maxima and minima are then filtered so that only those within one standard deviation of the mean *x* position of the contour are included. This excludes maxima and minima near the tip and head since they will be near *x=0* or above $x = \mu + \sigma$ and includes those which comprise the threads. What are left are the points at the top and bottom of each thread. These are plotted in Figure 20 on the original contour, 'x's are maximas and 'o's are minima.

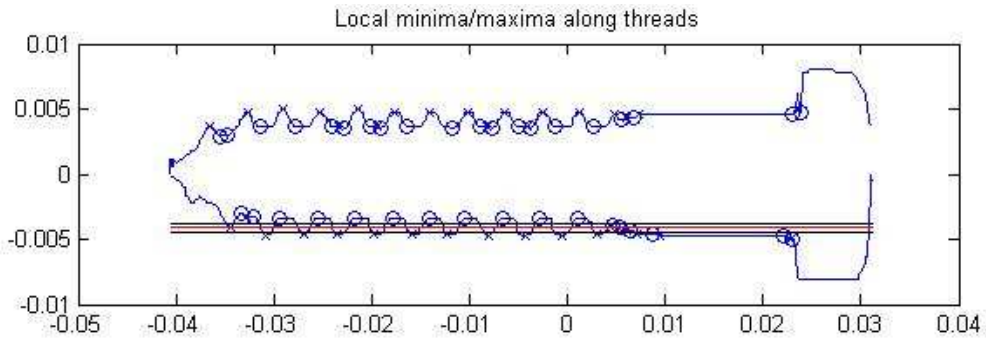

Figure 20 - Minima and Maxima of Contour

From the position of the minima and maxima it is trivial to determine thread width, thread height, and the inner and outer diameter of the fastener.

Although one could simply count the number of minima and maxim to determine a total thread count for the fastener, this is likely to be inaccurate as some extraneous points are likely to be included. The number of median crossings provides a more accurate count. Define M_{ed} as the median x value of the left half of the contour. Counting the number of times that $x_L(y)$ crosses M_{ed} gives an accurate account of the number of complete threads in the fastener. This is illustrated in Figure 21.

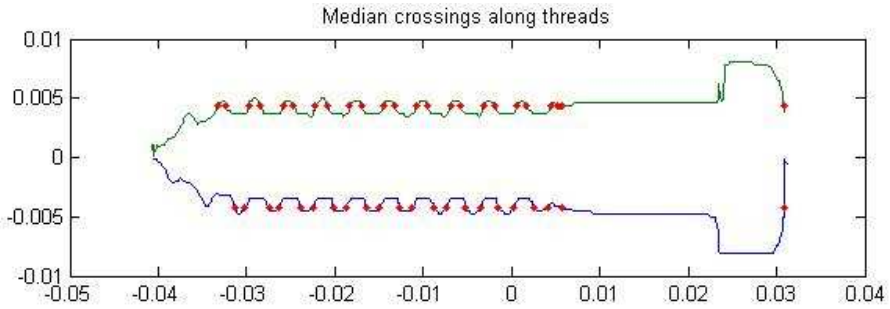

Figure 21 - Median Crossing Thread Count

Thread density, which we define as the total number of threads divided by the total length of the fastener, can give us an idea of the coarseness of the threading and how much of the shaft is threaded. This is a useful measure to add to our feature vector.

A systems view of the algorithm for determining all of the features discussed in this section is shown in Figure 22.

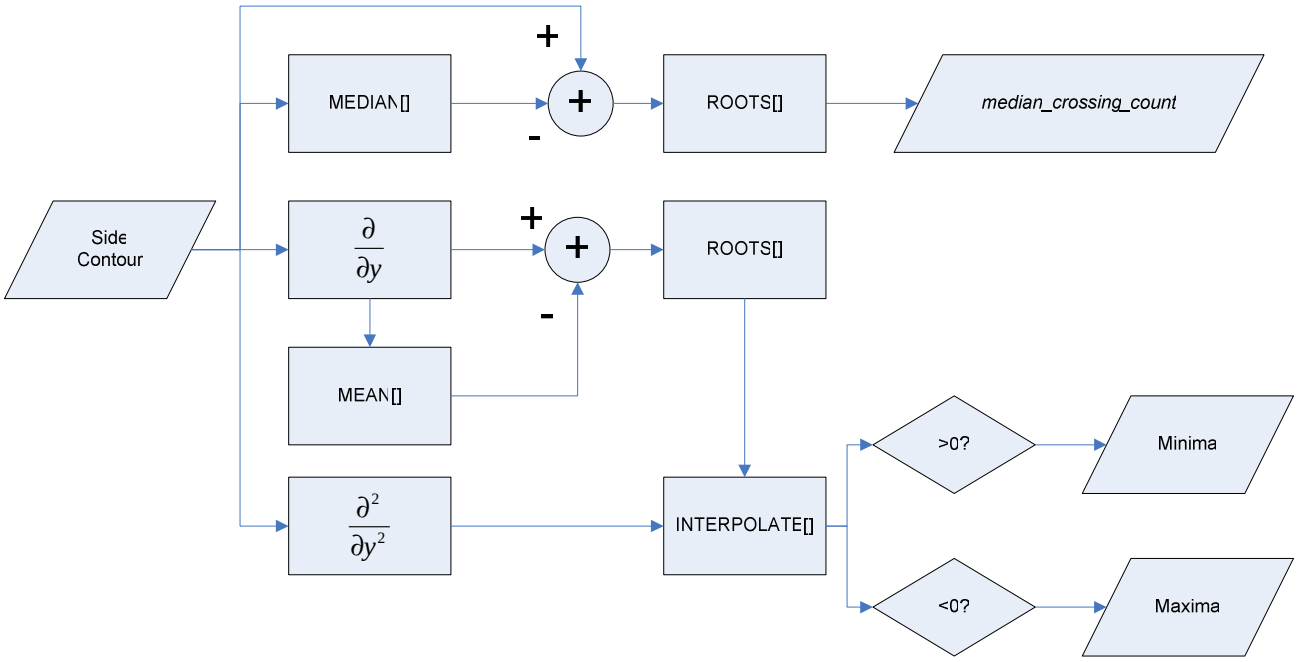

Figure 22 - Thread Feature Extraction

Tip Angle

To determine the tip angle we need only to know the endpoint of the contour and a single point that is within the unthreaded, angled part of the fastener. The endpoint is $x(y_{\min})$ and the first 5% of the contour points are in the set $Y_s = [y_{\min}, y_{5\%})$. We take the mean of *x* over Y_s, μ_s. We then find the first point (x_s, y_s) where *x* touches μ_s. The tip angle can then be calculated as:

$$\theta_{tip} = \tan^{-1}\left(\frac{x_s}{y_s - y_{\min}}\right) \tag{8}$$

An example is shown below.

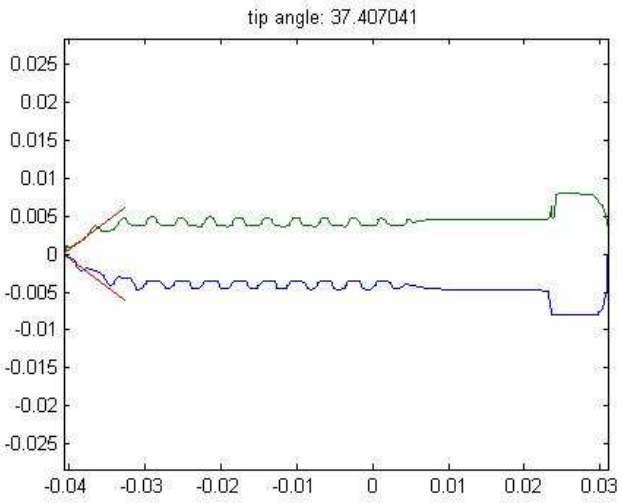

Figure 23 - Tip Angle

Fastener Head Type

An important part of the description of any fastener is its head type. During data collection we captured separate images of the head of each fastener. From these images we can extract information about the number of sides the head has. At first we planned to extract the drive type as well (phillips vs. standard) and other features. After several attempts to extract information about the drive type, we realized that the head view has a very low SNR. In general, images of fastener heads were on the order of around 35x35 pixels and had large amounts of spatially variant noise (e.g. salt & pepper). Applying corrective filters (Median, Wiener) removed noise, but also made discerning the drive type impossible.

One method we did not have time to test, but which we hope may be resilient to such noise, is provided in Appendix C.

Fastener Head Side Count

We had more success removing noise from the contour of the head. Our algorithm calculates the contour, corrects for noise, approximates the number of sides, and calculates a confidence value.

Let *T* denote the discrete transform from (*R*,*C*) space to (r, θ) space, and let *I* denote an image of the head after subtracting the background mean, squaring the result, and setting background pixels to zero. Note that we often have some error when subtracting the background.

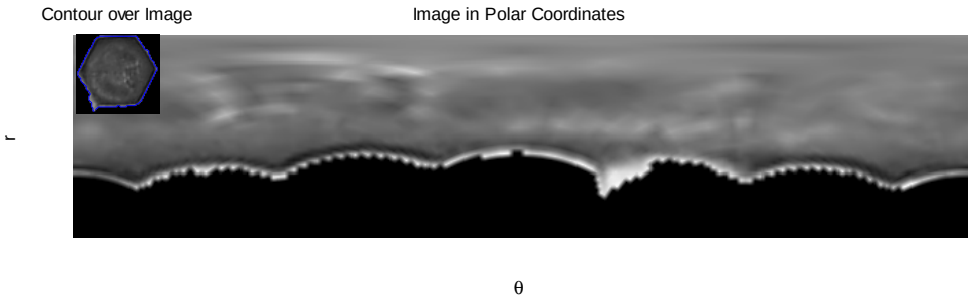

Figure 24 – Fastener head in *(R,C)* space and in (*r*, θ) space

After calculating the magnitude of the gradient of this image and thresholding to remove insignificant changes in slope, we can find an outer contour line for the fastener.

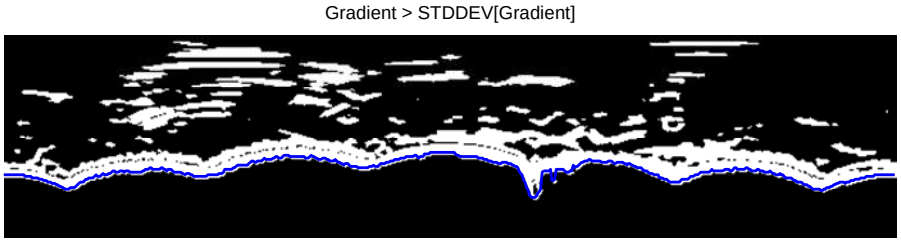

Figure 25 – Thresholded image with superimposed contour line

Note that this contour can be modeled in the following manner:

$$C(u) = c(u) + \mathbf{N}(u) + d(u) + \mathbf{U}(u) \tag{9}$$

Here $u = k\theta$, $\theta \in [0, 2\pi)$, and $u \in [0, U)$. *C(u)* is simply a function of a scaled angle. $\mathbf{N}(u)$ is a normally distributed random vector. *d(u)* is an offset that depends on the error between our calculated centroid and the true centroid; this value can be modeled as a slowly varying sinusoid (e.g., it runs through one period on the domain of θ). $\mathbf{U}(u)$ is a noise vector that we cannot model; we must assume that any non Gaussian noise is low in magnitude.

To account for $\mathbf{N}(u)$, we smooth *C(u)* with a Gaussian filter G_σ, with $\sigma = \frac{U}{30}$; the main support of this filter is approximately 10% of *U*. To decrease the effects of *d(u)* without explicitly calculating its parameters, we introduce what we call localized means:

$$\mu_L(C;u) = \frac{1}{L}\left[\sum_{i=u-L/2}^{u+L/2} C(i)\right] \tag{10}$$

If we choose *L=.2U*, we can say that $\mu_L(u) \simeq d(u) + \mu$, where μ is the mean of *C(u)* over its entire domain. Now let $\tilde{C}(u) = G_\sigma C(u)$ and $\tilde{c}(u) = \tilde{C}(u) - \mu_L(\tilde{C};u)$. Then $\tilde{c}(u)$ approximates the true head outline with its mean at *0*: $\tilde{c}(u) \simeq c(u) - \mu$.

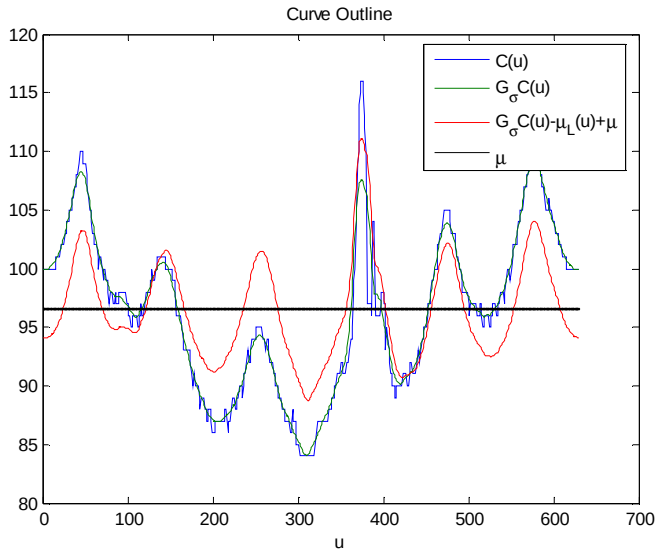

Figure 26 – Steps to approximating the true head contour

Critical points of $\tilde{c}(u)$ do not allow for an accurate side count because $\mathbf{U}(u)$ is nonzero and the derivative operator increases the error incurred by this noise: more minima and maxima appear than really exist in *c(u)*. The roots of $\tilde{c}(u)$, on the other hand, provide a very accurate count for multi-faceted heads (e.g., hexagonal types). In these cases, *R*, the root count, follows the relationship: *R=2f*, where *f* is the head facet count.

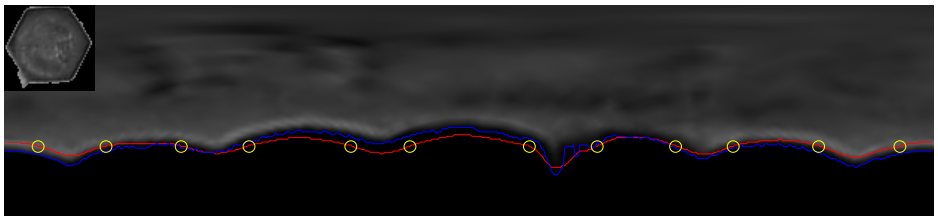

Figure 27 – There are 2 Roots (zero crossings) for each head facet

Unfortunately, circular heads also exhibit roots in $\tilde{c}(u)$; especially when there is a great deal of noise in the preprocessed contour *C(u)*. This noise also raises the standard deviation of *C(u)*, so we cannot discern between circular and faceted heads by comparing their standard deviations. Instead, we calculate the facet count of the head using a second method, and compare the two values. If they are very close (e.g., within 1 of each other), then we generate a high confidence value. If they are not close, then we generate a low confidence value and assume that the head is circular.

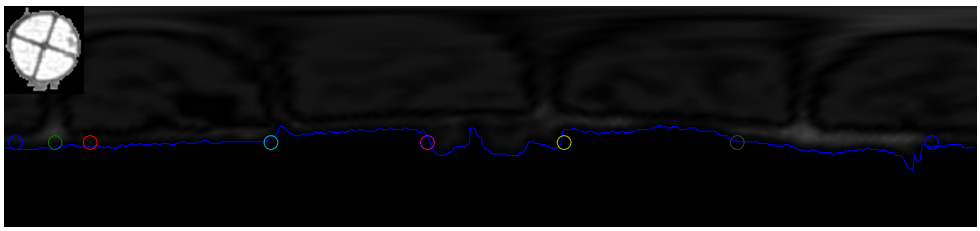

Figure 28 – Circular head with noise, the algorithm detects 4 sides

A second way to approximate the facet count is to generate a model of what the contour should look like if it has a specific number of facets, and to correlate this model with the original contour after noise

reduction. We can also generate this model and optimize for this correlation as a function of the facet count. We used a simple sinusoidal model for the contour:

$$m(f,u) = \cos\left(\frac{f}{2}\frac{2\pi u}{N}\right) \tag{11}$$

where f is the number of facets in the contour.

Let the cross-correlation between two contours be defined as $\kappa_m[C_1, C_2] = \sum_u C_1(u+m)C_2^*(u)$, and the cross-correlation squared norm be defined as $K[C_1, C_2] = \sum_m \kappa_m[C_1, C_2]^2$. Then the optimization problem we solved was:

$$f_\Xi = \arg \max_f \{K[m(f,u), \tilde{c}(u)]\} \quad s.t. \quad f \geq 0 \tag{12}$$

If this f_Ξ, the optimal facet count, is within 1 of the f count from our first algorithm, then the confidence in our original count is high. We chose the following formula for calculating the confidence:

$$\chi_f = e^{-(f_\Xi - f)^2} \tag{13}$$

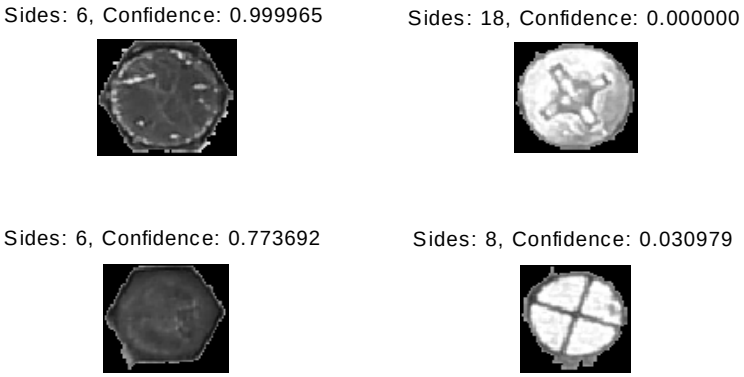

Figure 29 – Side count algorithm and associated confidence values

Feature Vector Creation

Our data analysis, verification, tolerance, and classification analysis is aided by the use of an algorithm we called *extract_data*, which takes as input a text file with a list of *(front,head)* JPG file pairs, and outputs a large data vector FASTENER_DATA. Figure 30 shows an example input text file:

```
front_016.jpg head_016.jpg
front_017.jpg head_017.jpg
front_018.jpg head_018.jpg
front_019.jpg NULL
front_020.jpg NULL
front_021.jpg head_021.jpg
```

Figure 30 – Sample *process_both* text file, provides *(front,head)* image pairs for *extract_data*

Note that in some cases, such as with multiple fasteners, we did not have images of the head view available. *extract_data* takes all these factors into account. Below is a pseudo-code implementation of this high-level function.

```
ALGORITHM extract_data
INPUT:
      D: Directory with cropped & rotated images
      L: List of 2-tuples: (front,head) image pairs
OUTPUT:
      FASTENER_DATA
If (exists FASTENER_DATA)
      Operations will append to current FASTENER_DATA list
Else
      Create new FASTENER_DATA list (length=0)


n=L.length()
state=process_init(background image, base parameters)
for i=1,2,…,n
      fasteners=front_image_process(L{i}.front,state)
      k=fasteners.length()

      if (k>1 AND L{i}.head NOT NULL)
            ERROR: Found multiple fasteners but head view was provided
      else if (k>1)
            for j=1,2,…,k
                  front_features=front_features(fastners{k})
                  head_features=NaN (Not available)
                  FASTENER_DATA.append(front_features,head_features)
            end
      else (k==1)
            front_features=front_features(fastneners{1})
            head_process=head_image_process(L{i}.head,state)
            head_features=head_features(head_process)
            FASTENER_DATA.append(front_features,head_features)
      end if
end for
```

Note that a number of other items are appended to the FASTENER_DATA list object, including the front view images after processing and the filename of the front view. An overview and a set of diagrams detailing the structure of FASTENER_DATA are available in Appendix D.

Distance Calculation

Perhaps the most crucial part of the project is the distance calculation. The feature space of this project is a mix of both discrete variables such as head type and continuous variables such as length or thread density. As such, we have defined a nonlinear metric within the feature space. We designed this metric to separate fasteners of the three main categories by large values. Fasteners of the same type will in general have small distances between them.

In addition, our metric and feature space need to satisfy the following conditions:

1. $D(x, y) \geq 0 \Leftrightarrow x \neq y$
2. $D(x, y) = 0 \Leftrightarrow x = y$
3. $D(x, y) = D(y, x)$
4. $D(x, y) + D(y, z) \geq D(x, z)$

The first and second conditions specify that two items should have a similarity distance of zero if and only if their feature vectors share the same values. In plain English, two fasteners will have a distance of zero only if they are in fact the same object. The third restriction specifies that the distance metric is commutative: the order of input cannot affect the distance computation.

The fourth restriction, also called the triangle inequality, states that if feature vector x is different from feature vectors y and z, then the distance between x and z is at least the sum of the distances between x and y, y and z. In other words, feature vector y cannot somehow make the distance between x and z shorter than a "straight line" distance.

Distance Metric

To derive the distance metric, we first sought to understand what features clearly segregated our three classes of fasteners. Upon consideration, we found the following three sets of major differences that should be weighted heavily to provide separation between classes:

Wood Screws vs. Machine Screws

Wood screws have sharp tips, machine screws have blunt tips.
Wood screws have coarse threading, machine screws have fine threading
Wood screws have a low thread density, machine screws have a high thread density
Wood screws are mostly circular headed, machine screws are mostly hex headed.

Wood Screws vs. Nails

Wood screws have threading, nails do not have threads.
Wood screws tend to be thicker than nails

Machine Screws vs. Nails

Machine screws have threading, nails do not have threading.
Machine screws are blunt tipped, nails have sharp tips.
Machine screws tend to be thicker than nails.

Within each group differences such as length, width, and thread count differentiate between individual fasteners, but should be weighted much lower to ensure two fasteners of the same class do not have a high difference value. This information is summarized in Figure 31.

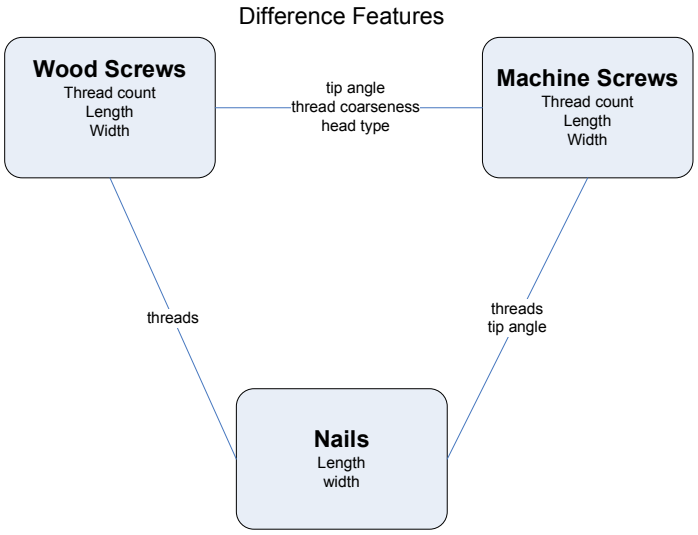

Figure 31 - Major and Minor Differences

Given these sets of differences, we decided to apply the following weights to each feature:

Threading (std of width)	15%
Tip Angle	34.5%
Head Type	10%
Threading Density	23%
Length	10%
Width (at outside of threads)	7.5%

The metric for the distance calculated from each feature is either a percent difference or an inverse exponential term. A linear combination of these two types of terms constrains the output to a range between zero and one and allows us to emphasize subtle variations in certain features. The full distance equation is shown below.

We use the following definitions:

σ_w	standard deviation of the width
θ_{tip}	tip angle
Σ	number of sides of the head
L	length
Δ	thread distribution ($\Delta = n/L$ where n is the thread count)
w_M	outer width

We then define the functions:

$$D_1(x_1, x_2) = \frac{|x_1 - x_2|}{\max(x_1, x_2)}$$

$$D_2(x_1, x_2) = 1 - e^{-2\frac{|x_1 - x_2|}{\min(x_1, x_2)}}$$

$$D_3(x_1, x_2) = 1 - e^{-2\frac{|x_1 - x_2|}{\max(x_1, x_2)}}$$

Then the distance $D(f_1, f_2)$ between two fasteners is:

$$\begin{aligned} D = {} & .15D_2(\sigma_{w,1}, \sigma_{w,2}) + .345D_1(\theta_{tip,1}, \theta_{tip,2}) \\ & + .23D_2(\Delta_1, \Delta_2) + .10(\Sigma_1 != \Sigma_2) \\ & + .10D_3(L_1, L_2) + .075D_1(w_{M,1}, w_{M,2}) \end{aligned} \tag{14}$$

This metric works quite well. Some examples of calculated distances are shown below.

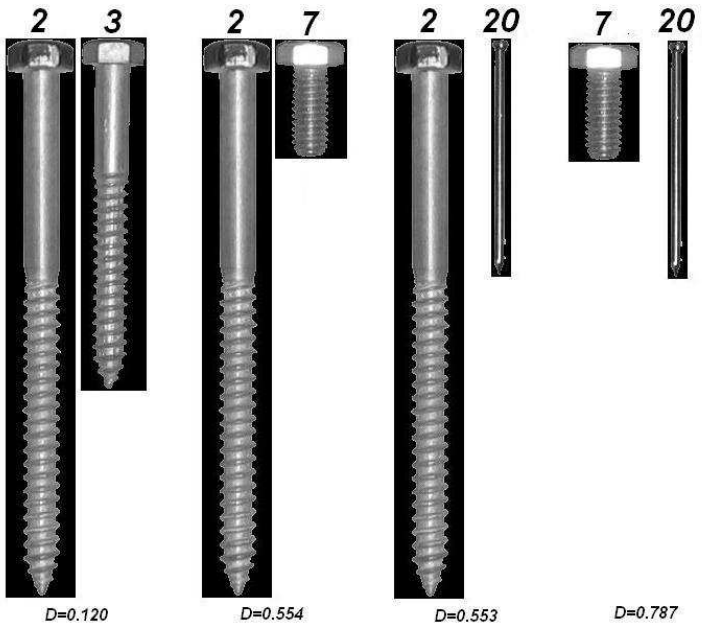

Figure 32 - Distance Examples

As can be seen in Figure 32, all of the values range between zero and one; this is a mathematic property of our distance formula, since it is a weighted average of deltas that are themselves restricted between zero and one. Objects that are similar, such as the first pair in the figure above, yield low distance values, while very dissimilar objects, such as the last pair, have a large distance. When run upon the full range of our first test set, it is clear from Figure 33 that the values produced run nearly the full allowable range. With more fasteners of different types in our other data sets we are able to get values approaching one. Values of zero are produced when comparing a fastener to itself, as can be seen from the diagonal of the matrix.

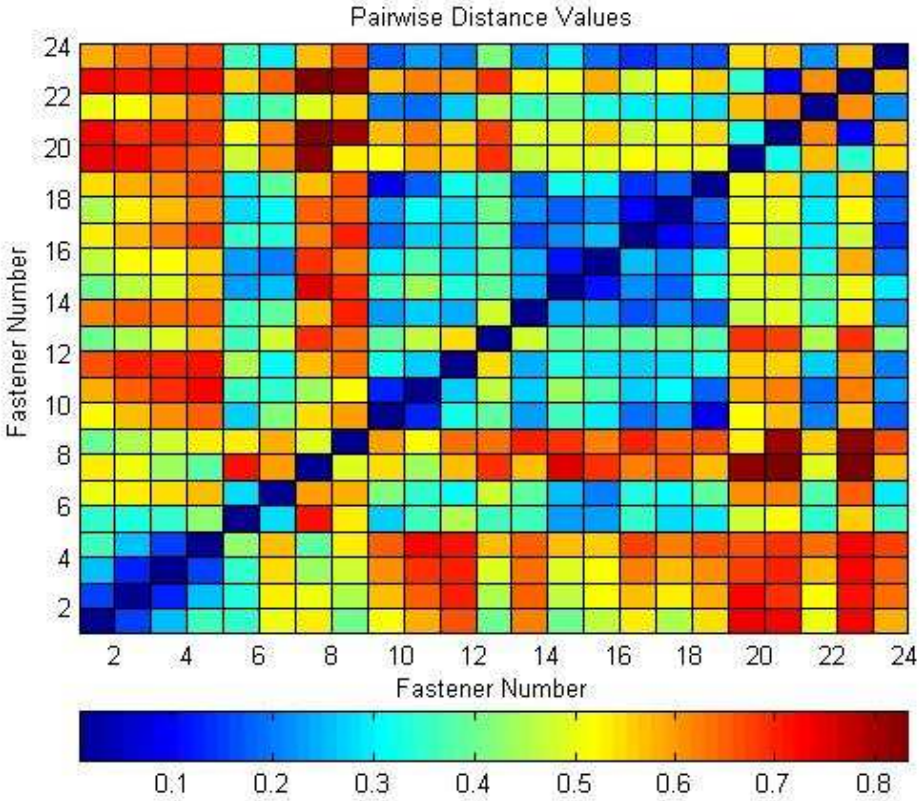

Figure 33 - Distance Values Matrix

Fastener classification

We implemented and successfully validated a multi-classification scheme for differentiating between 4 types of fasteners (*wood*, *machine*, *nail*, and *other*) using Support Vector Machines and One vs. One voting methods. This section is composed of an introduction to the theory behind SVM and One vs. One multi-classification, a subsection describing the details of our implementation, and finally our analysis and validation study.

Introduction

Support Vector Machines, and Supervised Learning Machines in general, work in two stages: training (learning) and classification.

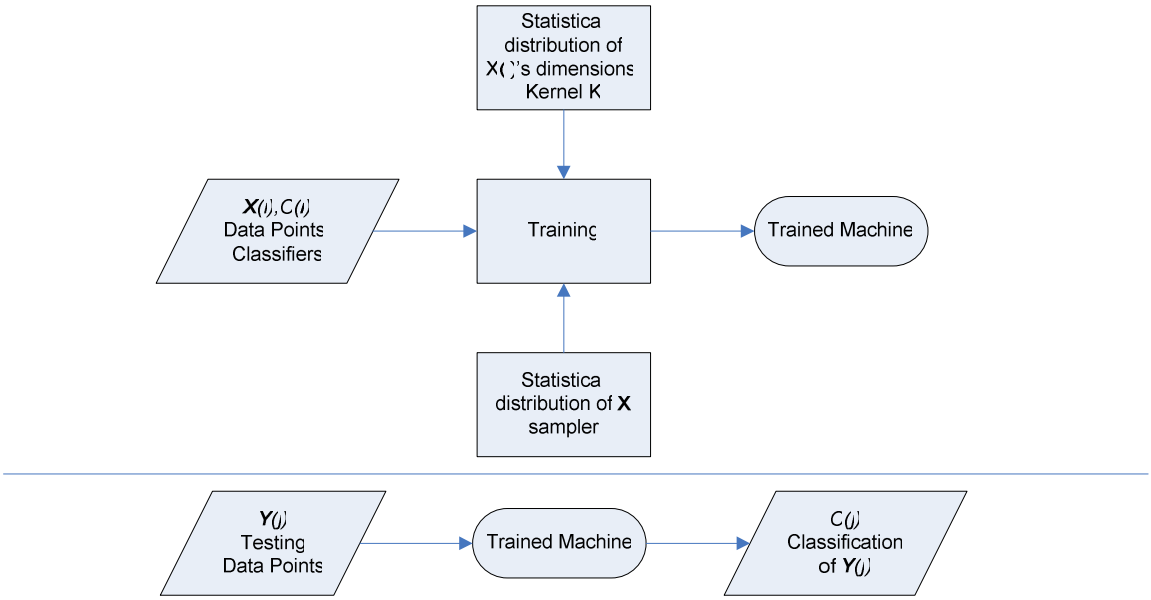

Figure 34 – Learning Machines in Two Stages

Support Vector Machines are a standard method for classifying a variety of data sets, and have recently become very popular in machine learning and classification due to their inherent ease of use and flexibility in dealing with nonlinear data separation.

The basic premise of Support Vector Machines in very basic terms is provided below. We are given a set of possible functions that take data points $\mathbf{x}^{(i)}$ that belong to one of two classes $c_i \in \{-1,+1\}$ (true or false, red or blue, tree or not tree, etc). We are also given an initial training set of points and classification values. We would like to choose the function which gives us the minimum expected "empirical risk"[6] in classifying not only the current points, but also points that will come out of the future data pool. We can define this risk as:

$$R_{emp} = \frac{1}{2l}\sum_{i=1}^{l} | c_i - f(\mathbf{x}^{(i)}) | \qquad (15)$$

If we know that this set of functions varies with some set of parameters α, then we can redefine our risk minimization step as:

$$\alpha_{\min} = \arg\min_{\alpha} \left\{ R_{emp}(\alpha) = \frac{1}{2l}\sum_{i=1}^{l} | y_i - f(\mathbf{x}^{(i)};\alpha) | \right\} \qquad (16)$$

While we usually only have a limited set of $\mathbf{x}^{(i)}$ points, they generally represent a sample of data from an infinite space. Our risk function, even our *separation function and separation parameters* $f(\mathbf{x};\alpha)$ and α, respectfully, depend on the type of data we are looking at, and must represent this infinite space accordingly.

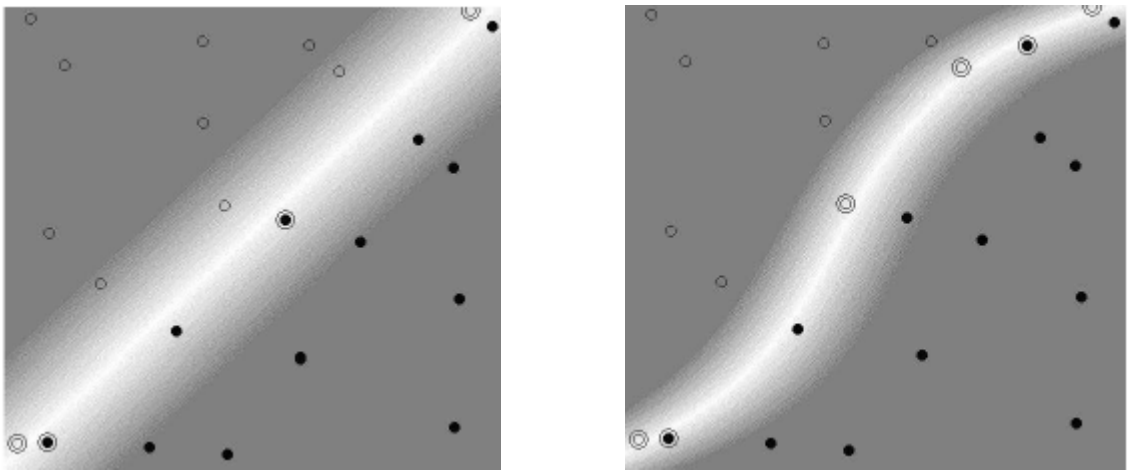

Figure 35 (a) and (b), Gradients of functions in Linear (a) and Nonlinear (b) SVM Classification

Figure 35 (a) shows a set of classification points in 2D. White and black stand for one of the two classifications. Note that the data is linearly separable: a line can be drawn between the two sets without any errors and a simple linear function can be generated based on that line. Points on one side can be classified as white, and on the other side as black. This classification is formally done by taking $sign[f(\mathbf{x})]$ (-1 implies the subspace belongs to first class, 1 implies the other).

Linear SVM classification with *n* dimensional data separates sets of points using a hyperplane. The function $f(\mathbf{x}^{(i)};\alpha,\beta)$ depends on *n* parameters in α and an offset parameter β. More specifically, it has been found these optimal functions have the following form [7]:

$$f(\mathbf{x};\alpha,\beta) = \langle w, x\rangle + \beta = \sum_{j=1}^{n} \alpha_j c_j \langle \mathbf{x}^{(j)}, \mathbf{x}\rangle + \beta \qquad (17)$$

Where the $\mathbf{x}^{(j)}$'s are taken from the training data and $\mathbf{x}$ is either in the training data or from some testing data set. The α and β values can be solved for (optimized for) with a training data set.

Nonlinear SVM classification can be "reduced" to linear SVM using Kernel methods. A properly defined mapping function $\phi(\mathbf{x})$ can "untwist" nonlinear data points into a linearly separable feature space. Both $\phi(\mathbf{x})$ and the related kernel $K(\mathbf{x}^{(i)},\mathbf{x}^{(j)})$ are generally approximated using intrinsic knowledge about and a physical interpretation of the data points. A graphical example of the underlying principles is shown in Figure 36.

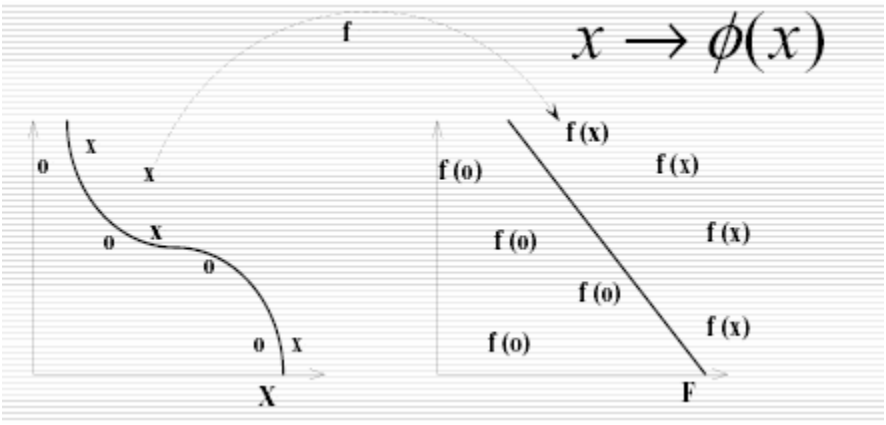

Figure 36 – Kernel Methods Map Inseparable Data into a Simpler Form

We found, after comparing classification performance with various kernels, that simple linear SVM on a 14 dimensional feature space (height, width, threading, etc…) provided the lowest error during data cross-validation. Specifically, we compared Linear, RBF (Gaussian), and Polynomial kernels (but we did not attempt any custom nonlinear functions). These Linear kernels are defined as the simple inner product $K(\mathbf{x}^1, \mathbf{x}^2) = \left\langle \mathbf{x}^1, \mathbf{x}^2 \right\rangle_2$, and the mapping function is simply $\phi(\mathbf{x}) = \mathbf{x}$.

Support Vector Machines have the limitation that they are designed to generate a function that discriminates between only two classes. We must discriminate among four different classes, and for this we tested several well known classification schemes, including One vs. One, One vs. Rest, and Multi-Class SVM. Once again, we chose the method that provided the best results during cross-validation: One vs. One classification [8].

One vs. One (1-vs-1) classification, proposed by Hastie and Tibshirani as Pairwise Coupling [9], generates a set of binary SVM classification problems out of an SVM multi-classification problem. Given *m* classes, (1-vs-1) constructs $Cm = \binom{m}{2}$ SVM machines and builds *Cm* boundaries within the feature space; in our case these boundaries are linear.

Implementation

We implemented our classification method using the SPIDER object-oriented machine learning library, as developed at the Max Planck Institute for Biological Cybernetics [10]. SPIDER allowed us to very quickly test a variety of kernels, classification algorithms, and multi-classification extensions on our data. Furthermore, we used SPIDER to validate our results and to calculate the approximate error incurred in our classification.

Our full dataset consisted of 73 pre-processed fasteners in a FASTENER_DATA object. As a number of these did not have a *head* view, our data vectors consisted of the 14 features extracted from the front view. Further, our feature extraction algorithms failed to extract one specific feature for 6 of the fasteners, and returned a *NaN* value. We chose to set all *NaN* values to *0*. These 6 fasteners were affected. These values did not affect the classification results, as we found during the validation study.

We grouped fasteners into one of four classes by hand: *wood*, *machine*, *nail*, and *other*. We had 29 fasteners of class *wood*, 27 fasteners of class *machine*, and 16 fasteners of class *nail*. Note that only one fastener was classified as *other*, as seen in Figure 37.

Figure 37 – Fastener 25, the only fastener classified as *other*

The SPIDER classification system takes in training data in 2-matrix-tuple form. The first matrix in has the dimensions *NxP*, and consists of *N* rows of *P*-dimensional feature vectors. The second matrix has the dimensions *NxM*, with *M*-dimensional rows that denote to which class the *N* feature vectors belong. A row-set from the 2-tuple might look like the following:

T_1 : `0.1088 0.0124 0.0056 0.0008 8.7838 19.2567 24.0000 0.0051 0.0062 0.0057 0.0013 19.2083 24.7755 220.6689`

T_2 : `1 -1 -1 -1`

Note that T_2 for this row implies that the fastener is of type 1, a wood screw. Values of T_1 are the fastener *height, head width, width median, …* in that order.

Validation and Cross-Validation

We ran 4 validation studies. The first 2 classification studies were taken on all 73 fasteners with *NaN* values set to *0*. The last 2 studies were taken on the 67 fasteners that did not have *NaN* values in their feature vectors. The two types of validation studies we ran are: *K-Fold* cross validation with *K=10* folds repeated *10* times on randomly chosen subsets of the data, and *Leave One Out* validation.

K-Fold Cross Validation is a method for deriving an expected error calculation when you have a training set for classification, but want to study its performance on expected future data sets [11]. Below is a pseudo-code algorithm for *K-Fold* cross validation.

ALGORITHM K-FOLD CROSS VALIDATION:
INPUT:

K: Number of Folds
N: Number of Feature Vectors (all of which have been Classified beforehand)
V: The set of Feature Vectors
T: The classification machine

OUTPUT:

μ *: The mean % error in the expected classification rate*

σ *: The standard deviation in the % error of the expected classification rate*

Generate K subsets (S{1}…S{K}) of randomly chosen vectors from V. The size of each set S{i} is approximately N/K

for i=1,2,…,K
Train machine T using the K-1 subsets S{1},…,S{i-1},S{i+1},…,S{K}
Classify subset S{i} using machine T
Calculate e{i}:
B=number of feature vectors in S{i} incorrectly classified
L=total number of feature vectors in S{i}
e{i}=B/L
end for

Calculate μ *=mean(e{1},e{2},…,e{K})*
Calculate σ *=std(e{1},e{2},…,e{K})*

Note that each time we chose *K=10* and repeated the experiment *10* times, we constructed a total of 100 random subsets. *Leave One Out* cross validation is simply *K-Fold* validation with *K=N* (e.g., one feature vector is taken out at a time, the machine is trained with the rest of the *N-1* feature vectors, and the error for each validation is either 100% of 0%).

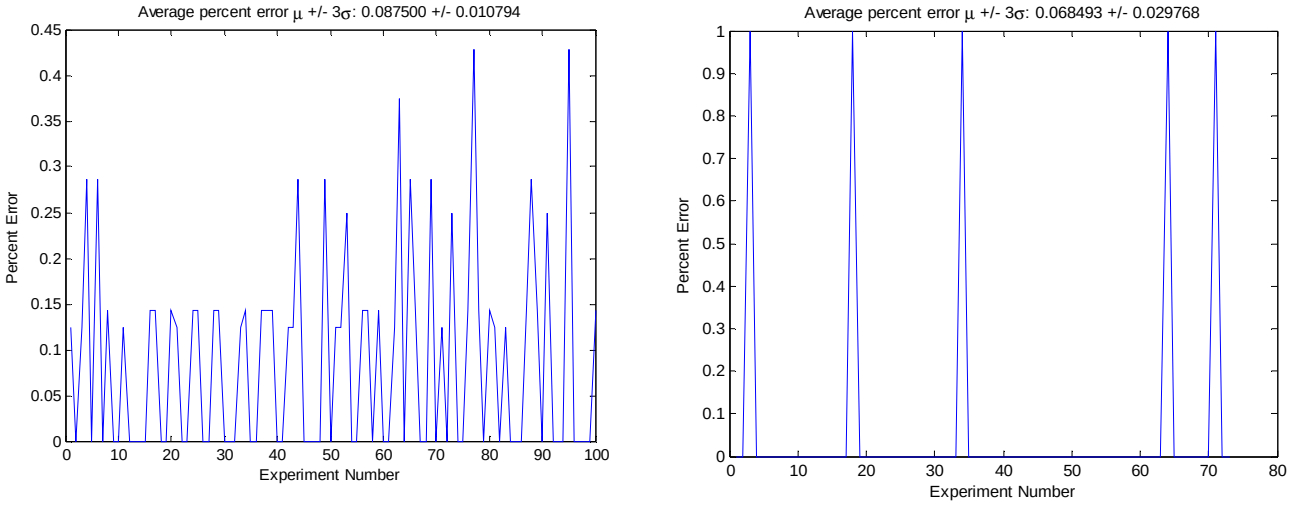

Figure 38 – (left) K=10 cross-validation study on data set, (right) Leave One Out Validation Study on Data Set

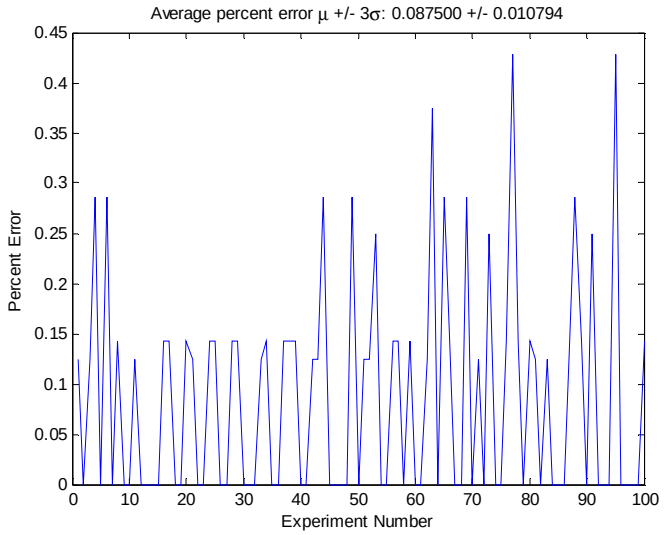

Figure 39 – *K=10* cross-validation study on reduced data set with features containing *NaN* removed

Note that the average error in our classification scheme is consistently around 8% in all of our studies. We consider this to be excellent performance, given our constraints. Our data set was very small compared to most data sets used for classification, and contained between 65-73 fasteners at any one time. One of these fasteners belonged to its own class, and we had no other fasteners similar to it. Most of the *K=10* data sets either had no misclassification or one misclassification. Our data was also incomplete in some cases. While the feature space contained 14 dimensions, a number of these values were dependent on the others (we had a smaller dimensional size in the actual physical extracted data). Most of the fasteners were also very small and we had problems extracting the threading due to noise in these cases.

Professional and Societal Consideration

Throughout the design and implementation of this project, societal considerations were not ignored. We were aware that anytime you create a feature extraction algorithm, you are making decisions based solely on appearance. In the general sense, any machine that allows automatic discrimination based on appearance can be used unethically. While our algorithm does not have the capability to make comparisons of any political or social consequence, it is important to consider these possibilities when designing automatic recognition systems.

To ensure compliance with all copyright and software usage agreements, our team adhered to proper citation practices. This includes properly citing the scholastic papers used in both algorithm development and our report. The Matlab toolboxes we used, such as the toolkits for rectification and classification, were also properly cited.

When designing the hardware used to capture the images, repeatability and reliability were the two main concerns. To ensure that the data could be obtained rapidly and with consistent results a simple wooden frame was used to mount the camera. Capturing images without use of zoom alleviated another reliability concern. This consistency of position and camera characteristics models closely the situation most desirable in a factory setting, where simplicity, standardization, and ease of installation are prized.

The economic impact of this project is clear, it will cut costs associated with stock loss. According to Home Depot, currently when fasteners are dropped on the floor they are simply placed in the rubbish. If a box breaks open, the entire box is discarded. With the ability to sort and classify fasteners it is possible to eliminate the need to dispose of entire boxes of fasteners when an accident occurs. This

ability to automatically scan fasteners also has the ability to improve distribution of stock, since the contents of packages can be scanned before shipment, ensuring no mistakes are made.

Speed of Processing

The time it takes to process a single image of a fastener is dependent on the amount of noise within the image. Large amounts of noise create additional objects that must be visited and evaluated to determine whether or not they are a fastener. This is done at the cost of additional processing time. On average, over our 73 fastener data set, it took between 8.5 and 9.5 seconds to extract all of the feature data from a single image. Once this data has been extracted into a feature vector, the distance calculation takes a negligible amount of time, allowing multiple calculations per second. In addition, our extra classification step takes under a second per data point to train the SVM and also under a second to classify each item.

The pre-processing (rectification) of the data takes the most time out of all the processing. Although this is only performed once, the time spent is significant; on the order of several hours for 20-25 images. Although this pre-processing is performed by a toolkit we did not write, this time could be reduced dramatically through two changes: conversion of the Matlab routines to C code and only rectifying the parts of the image that will not be cropped from the image later. The efficiency of the rest of the processing is already quite high, although the feature extraction algorithm could be improved through additional routines to quickly eliminate noise artifacts with less processing than currently required.

Design Verification

The preprocessing stage was the first stage to be verified. The Camera Calibration toolbox, after finding the camera parameters, calculates the error between the corners generated by a forward model of the checkerboard after image rectification (the model) and the corners found by an image processing algorithm on the original image (the truth). Below is a plot of these errors for each of the checkerboard corners over 15 calibration images. Note that the error values are very small relative to the 2272x1704 image size. In the preprocessing detailed design section, we also confirmed the accuracy of the rectified images when we noted that the difference between the R_C and R_R constants was small.

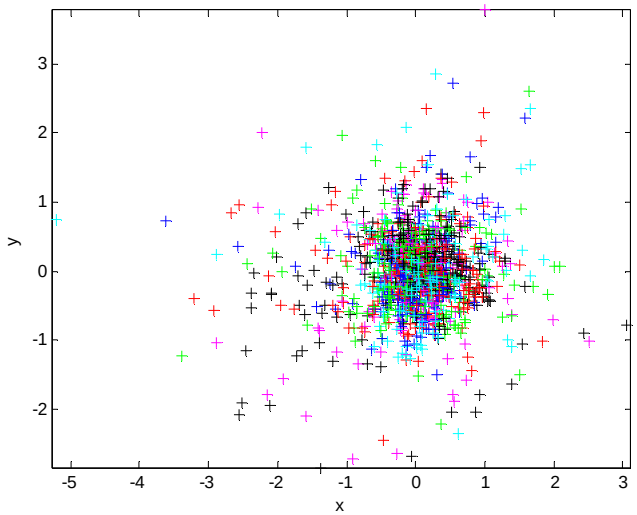

Figure 40 – Reprojection Error (Model – Truth), Units in Pixels

In other processing stages, design flaws only become apparent when an algorithm had to deal with data it was not designed to process. As a result, having a large heterogeneous dataset helps in finding these special cases. Background subtraction may remove the fastener head from image A, while the alignment algorithm may calculate an erroneous offset angle in image B. As the data extraction

algorithm only requires several seconds to run in Matlab, we simply processed our entire original data set (73 images) and inspected the results visually. Appendix E contains a subset of these images, with fasteners extracted, oriented, and borders outlined in red.

One measure of the robustness of the system is its tolerance to noise. Gaussian noise can enter the data set from a multitude of real world conditions such as excess heat around the camera. To test the tolerance of the algorithm, a pair of fasteners was examined over a range of variances of Gaussian noise and the effect on the feature vector was recorded. This data was then plotted over the range of variances for a few of the feature vector components. As shown in Figure 41, the system is extremely stable with respect to noise until the variance of the noise becomes larger than .0095. Prior to reaching this threshold value, the data set showed good resilience as seen in the Table 2 below.

Feature	Original Value	Mean Value	Standard Dev	Error (%)
Width	0.0082546	0.0081822	1.7737e-004	0.88
Height	0.034298	0.034317	4.6091e-005	.055

Table 2 - Gaussian Noise Error

However, as can be seen in Figure 41, as soon as the system hits a certain threshold of noise tolerance, it cannot produce anything similar to the expected output.

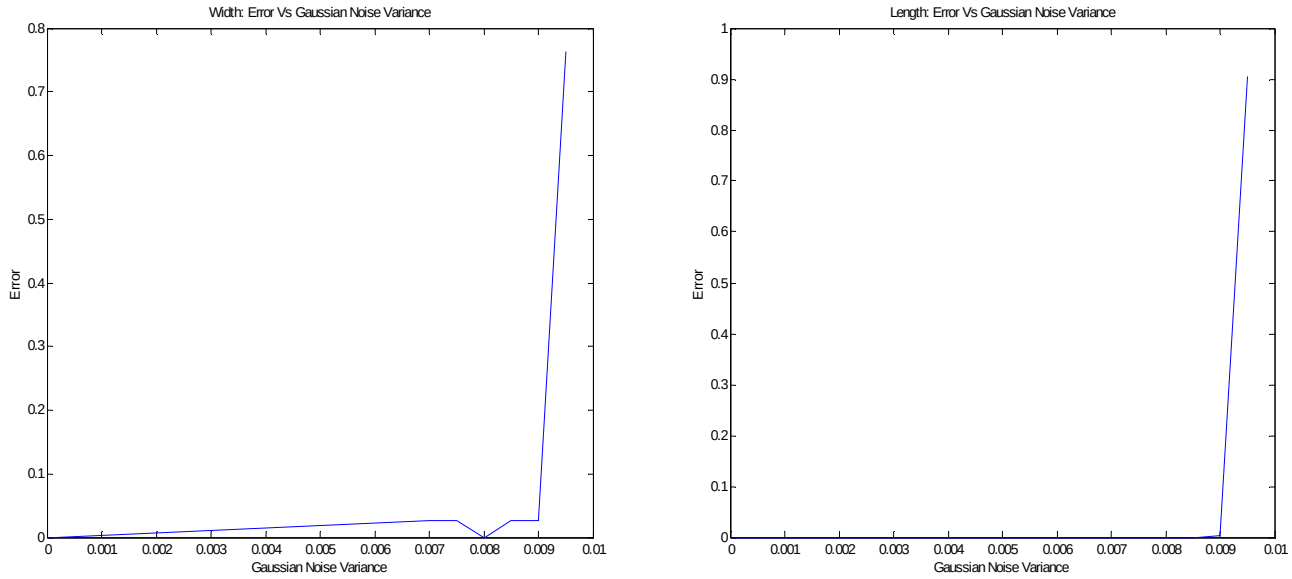

Figure 41 - Width, Length, and Error vs. Gaussian Noise

Our next investigation was to see what effect varying the threshold used to subtract the background had on the extracted feature vectors. To do this, we ran a sweep of the lower and upper threshold boundaries. Each of the widths, lengths, and thread counts was then tested against the original measured values. When the threshold was varied over a range of 2.5 to 5 standard deviations from the mean, the output showed that all the error observed was well within the original measurement error. Thus, our algorithm is invariant to the threshold data within a fairly large margin, as shown in Appendix F.

Next, the thresholding sweep was attempted on an image that ordinarily returned improper thread counts. When the sweep was completed, the datasets returned all produced very similar values for the thread count of the fastener. Thus, the algorithm works independently from the thresholding values over a rather large range. From this we concluded that our original thresholding levels of –3 and 4 standard deviations away from the mean was an acceptable range over which to threshold. The threading count

could not be improved through altered thresholding since in most cases it was the quality of the image that created the error. In many of the threading misrepresentation cases, the threading was not discernable even to the human eye.

In conlusion, our algorithm is fairly invariant to both noise and thresholding error over an acceptable range. The major limitation in the testing was simply the resolution of the images, for large fasteners with well-defined threading; the algorithm was very successful in picking up the feature values, yet for small fasteners, with minimal separation between threads, it was not as accurate. Accuracy could be improved through higher resolution images, or images taken closer to the subject material.

Costs

The cost of our project includes both software and hardware. We assume that testing objects can be obtained free of charge from construction sites and hardware stores who need to dispose of these items anyway.

Line Item	Cost and Multipliers	Subtotal
Labor	$30 x 15hrs/week * 15 weeks * 3 * 2.5	$50625
Matlab License	$1900 * 3	$5700
Workstations	$1500 * 3	$4500
Optimization Toolbox	$900 * 3	$2700
Statistics Toolbox	$800 * 3	$2400
Signal Processing Toolbox	$800 * 3	$2400
Conveyor Belt	$1200	$1200
Hi-Res Digital Camera	$1000	$1000
Misc. Supplies	$500	$500
Camera Mount Supplies	$500	$500
Open CV	$0	$0
	Total	$71525

Table 3 – Cost Estimates

The total budget for our project is estimated at $71,525.

Conclusion

The goal of this project was to take any two fasteners from three main categories and calculate a distance between them. Comparisons were to be based on images of the fasteners. Our base requirement was to be able to process images of single fasteners; however, we extended the project to be able to handle multiple fasteners in the same image.

The task of separating the fastener from its background was accomplished through thresholding techniques. The background had a known mean and distribution, both well separated from the object of interest. This allowed simple background subtraction to be performed. Morphological operations and filtering then gave a clean outline of the fastener. Following this segmentation, the fastener was rotated to a standard orientation and a contour describing the outline of the fastener was generated.

From this outline, a number of features were extracted through both traditional and some custom image recognition techniques and algorithms. After extraction of this data, a feature vector containing the extracted information was created. The features measured and included in the feature vector were length,

diameter of shaft, width of head, head type, tip angle, number of threads, thread distribution, standard deviation of width, and inner and outer threading widths.

After careful consideration of the differences that most separated wood screws from machine screws from nails, it was decided to calculate the distance based on length, shaft diameter, head type, tip angle, thread distribution, and standard deviation of width. The distance metric was based upon weighted averages of a mixture of percent differences between parameters and negative exponential scales of the differences. This metric returns a value of zero for identical fasteners and returns a one for fasteners that are completely unalike. This fulfils the basic deliverable of the project proposal.

The bells and whistles implemented in this project are image rectification that fixes radial distortion to improve accuracy of measurements near the edges of our images, recognition and processing of multiple fasteners in the same image, and classification of fasteners through use of Support Vector Machines.

Overall this project requires only a small amount of modification to be realistically useable in an industry setting.

References

1. D.A. Forsyth, J. Ponce, *Computer Vision: A Modern Approach*, Prentice Hall, 2002.
2. M. Wasilewski, “Active Contours using Level Sets for Medical Image Segmentation”, http://www.cgl.uwaterloo.ca/~mmwasile/cs870/.
3. J-Y. Bouguet, “Camera Calibration Toolbox for Matlab”, http://www.vision.caltech.edu/bouguetj/calib_doc/.
4. Z. Zhang, “Flexible Camera Calibration by Viewing a Plane From Unknown Orientations”, International Conference on Computer Vision (ICCV'99), Corfu, Greece, p. 666-673, September 1999.
5. J. Heikkilä, O. Silvén, “A Four-step Camera Calibration Procedure with Implicit Image Correction”, CVPR 1997.
6. C. JC Burges, “A Tutorial on Support Vector Machines for Pattern Recognition”, 1998, http://citeseer.ist.psu.edu/burges98tutorial.html.
7. N. Cristianini, “Support Vector and Kernel Methods for Pattern Recognition”, http://www.support-vector.net/tutorial.html.
8. T. Hastie, R. Tibshirani, “Classification by Pairwise Coupling”, Ann. Stat., Vol. 26, No. 2, 451-471, 1998, http://www-stat.stanford.edu/~trevor/Papers/2class.ps.
9. J. Weston, C. Watkins, “Multi-class support vector machines", Technical Report CSD-TR-98-04, Department of Computer Science, Royal Holloway, University of London, Egham, TW20 0EX, UK, 1998, http://citeseer.ist.psu.edu/8884.html.
10. J. Weston, A. Elisseeff, Gokhan Bakir, et al, “SPIDER: object-oriented machine learning library”, http://www.kyb.tuebingen.mpg.de/bs/people/spider/.
11. R. Guiterrez-Osuna, “Intelligent Sensor Systems, Lecture 13: Validation”, PRISM: Pattern Recognition and Intelligent Sensor Machines, Texas A&M University, http://research.cs.tamu.edu/prism/lectures/iss/iss_l13.pdf.