18-551, Fall 2005

Group 1, Final Report

Can’t Copy This!

Daniel Weller, Yu Hsin Lin, Ruben Perez

{dweller, yhl, rubenp}@andrew.cmu.edu

Table of Contents

Introduction

In the 21st century, digital technologies, such as complex scanners, copiers, and computer software, have vastly increased one's ability to counterfeit just about anything. For instance, even students can manufacture bank notes realistic enough to fraudulently purchase school lunch. Money, stamps, checks, coupons, IDs, signatures, and product labels have become the subject of ever-increasingly sophisticated counterfeiting techniques. In order to diminish such an illegal phenomenon, we propose to develop a system that can be integrated into scanners, copiers, and other electronic equipment frequented by counterfeiters. With current technology, fake money is within reach of even average citizens; with the "Can't Copy This", authorities could begin to curb counterfeiting among the masses. We plan to combat this problem through the application of image processing and pattern recognition. The Central Bank Counterfeit Deterrence Group (CBCDG) has already organized hardware and software manufacturers to fight this problem through the development of the Counterfeit Deterrence System (CDS) [1]. Despite this, people are able to outwit these safeguards and utilize commercial products to counterfeit currency. Therefore, more robust solutions are required.

Some pre-existing efforts, in coordination with the CBCDG, have found their way into the mainstream. Current versions of Adobe Photoshop, Jasc Paint Shop Pro, and similar software have integrated simple schemes to thwart counterfeiters. Printers in Japan are required to have embedded features to place microscopic dots on all their printouts that indicate their sources through identifying serial numbers. While the details of these schemes are unknown, they have proven to be less than perfect since counterfeiting still presents to be an issue nowadays [2]. A different approach pursued by MediaSec Technologies involves embedding special (invisible) images on product labels and CDs to enable a scanning device to differentiate real products from fake look-alikes [3]. However, this approach will only stymie counterfeiters for so long since it does not actually address the heart of the problem. In order to stop counterfeiting for good, we have to prevent the duplication of anything that resembles treasury notes. Unfortunately, resemblance is a difficult concept for a computer to grasp without aid of software that is capable of carrying out detection and classification of prohibited objects that are not meant to be photocopied or scanned. This fundamental need is what sparks our group to come up with the idea of "Can't Copy This" as our term project.

The United States Secret Service describes the requirements for the legal reproduction of treasury notes:

> The Counterfeit Detection Act of 1992, Public Law 102-550, in Section 411 of Title 31 of the Code of Federal Regulations, permits color illustrations of U.S. currency provided:
>
> 1. the illustration is of a size less than three-fourths or more than one and one-half, in linear dimension, of each part of the item illustrated;
>
> 2. the illustration is one-sided; and
>
> 3. all negatives, plates, positives, digitized storage medium, graphic files, magnetic medium, optical storage devices, and any other thing used in the making of the illustration that contain an image of the illustration or any part thereof are destroyed and/or deleted or erased after their final use. [4]

Using advanced signal processing techniques, we have come up with a scheme that detects targets of probable counterfeiting (treasury notes in our case) by conducting pattern recognition. The first step for our project is to construct a reference database that contains images of front and back of the various denominations ($1, $5, $10, $20). With the reference database, we are able to classify prohibited objects using correlation and ultimately prevent them from being copied or scanned. The nature of our technique differs from what has been done in the industry because our goal is to achieve prevention (more effective and thorough in our opinion) rather than detection and recovery.

Previous Work

Looking through the previous projects done in 18-551, we did not find any project that addresses the issue of counterfeiting. Nevertheless there have been a number of projects that deal with detection and classification. "Where's the Ball?" from Spring 2004 and "Face Detection for Surveillance" from Spring 2002 are some of the projects that fall into the category above [5, 6]. Some of the techniques that we considered are previously investigated by some the other groups; for instance blob labeling, linear correlation, PSR (peak to side-lobe ration), geometric moments, and morphological processing. However in terms of linear correlation, our project is slightly different because we correlate through the entire image while the other groups run correlation on small areas containing dominant identifying features, such as people's faces.

As for details of work done previous in the industry, Adobe Photoshop CS digital editing package includes a counterfeit deterrence system designed to prevent users from accessing images of currency. When the deterrence system detects an attempt to access a currency image, it automatically aborts the operation, displaying a warning message and directs the user to a website with information on international counterfeiting laws. This anti-counterfeiting system actually exceeds the requirement of U.S law which allows color reproductions of U.S. bank notes so long as the reproductions are smaller than 75 percent or larger than 150 percents of actual size [2]. Some other companies with similar products actually voluntarily embed features of anti-counterfeiting similar to that of Adobe to support the counterfeiting laws because cases of counterfeit crime have soared tremendously in the recent years. Despite the similarity in concepts, the purpose of our project is actually very different from those of what exist in the industry nowadays. Our program prevents people from acquiring images of prohibited objects while most of the programs out there either prevent people from modifying an image or leave traces of where the printouts are made. To further emphasize the difference, our project aims to stop people from scanning or copying images of treasury notes, thereby fundamentally eliminating counterfeiting by targeting the root of this crime.

System Level Design

Our project consists of a HP PSC 1610 scanner, a PC and an EVM. The scanner will be used to scan in an image, which will then be transferred to the PC side by means of Windows Image Acquisitions (WIA). The PC receives the image and performs scaling, recoloring, image morphing and blob labeling. Then the PC sends the processed image to the EVM. On the EVM side, we first determine the rotational angle of the object using geometric moments. After acquiring the rotational angle, we then un-rotate the object using bilinear interpolation. Edge enhancement using Sobel technique is performed after the process of un-rotation to improve the PSR of our correlation process. Finally we perform linear correlation on the test image against our reference database and send the classification result back to the PC, where the results will be displayed by our GUI, called ScanView.

To further illustrate our system level design, we will discuss briefly the tasks that are done on both the PC and the EVM sides:

The PC communicates with the scanner using the Windows Acquisition Application (Handled by our ScanView software) to retrieve the image. The PC begins by resizing the given image to 100 dpi resolution. We found that the 100 dpi resolution produces the best results for our detection system. Our system does not rely on color recognition, so we must convert the image to grayscale. We create a binarized representation of the image to aide in our image extraction protocol. The PC then performs morphological processing on the binarized image, which is intended to close the tiny spaces between the numerals in the bottom right of the $5, $10 and $20 bills in order to avoid floating numbers. The PC then performs a blob labeling process that extracts individual objects from the input image and sends each extracted object to the EVM for image classification. Simultaneously, the PC transmits the reference database to the EVM for future correlation processing.

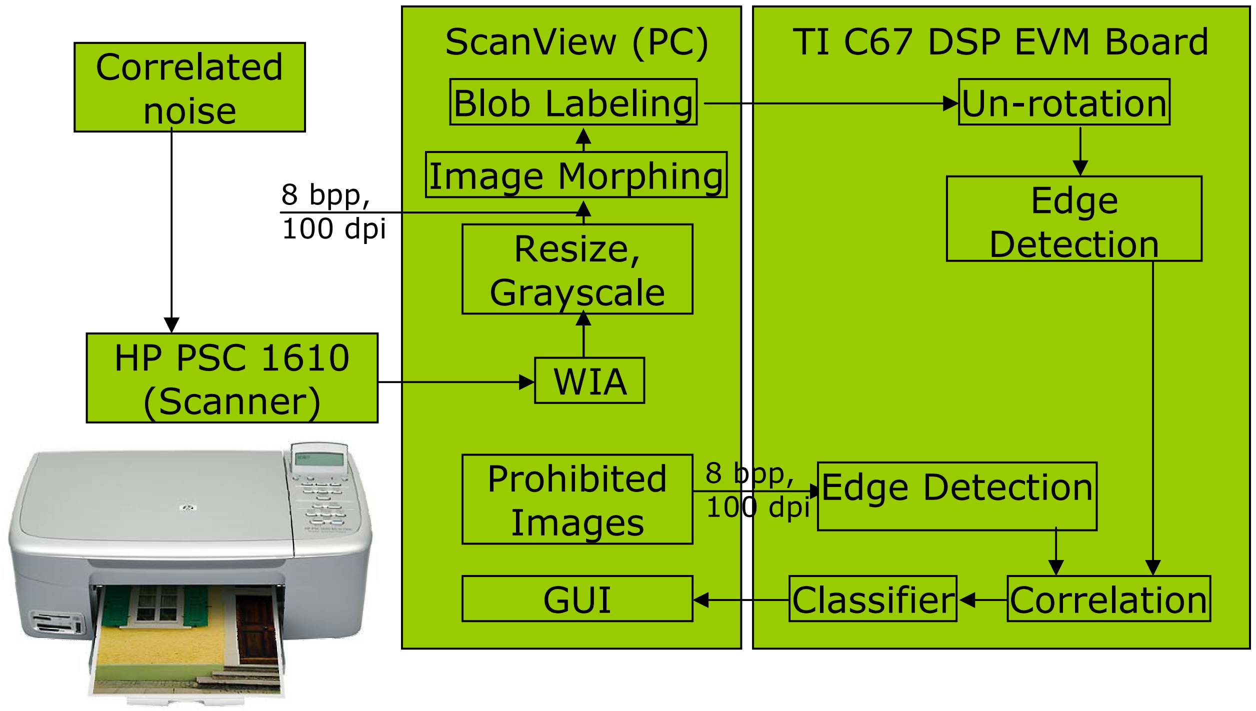


Figure 1: System-Level Block Diagram

The first task the EVM must do with the received extracted images is to un-rotate them. We utilize a trivial mathematical computation that relies on moment extraction to determine the proper angle that the extracted object must be rotated to help in classification later. With the un-rotated image present, the EVM then performs a Sobel computation on both the extracted images and the list of reference images in our database to help determine the surrounding edge of each image. These edge enhanced images (extracted objects and reference images) are correlated together, which will be crucial in our peak-sidelobe ratio functionality. The PSRs for these images are utilized as our classifier mechanism to determine if any of the extracted objects are illegal to copy. The legality of the objects is passed back to the GUI where we specify which objects cannot be copied (as well as a classic musical interlude).

This handshaking process between the PC and EVM takes some time. Ideally we preferred that our entire system would take no more than 10 seconds to receive an image, process that image, and return valid or invalid objects to be copied. However, we realized that our system takes just over 1 minute to perform our processes. This is due to many underlying transfers in memory. The transfer time from the scanner to the PC, via a USB port, takes approximately 3 to 5 seconds. We bounded our system to have a maximum extracted image of 500x500 pixels, at 8 bits per pixel (in grayscale); this translates to 250,000 bytes per extracted image. The estimated transfer time for the extracted image from the PC to the EVM is less than 0.1 seconds, using a Host Port Interface (HPI); which translates to 3 Megabytes per second. The reference database that the extracted image must be tested against holds 8 images (the front and back of each un-rotated bill), each approximately 1.5”x5”; which translates to 75,000 bytes per reference image (600,000 bytes for all 8 images). The PC must transfer the reference images as well to the EVM, which takes approximately 0.19 seconds (for all 8 images) using a HPI; which translates to 3 Megabytes per second. Each of the images, however, is transferred one at a time and is fairly time intensive. The transfer of all these images requires quite a bit of the EVM’s memory. The external memory that is utilized is approximately 3.5 Megabytes and the internal memory that is utilized is approximately 64 Kilobytes. The off-chip SDRAM is used to store the extracted image (250,000 bytes), reference image (250,000), un-rotated image (1 Megabyte), and a large temporary buffer for Sobel and correlation outputs (2 Megabytes). The on-chip SDRAM is used for our ping-pong buffers, program memory, stack, etc.

Algorithms Used

Image Scaling

The first algorithm that is performed in our program is image scaling, which simply takes the input image and resize to 100 dpi (using the scaling factor *s*) for obtaining a more desirable PSR for classification. This is achieved using nearest neighbor interpolation. For each destination pixel in the destination image, we use a scaling transformation (1) to map that pixel to a pixel in the source image. However, this pixel does not have to be an integral value, so to get the non-integral value of the pixel from the source image, we use nearest-neighbor interpolation – essentially using the value of the pixel closest to our non-integral source pixel. Due to the way how Windows handles scaling with nearest neighbor interpolation, this might lead to a poor correlation result as illustrated by our demonstration and data analysis. Nevertheless, this step is essential to the completion of our system because our match filter is not scale invariant; therefore we have to make sure that the images that we are analyzing are the same resolution as the images in our reference database.

$$\begin{bmatrix} x \\ y \end{bmatrix} = \frac{1}{s} \begin{bmatrix} x' \\ y' \end{bmatrix} \quad (1)$$

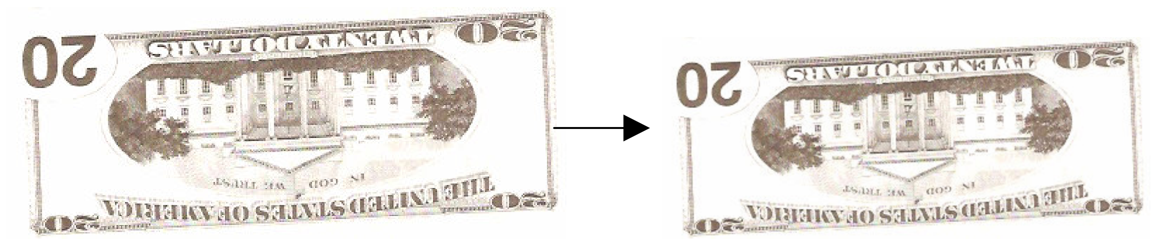


Figure 2: Rescaled Image

Grayscale Computation

The second algorithm that our program employs is grayscale computation, which transforms an image from the RGB to the grayscale domain. The reason that we perform this process is that it is difficult to carry out our operations in each of the color domain independently, and converting images to grayscale and working with light intensity rather than colors simplifies our problem. In addition, by using grayscale images instead of color images, we are able to save some memory. This is achieved through the use of NTSC-approved formula (2), generously provided by the Mathworks [7]. Note that this process is carried out on an inverted image, so the actual gray intensity is specified by (3).

$$gray = 0.3R + 0.6G + 0.1B \quad (2)$$

$$gray = 255 - (0.3R + 0.6G + 0.1B) \quad (3)$$

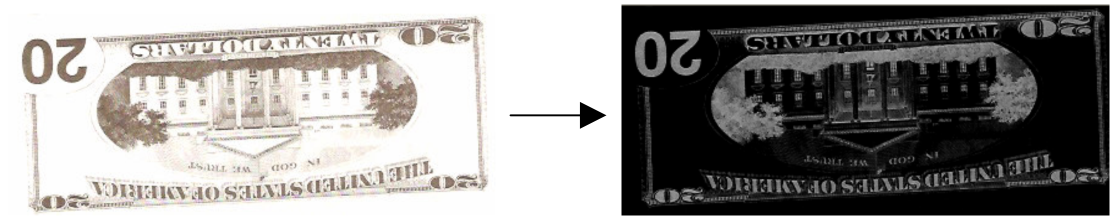


Figure 3: Object Converted to Grayscale

Black and White [Binarizing]

Following converting the images from the RGB to the grayscale domain, we proceed to binarize images by comparing each pixel's intensity against a threshold that is previously computed on grayscale images. After analyzing several input images, this threshold is found to be 1.5 / 255. This threshold was assumed to be approximately universal; however, during our demo, we discovered that thicker objects did significantly alter the hue of the background. Note that we perform binarization on inverted images; as a result, our background color will be black while our foreground color will be white. The reason that we decide to take this extra step is that binarized images would aid us to do our next algorithm morphological processing, which includes erosion, dilation and closing on an image as described in the section below.

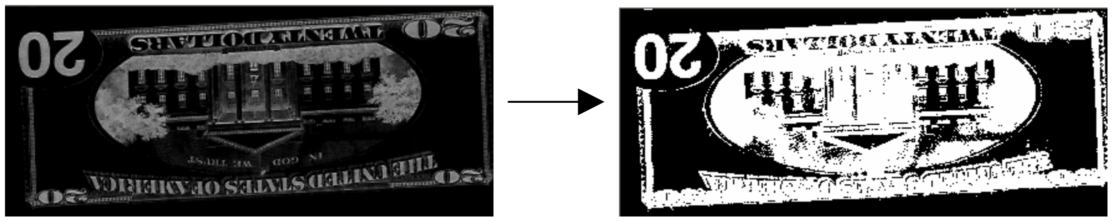


Figure 4: Object Binarized

Morphological Processing

Morphological operations are used in the process of segmenting images. Image segmentation refers to the process of finding regions in an image that represent objects or meaningful parts of objects. Morphological processing can be used for filling small holes in objects, isolating adjacent or slightly overlapping objects, or joining broken boundaries into continuous segments. For our case, we are using it for joining the floating numbers on the bottom right of the back of the $5, $10, and $20 bills (Fig. 5). For our project, we perform closing which involves erosion followed by dilation, two of the principal morphological operations. Dilation allows objects to expand, thereby filling in small holes and connecting disjoint objects. Erosion shrinks objects by eroding their boundaries.

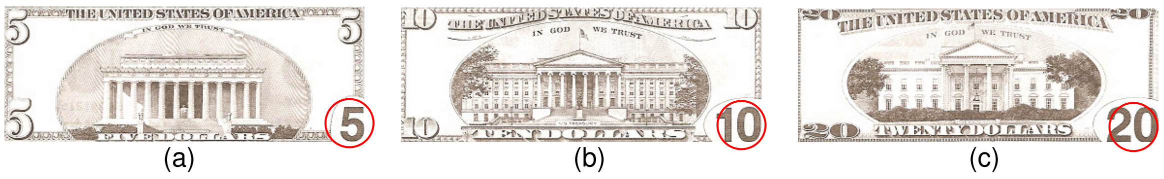


Figure 5: Problematic Reference Images of the (a) $5, (b) $10, and (c) $20 bills

The dilation process, similar to convolution in a manner, slides the overlaying structuring element across the image. Dilation does nothing to background pixels, but it grows foreground pixels by performing a bitwise OR operation on every pixel under the structural element when the center of the element is on the foreground. The dilation process is illustrated in Figure 6.

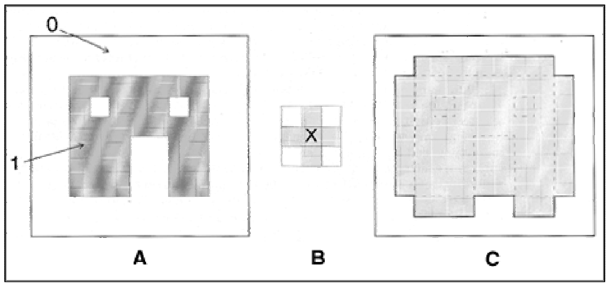


Figure 6: Dilation process – A) Original image, B) Structural element, C) Image after dilation [8]

Erosion does the opposite of dilation, thinning foreground areas, by changing foreground pixels to background pixels when all the ‘1’s in the structural element are not present in the corresponding locations in the original image. Thus, erosion only affects foreground pixels on the edge of foreground regions.

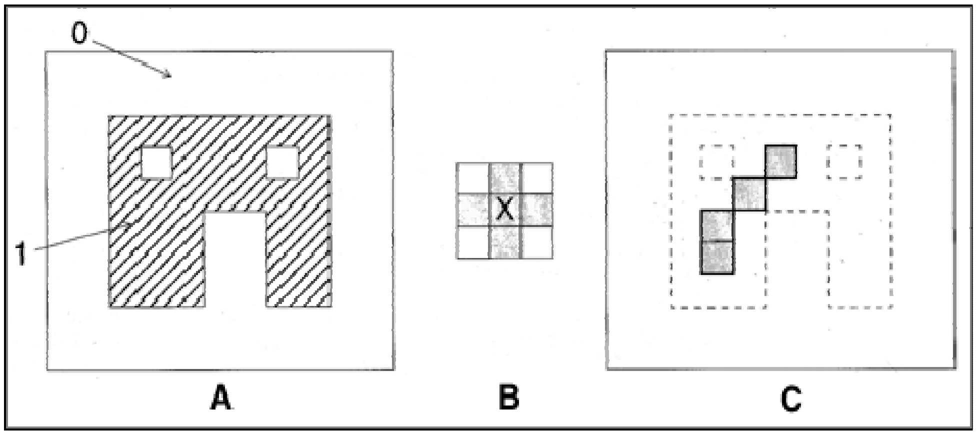


Figure 7: Erosion – A) Original Image, B) Structural element; x = origin, C) Image after erosion; original in dashes [8]

Morphological closing is simply the process of performing dilation, followed by erosion, using the same structural element for both operations. Note that the sequence of operations is important; an erosion followed by a dilation is termed “opening,” and will yield entirely different results. The size and shape of the structural element also significantly impacts the result. A square structural element will emphasize sharper edges than a circular element, and an element’s size dictates how large a section it can close. In general, an n x n structural element can close a gap of size n-1. As illustrated in the diagram below, the original image has a number of small holes and spacing in the object, which get “closed” or filled after the closing process. The size of the structuring element can influence the size of the area which the closing process closes while its shape will affect how sharp the object will end up after the closing process.

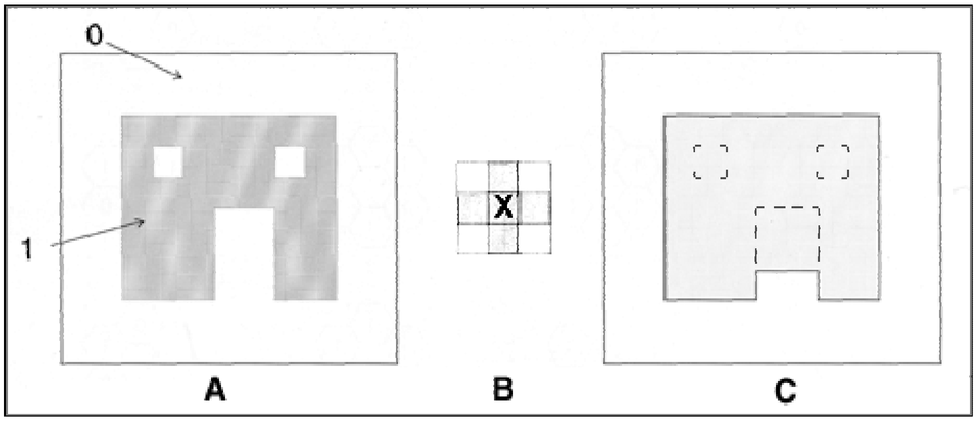


Figure 8: Closing – A) Original Image, B) Structural element; x = origin, C) Image after closing; dilation followed by erosion; original in dashes [8]

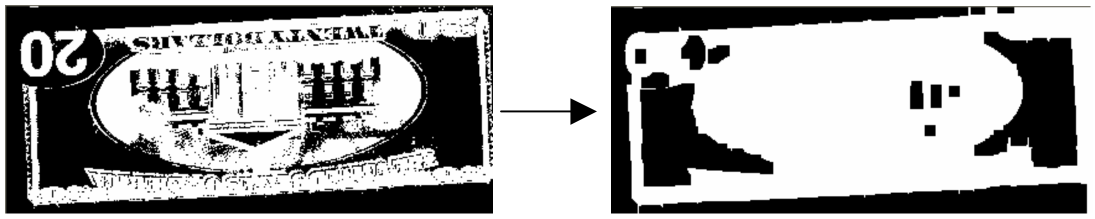


Figure 9: Morphological Processing

Blob Labeling

The blob labeling technique is intended for image segmentation, particle size characterization as well as microscope magnification calibration. In our case, we are using it for the first purpose stated. Our blob labeling uses a four-way connected square blob pixels with four neighbors, which makes the background eight-way connected. Blob labeling images involves "scanning" the image in raster order, labeling pixels according to their already labeled neighbors. If there are no labeled neighbors, then a new label is used. Some of the common forms of the labels are integers stored in a new "image" or array which will become blob images and are used to be associated to the corresponding pixels in the original image. If the blobs are skinny and curved, then various ends or humps will appear as separate objects during this labeling scan, and will be joined together later on when sufficient rows have been scanned to connect all the pieces together. While this process continues, two or more different labels will be associated together as one object. This is achieved by maintaining an equivalence table during the initial labeling scan, then simplifying the table so that all labels in any one blob are known to be equivalent to one label that is eventually used for the whole blob. This re-labeling iteration is done in a second scan of the blob image. To further illustrate the methodology described above, we provide a simple transitional diagram (Fig. 10) outlining the process of blob labeling.

0	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	1	1	0	1
0	0	1	1	1

Step1

0	0	0	0	0
0	1	0	0	0
0	0	2	0	0
0	3	2	0	4
0	0	2	2	2

Step2

0	0	0	0	0
0	1	0	0	0
0	0	2	0	0
0	2	2	0	2
0	0	2	2	2

Step3

Figure 10: Blob Labeling Example

After the re-labeling iteration using the equivalence table where it states 3->2 and 4->2, the blob labeling process concludes that there are two blobs or objects in this image. In addition the following are some the properties associated with our blob labeling technique.

1. The input is an inverted binary image.
2. The output is a blob image or an ugly blob mask image.
3. Light intensity threshold based from the binarizing process is used to identify pixels as either object or background.

The reason we incorporate the blob-labeling technique in our project is that an input image may have multiple objects contained in it; blob-labeling is a simple method to extract disjoint objects from an image [9]. In order to properly detect and classify each object, we must be able to tell them apart from one another, extract them from the original multi-object image, and make them into a separate image where only one object is present. Dealing with images where only one object is present greatly simplifies our life in terms of classification.

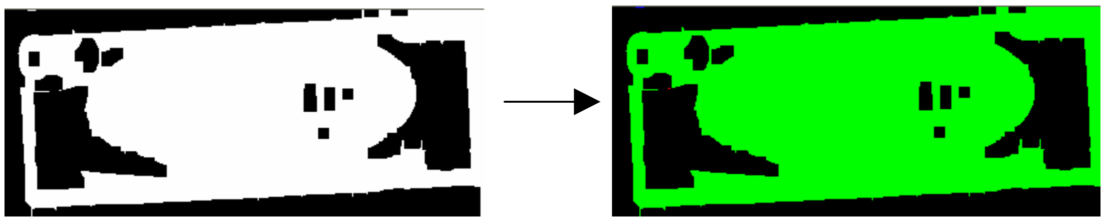


Figure 11: Blob Coloring

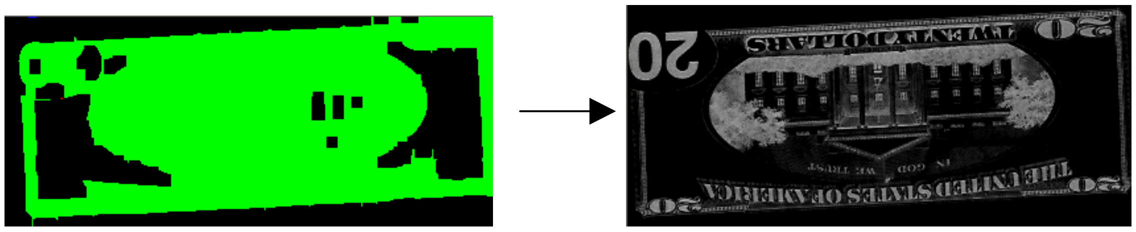


Figure 12: Extracted Object from Blob Labeled Image

Orientation Estimation using Geometric Moments

We use geometric moments to estimate the orientation of rotated objects. Geometric moments are easy to implement and offer fast computation of image rotation; however, they are also highly sensitive to image noise and also exhibit large variation in the dynamic range of values, thus yielding wrong results in a correlation matching algorithm. By computing the principal axis of rotation of the input image using geometric moments as described in equations (4) and (5), we are able to conclude by what angle we need to rotate the image to align its principal axis to those of reference images (un-rotated) [10]. After obtaining the rotational angle, we can proceed to un-rotate the image using bilinear interpolation.

$$m_{p,q} = \sum_{y=-\infty}^{\infty} \sum_{x=-\infty}^{\infty} x^p y^q I(x,y) \quad (4)$$

$$\tan 2\theta = \frac{2m_{1,1}}{m_{2,0} - m_{0,2}} \quad (5)$$

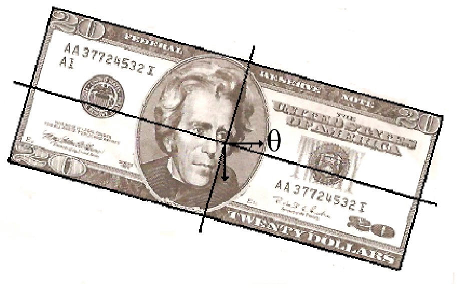


Figure 13: Principle Axes of a Rotated Image

Un-rotation Using Bilinear Interpolation

Before two images can be compared using linear correlation, their principle axes must be properly aligned, since the linear correlation operation is highly sensitive to rotation. Image rotation is generally performed by mapping the un-rotated pixels to pixels in the input image using a transformation matrix. Since we are performing un-rotation on digital images, the un-rotated image cannot be a perfect representation of the original image except in certain specific circumstances (rotation by any multiple of

90 degrees, or if the input image is rotationally symmetric). This is a result of the fact that the transformation matrix does not constrain the coordinates in the original image to integral values. Therefore, we must resolve these non-integral values using a form of interpolation; we choose bilinear interpolation because of its smoothing properties and ease of computation.
To un-rotate an image, one simply maps every output pixel to the corresponding input pixel using the following transformation (6):

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \quad (6)$$

The simplicity of this formula lies in the pre-computation of the cosine and sine of theta, which reduces rotation to simple additions and multiplies.

Once we have the corresponding input pixel, we have to perform bilinear interpolation to compute the value at that pixel (and the corresponding output pixel). Bilinear interpolation performs the weighted average of the four surrounding pixels in the input image using the area overlap of the input pixel over each surrounding pixel as the weights for each surrounding pixel. Some pixels lie along the edge of the input image, in which case, some surrounding pixels may be outside the valid image area. To produce correct results, one simply must use the value of the nearest valid pixel for those surrounding pixels outside the valid image area. Assuming a mapping from the destination image to the source image such that the desired pixel lies (Δx, Δy) away from the upper-left nearest source pixel, the value of the bilinear interpolation, *p*, is computed according to (7).

$$p = p_{0,0}(1-\Delta x)(1-\Delta y) + p_{0,1}(1-\Delta x)(\Delta y) + p_{1,0}(\Delta x)(1-\Delta y) + p_{1,1}(\Delta x)(\Delta y) \quad (7)$$

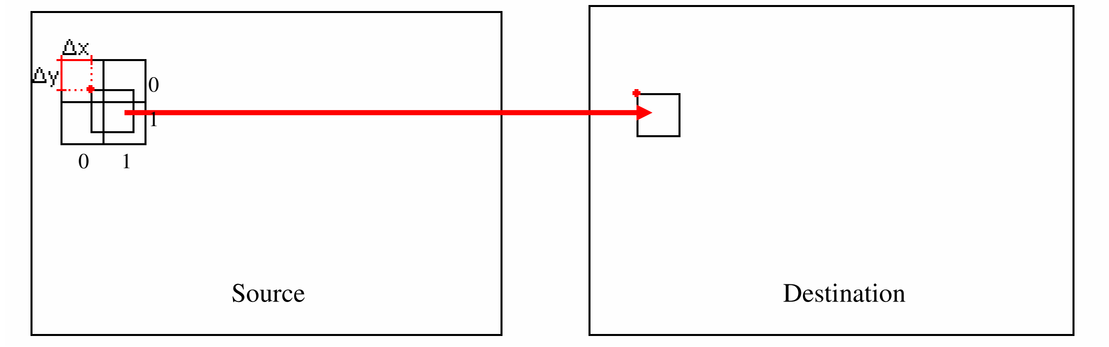


Figure 14: Bilinear Interpolation

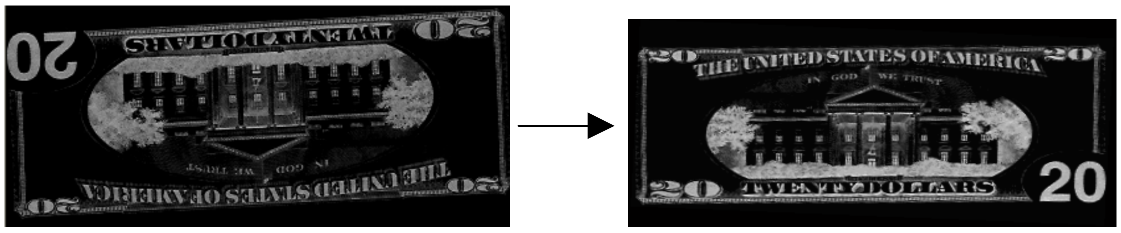


Figure 15: Image Un-rotation

Edge Enhancement Using Sobel

Edge enhancement is a technique for filtering of images. The principle behind edge enhancement is that different parts of an object are represented by different pixel intensities; therefore we can also imagine that object boundaries can be characterized by rapid intensity changes in the image. Nevertheless, this intensity pattern or discontinuities may also originate from noise in the image, which can be suppressed by the use of suitable filter kernels during the process of edge enhancement. The idea behind edge enhancement is to calculate the derivative of the image intensity, sine a large derivative represents a local intensity change. However, the direction of the derivative is also important, so edges are only detected in a preset orientation. For our project, we use the Sobel filter described in Matlab [9], which calculates the magnitude of the directional derivatives along the horizontal and vertical orientation. (2-D Edge Enhancement) It is one of the most commonly used edge enhancement filters nowadays. The reason we decide to perform edge enhancement is that we notice a dramatic improvement to our PSR when we perform edge enhancement. Finally, please note from our system level diagram that the edge enhancement process is performed on both the reference images and the input images and is carried out after the unrotation of the input images.

$$\underset{(a)}{\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}} \underset{(b)}{\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}}$$

Figure 16: 3x3 Correlation kernels for (a) horizontal and (b) vertical gradient components

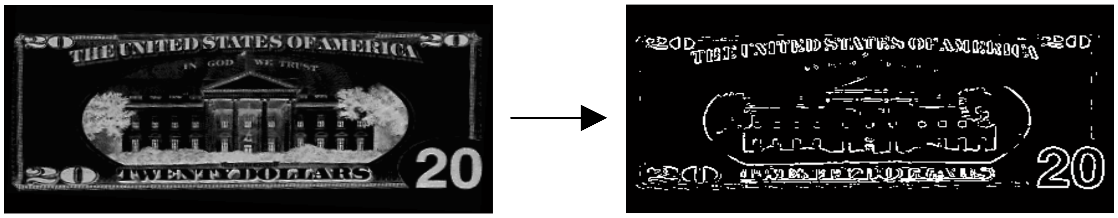


Figure 17: Edge Enhancement

Linear Correlation

Linear correlation, computed according to (8), is a straight forward type of correlation that indicates a measure of the similarity between two signals as a function of time shift between them. When the two signals, in our case two images, are similar in shape and unshifted with respect to each other, their product is all positive, forming a constructive interference where the peaks add up and the troughs subtract, further emphasizing the peak. The area under this curve gives the large value of the correlation function at point zero.

$$(f * g)(t_1, t_2) = \sum_{\tau_1=-\infty}^{\infty} \sum_{\tau_2=-\infty}^{\infty} \bar{f}(\tau_1, \tau_2) g(t_1 + \tau_1, t_2 + \tau_2) \quad (8)$$

PSR Computation

Finally we use peak to sidelobe ratio (PSR) for classifying if an object is prohibited or not. PSR is used to measure the peak sharpness and can be computed according to (9) [11]. In order to compute PSR, we take the center part of the correlation plane of two images and define a peak radius which in our case is three. This translates to a maximum peak neighborhood of 6 x 6. As for the neighborhood radius, which is 7 in our case, forms a region of 14 x 14. During the final stage of our project, we came to the realization that by relying only on PSR, we might run into the problem of inaccurate classification whenever there is a small peak somewhere in the designated region with a good PSR. If we only use PSR for our classification purpose, we would falsely identify an object to be something that it is not. Fortunately, by adding additional constraints, like a threshold on peak values, we will be able to discern if a good PSR is valid or not by assessing the height of its peak value.

$$PSR = \frac{peak - \mu}{\sigma} \quad (9)$$

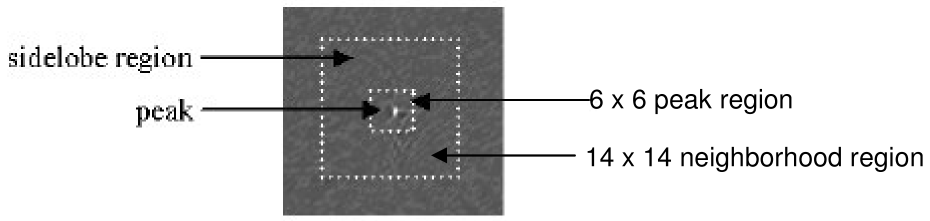


Figure 18: PSR Peak and Neighborhood Regions [12]

Since each reference object returns different PSRs, we decide to set individual PSR threshold for each reference image and perform our classification based on those eight PSR thresholds.

Implementation

The implementation of our system involves two software packages – a graphical user interface (GUI), called ScanView, written in Microsoft Visual C++ (6.0) that uses the Microsoft Foundation Classes (MFC) and code, called ScanEVM, for the TI C67 EVM, a digital signal processor, written in machine-optimized C. The GUI must handle communication with the scanner/imaging device, as well as perform preprocessing of the input image, and communicating with the EVM. The EVM software handles the classification of each object found in the GUI's input image and sends the required information back to the GUI for feedback to the user.

PC-Scanner Communication

In order to acquire images from the scanner or another image acquisition device, we chose to use the Windows Image Acquisition (WIA) interface specified in the Microsoft Platform SDK. Because of the platform dependency of this interface, unfortunately, this software is only compatible with the newest versions of the Microsoft Windows operating system (Windows 2000, Windows XP). One accesses the features of WIA using Microsoft's Component Object Model (COM), a specification for implementing and using interfaces abstracted across applications. COM is a widespread standard throughout the Windows programming world and serves as the primary interface for the multitudinous features of the Windows shell.

Because of its low-level presence in the Windows operating system, COM must be initialized as one of the first tasks of a COM-using application. Since our software also utilizes MFC, we utilize the global function AfxOleInit() instead of the more traditional CoInitializeEx() to initialize our COM interfaces. During application initialization, we also initialize our interface to WIA's device manager using the familiar CoCreateInstance() function, as well as our EVM board communication, which is described later.

Once we have obtained an interface to the WIA device manager, we can now (or ask the user to) choose an imaging device to use to acquire an image. WIA has built-in support for many of the newer scanners and cameras presently on the market; however, as we found out during our use of the HP 6200C flatbed scanner, that support among legacy devices (that use TWAIN drivers) is practically non-existent. In any case, WIA provides a method to prompt the user to choose an imaging device through the SelectDeviceDlgID() stub of the WIA device manager interface. This function uses the Windows common dialog typically used to select a scanner or camera. Of course, if there is only one scanner on the system, the dialog box does not actually appear but simply chooses that device automatically. While one can hypothetically download images from cameras in this manner, we chose to restrict our software only to scanners that support WIA to reduce the complexity of the acquisition process.

Once we have chosen a device, we can use that device to enumerate the hierarchical tree of WIA items that describe that device. This tree usually includes a root item, used to access general properties of that device, and subitems for each individual imaging option (flatbed, ADF, etc.) present. A camera, for instance, would have a subitem for each picture stored in its memory. We can use the root WIA item to prompt the user for the scanning resolution, color depth, brightness, and contrast levels, as well as the area of the image to scan, using the DeviceDlg() stub of the root WIA item interface; this dialog contains a preview pane to simplify the specification of the selection area.

Once we have obtained the desired device parameters, we can perform the scanning operation. There are two modes supported in WIA: synchronous and asynchronous operation. Synchronous operation involves acquiring the entire scanned image in one step; however, this method does not allow for paging the data in subsequent processes. Asynchronous, or banded, operation utilizes a callback function class that contains a BandedDataCallback() function; we chose this process because of the potential for paging data. This function is called whenever a band of data is ready to be saved on the PC. Unfortunately, a rather annoying feature of the image acquisition phase is the incessant appearance of a “Server is busy” message popping up from the Windows operating system, stating that the user is attempting to switch to a task that is busy. Of course this is not ideal behavior (one must repeatedly press the Cancel button to dismiss each dialog box), it does not appear to interfere with the overall operation of the scanner. The acquired image data represents a device-independent bitmap (DIB) and thus does not require special conversions or decoding. One can simply use the CreateDIBitmap() Windows API function to acquire an image handle (HBITMAP) to the data. The data can be in almost any resolution, and thus must be scaled to a resolution of 100dpi, and have almost any color depth. For the purposes of our demonstration software, we only support input color depths of 24-bit and 32-bit color. Grayscale and black-and-white scanned images are not supported; of course, counterfeiting grayscale or black-and-white images makes no sense. The image acquisition procedure is analogous to that discussed in MSDN’s WIA tutorial [13]; much of our WIA-related code is based on code found in the tutorial.

Once the image is fully acquired from the scanner, along with the resolution, color depth, and other important statistics, the communication with the scanner is done.

PC Implementation of Algorithms

When the reference database is loaded (during the start of the program), each reference image is converted to an inverted grayscale image (resolution assumed to be 100dpi) from whatever form it was before. Currently, our software supports 24-bit and 32-bit bitmaps. Once the inverted grayscale image is computed, the rotational angle of each reference image’s principle axes relative to the Cartesian coordinate system is determined using geometric moments as described in the algorithm section above. This process is performed only once, during program initialization, so we don’t count it towards our processing time.

Once an image has been acquired and is loaded into program memory, we must execute a preset procedure for detecting and extracting all objects of interest in an image. Because of the size of our reference images, we will only consider those objects whose each dimension is within the range of 100 and 500 pixels; other objects are considered to be too big or too small to be money and will not be tested. Each object, as it is extracted from the scanned image, is sent to the EVM for processing and classification.

The first step of our algorithm involves resizing the scanned image appropriate so that its resolution matches that of images in the reference database, i.e. 100dpi. In order to achieve this goal, we use the CopyImage() function and specify the desired width and height. This function uses whatever interpolation method deemed appropriate by the Windows operating system; we assume it uses nearest-neighbor interpolation, as described in the Algorithms section of our paper. Note that this yields poor resizing results, leading to poor recognition of objects; we recommend, at least for our demonstration software, that all images be scanned at 100dpi to maximize the effectiveness of our classification technique. Two solutions to this problem which we did not have the time to explore include better interpolation (such as bilinear or bicubic) or constructing a reference database with images at each resolution supported by the scanner. This process works on any color depth, so we use the original scanned image for this step.

The next two steps, morphological processing and blob-labeling, both operate on binary images. The EVM requires a grayscale image of each extracted object. These algorithms also assume that the background is zero and the foreground is nonzero, the opposite of what is the case in most scanned images. Therefore, we must convert the color image (24- or 32-bit) to both grayscale and binary, as well as invert the image, using the algorithms described in the Algorithms section. Our implementation for both uses the inverted form of the scaled color image, since the binary threshold does not have to be an integral value. In our tests, we determined that a threshold of 1.5 out of 255 properly binarized scanned images as long as the hue of the background was not significantly altered by scanning a thicker object (such as a book or even an ID card). A possible solution to this shortcoming would be to automatically detect the background threshold of each scanned image during runtime; however, this was not the primary focus of our project, and when one attempts to counterfeit money, he or she wants as little noise present in the scanned image as possible; the added hue takes the form of background noise.

Once the grayscale and binary versions of the inverted scaled image are available, our implementation performs morphological closing to attach foreground regions separated by the binarizing process. We perform morphological closing according to the algorithm described above, using a 5x5 square kernel for both erosion and dilation. This procedure modifies the binary image to reflect the result of the closing operation. The morphological processing code is based on the code from the IM image processing library [14].

Once closing has been completed, our preprocessing is almost complete. The remaining phase on the PC involves detecting objects in the scanned image and sending the individual objects to the EVM for processing. This procedure involves blob-labeling the binary image resulting from the closing operation performed above. Blob-labeling, as described in the Algorithms section, involves two passes through the binary image, and the use of a map data structure to store identifier mappings for joined regions. Although perhaps not terribly efficient, our iterative implementation of the algorithm essentially eliminates the possibility of stack overflows typical with a recursive implementation during the processing of large images. Each binary pixel is replaced with a 32-bit number identifying the object to which that pixel belongs (zero = background).

We can then extract each object from the blob-labeled image in at most N passes through the entire image, where N is the number of objects. To create our extracted object image for a particular object, we locate all the pixels that belong to that object, compute the bounding rectangle from those pixels, create a new grayscale bitmap the size of the bounding rectangle, and copy the grayscale pixel values from the inverted scaled grayscale image for each pixel described by the blob-labeled image as belonging to that object. We assume that objects are actually whole, so that every pixel in a row between two pixels belonging to an object belongs to that same object, so we copy whole pixel ranges into our extracted image.

Each extracted object image, along with its dimensions and size (in bytes), position in the original image, and other useful information, are saved in a simple data structure for extracted objects and placed at the end of a queue of images to be processed. The processing occurs asynchronously; the communication with the EVM occurs in a separate thread to be discussed in the next section.

PC-EVM Communication

The PC communicates with the TI C67 EVM using the API included in the file evm6xdll.h. This API enables the ScanView software to acquire a handle to the board, perform Host Port Interface (HPI) transfers in both directions, send and receive 32-bit (1 word) messages in any of four mailboxes, and use event-driven, non-blocking send and receive functions. Unfortunately, many of the systems do not have fully functioning event systems; in particular the event does not exist (OpenEvent() returns NULL!). To get around this limitation, we support both polling and events in our software. To reduce the overhead due to polling, we inserted Sleep(0) to give up the rest of our timeslice after we check to see if we've received a message.

Our PC-EVM communication is bi-directional and is synchronized by way of the mailboxes. To send data, we transfer the data to the address provided by the EVM using evm6x_hpi_write(). We signal to the EVM that the transfer is complete by putting a message that relates to what we just transferred in mailbox 1.

The EVM has only to wait for this message to ensure that the transfer is complete before using the data. To receive data, the EVM sends a message to the PC describing the transfer; the PC performs an HPI transfer using evm6x_hpi_read() to retrieve the data; the PC then sends a message to the EVM notifying it that the transfer is complete. The EVM has only to wait for this message to ensure that the transfer is complete before modifying the data.

Our implementation of the multi-step communication scheme used to send images to the EVM and retrieve the result from the EVM uses a finite state machine on both sides. The finite state machines and how they interact is described in the below figure. When not in debug mode, the PC will timeout if the EVM does not respond to a message within 30 seconds.

The EVM procedure is described in more detail in the section below.

EVM Implementation of Algorithms

Upon receiving an extracted object, the EVM computes geometric moments and the corresponding rotation angle from using the algorithm described in the Algorithms section. This processing can be performed in parallel with requesting the first reference image, to save some time. This is the only processing performed on the EVM independent of reference images; the other algorithms will be performed for each reference image.

Once the desired reference image is received, the EVM computes the cosine and sine of the angle of un-rotation. This angle is the difference between the rotation angle of the extracted object and the rotation angle of the reference image, which was computed beforehand on the PC. We compute the size of the un-rotated image by mapping each corner of the extracted object box to the coordinates in the un-rotated coordinate plane. The extracted object is un-rotated by this angle, using bilinear interpolation and the algorithm described earlier in the paper. The image un-rotation code is based heavily upon the code in the IM image processing library [14].

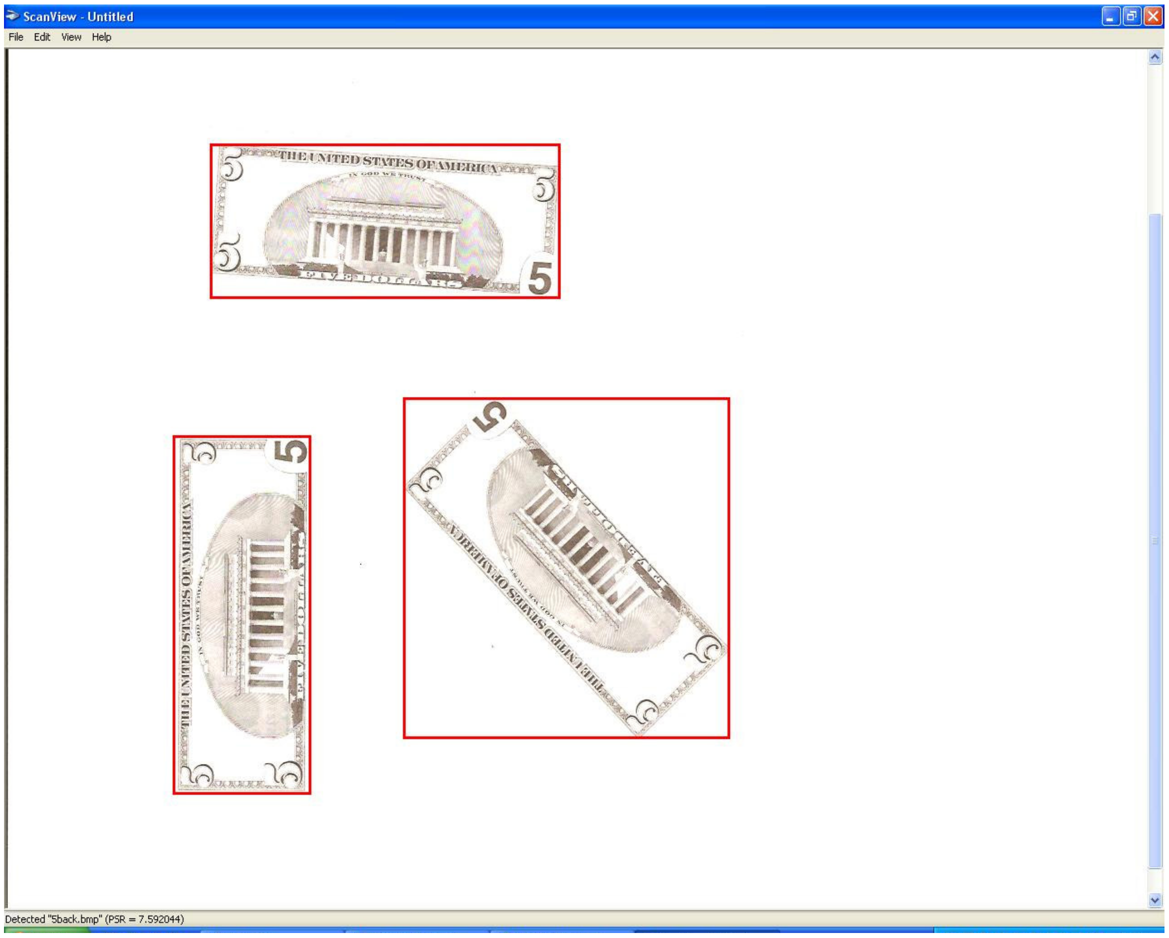
Once the image is un-rotated, so that the principle axes of both the extracted and reference images are aligned, we extract the edges from both images using the Sobel method. This method involves taking the gradient in both directions and identifying gradient values above a certain cutoff as edges. This process results in binary images of both the reference and un-rotated extracted objects. Our implementation takes advantage of as much parallelization as possible, since we use only multiplications and additions to compute each gradient.

Once we have the binary images describing the edges of both images, we compute the linear correlation of both images over a limited range of offsets, using code similar to that in Lab 3. This range of offsets is centered at the offset that corresponds to the centers of the two images being aligned. In all, 1296 pairs of x and y offsets are used to generate a 36x36 matrix of correlation outputs. If the correlation peak is above a certain threshold dependent on the reference image, these results are used to compute a PSR value according to the algorithm described in the Algorithms section. If the resultant PSR is also above a certain threshold, also dependent on the reference image, and by an amount proportionally greater than any previously calculated PSR for this extracted object was above the other related reference image's PSR threshold, the object is classified to be that particular reference object. We repeat the process of edge extraction, linear correlation, and PSR computation on the extracted object, rotated by 180 degrees, since we cannot be sure just from our geometric moments that the image rotation angle is periodic by 180 degrees, not 360 degrees (principle axes are straight lines that pass through the center of rotation).

Finally, these algorithms (except for the geometric moment computation) are repeated for each reference image. Once the EVM receives a reply from the PC stating that there are no more reference images to test, the EVM sends a message to the PC describing the classification results. The PSR associated with the reference image the extracted object most closely resembled is also sent to the PC. If no image matched adequately, the EVM returns a zero.

After receiving this information for each extracted object, the PC will conclude whether or not the user can scan or copy the scanned image. The presence of one prohibited image is sufficient to block copying.

ScanView – the GUI



Detected "5back.bmp" (PSR = 7.592044)

Figure 19: ScanView Software

In support of our algorithm development, our GUI blends ease-of-use and functionality in one sleek, professional-looking product. Although not as full-featured as most scanning programs on the market (our software wouldn't even hold a candle to Microsoft Paint when it comes to editing, etc.), our program does allow the user to scan images and copy sections of images to the clipboard in standard Windows bitmap format. Each prohibited object that the system detects has a red rectangle drawn around it; a simple mouse-over a rectangle will cause information about the prohibited image (what reference image it resembles, and its PSR) to appear in the status bar. When an image containing prohibited content is detected, our software alerts the user with a message box and a theme related to our project title, M. C. Hammer's "Can't Touch This!" playing softly in the background.

Our GUI, upon loading, will notify the user if it could not find a WIA-compliant scanner or connect to the EVM. Although one can ignore both warnings, doing so will limit the program from working properly and is not recommended except for debugging purposes. A message may also appear if the EVM event handle could not be acquired; this message is simply for the user's information and will have little impact on the function of the software.

Once these tests are passed, the user can enter the path to a reference database file (this can only be done upon loading the program; there is no way to load a reference database beyond this point). This file is a Windows-compatible text file that specifies one per line the filename to a reference image, and the associated peak value and PSR cutoffs. The three fields are delimited by commas. You must enter a value in all of these fields; failure to do so may cause undesirable consequences in the operation of the software.

Once a reference database has been selected, and all the files within it have been parsed, the program begins. If there is a scanner available, the program begins the scanning process, causing the standard popup dialog for scanning to appear. Of course, the user can also begin a new scan by selecting the File -> Acquire from Scanner… menu option. The user can also load an image from a Windows bitmap (.bmp) file using the File -> Open… menu option; however, these images have no scaling information and are therefore assumed to already be at 100dpi.

The File -> Save As… operation has not been implemented, being an extraneous feature to our program; this menu option is always disabled. The Edit -> Copy operation can copy the selected region to the Windows clipboard; the user can then paste the image into another program; this menu option is disabled when there is no selected region. The Edit -> Select All operation chooses the entire scanned image as the selected region; it is disabled if there is no image to select. Otherwise, one can select a region manually, pressing down the left-mouse button, dragging the mouse to form the selection rectangle, and depressing the left-mouse button to complete the selection. The View menu contains an option enabling the user to show/hide the status bar, normally visible at the bottom of the screen. The Help menu contains an About ScanView… menu item, which displays basic copyright information for our application. To exit, one can simply click the “X” at the top right hand corner of the window or by selecting the File -> Exit command.

Note that exiting the program will terminate the code running on the EVM. The user must restart the ScanEVM software running on the EVM before running ScanView.

Implementation Issues

During the initial development and testing phases, several problems became apparent. The most immediate problems dealt with memory – how could we cram all the data we would need onto the EVM? It seemed apparent that there was no way to store an entire FFT on the EVM, since we are limited to blocks 4 megabytes in size; a 500x500 image, the FFT of which would require each pixel to be replaced with two floats, would consume 2 megabytes minimum. Since the order of the FFT necessary is approximately the sum of the sizes of the two images we are correlating, the size is actually on the order of (500+500)x(500+500), or 8 megabytes. Assuming that our un-rotated image can be even larger than our extracted object, we have already far exceeded the size of an entire memory bank with just one FFT. There would be no way to get one FFT into memory on the EVM, much less two! Therefore, it became apparent that our correlation would have to be performed in the spatial domain instead of the spatial frequency domain, which is definitely much slower, but more suitable to our memory constraints.

Of course, performing a spatial correlation means that our algorithm will run much, much slower than originally planned. Our desired overall runtime to process an entire scanned image was under 10 seconds, ideally something comparable to the time it takes to acquire an image using an average scanner. However, our initial tests suggested that our process took several minutes just for one correlation. In an effort to speed up the correlation, we only perform the correlation for a limited range of offsets. Nevertheless, the linear correlation proved to be a horrendous bottleneck.

In addition to the mundane problems with memory and speed – where we simply needed to make the program take less memory and run faster – we also encountered issues with our scanner. Although our tests showed that for scanning money, the background was approximately white in all of our tests, some tests showed that scanning objects thicker than normal paper left sufficient spacing between the scanner’s glass and the cover to add reddish hues to our background, thus dramatically altering the background intensity. We never were able to overcome this issue; however, a better binary threshold algorithm than a constant threshold intensity should ameliorate this problem.

Another issue involved the placement of images too close to each other, or even overlapping each other. Blob-labeling is not a particularly intelligent algorithm, requiring an object to be enclosed by background on all sides, and simply cannot handle adjacent or overlapping objects. After much discussion, we bounded our problem, realizing that a serious counterfeiter would hesitate to cover part of the money he or she is copying or place it too close to another object and potentially have them interfere with each other in the image.

In the interests of the goals of the project, we focused mainly on the memory and speed issues. Our efforts to optimize our design, in particular, our linear correlation, are described in the next section.

Optimization

Initially, for debugging reasons, we developed our EVM software with no compiler optimizations, especially to prevent the "optimizing away" of variables. Therefore, no loop unrolling, parallelization, or any other automatic optimizations were performed.

Once we were sure of our code's correctness, we turned on file level (-o3) optimization, which enabled the compiler to heavily parallelize the innermost loop in our linear correlation code. With compiler optimizations built in, we were able to have seven iterations of the innermost loop execute in parallel.

While parallelism was great, we realized that we were processing data directly from DRAM, an excruciatingly slow process. While simple, this clearly would not work. As a solution, we created buffers in internal (on-chip) memory and used memcpy() to transfer blocks of data between them. This gave us noticeable speedup.

However, the code was still too slow. We traced the slowdown to the use of memcpy() to transfer data from the images in external memory to the buffers in internal memory. Since memcpy() is a blocking operation, we were losing time to data transfers while we might have instead been processing. Using Direct Memory Access (DMA), we were able to decrease the amount of time lost to data transfers; however, we were still wasting time that we could spend processing.

Through the use of asynchronous DMA and ping-pong buffers, we were able to achieve tremendous speedup, since we were not waiting for the data to be transferred until we were ready to process it. Our initial transfers are still essentially synchronous, since we need the first blocks of both images to start our correlation; however, we can perform the next transfer, writing the data to other buffers, while we are processing the data in the current buffers. Then, when the transfer is complete, we process the data in the other buffers, while we transfer data to the first buffers. We alternate processing and transferring until all the data has been processed. This way, we can (mostly) eliminate the time taken by transfers from our correlation.

The results for each of these steps are published below:

External memory (no optimization): 5,866,260,572 cycles, or 38.72 sec.
External memory (with optimization level –o3): 4,552,795,945 cycles, or 30.05 sec.
Internal memory, using memcpy(): 1,255,665,124 cycles, or 8.29 sec.
Internal memory, using synchronous DMA: 187,891,628 cycles, or 1.24 sec.
Internal memory, using asynchronous DMA and ping pong buffers: 120,503,763 cycles, or 0.795 sec.
Overall speed improvement: 1 correlation is approximately ***48.7 times faster!***

Data Analysis

Before we conducted our data collection, we compiled several databases: a reference database for classifying detected objects, a training database for estimating parameters, and a test database for testing our system. Each database contains images, both front and back, of play money. We use play money to avoid the legal issues involved with digitizing federal reserve notes. All images are in color (converted to grayscale and binary where required) and have a resolution of 100dpi.

Our reference set consists of one image for the front and back of each bill ($1, $5, $10, $20), in total, 8 images. The reference images are completely un-rotated and have (approximately) no noise.

Our training set consists of several images of each bill, at different rotation angles. This set is used to assist us in determining our PSR thresholds and quantifying the required rotation sensitivity. This set consists of 23 images.

Our testing set consists of several images of each bill, not included in the training set, at different angles of rotation. In addition, the testing set contains images with multiple objects, a mixture of play money and other objects commonly scanned (photos, business cards, etc.). This set consists of 22 images.

The first test we implemented was utilizing the peaks of an objects correlation with our database to aide in our classification system. We believed that the peaks for each object would be sufficient enough to discriminate between positive classifications (a match for the object in question) and negative classifications (the other bills in the database, as well as other images). Figure 20 shows the results from our testing.

Figure 20 shows that there can be negative classifications with a higher peak than a positive classification, which would make it nearly impossible to determine a useful threshold between the two. Therefore, we decided to look elsewhere for a better test that would help our classifier.

To improve on our previous test, we decided to see how effective peak-sidelobe ratios are in our classification system. The PSR implementation takes into account not only the peak of the object's correlation with our database, but also those of the surrounding neighborhood. Figure 21 shows how effective PSRs are in discriminating between positive classifications and negative classifications.

Figure 21 shows a clear delineation between the positive and negative classifications, which would make it easier to determine a threshold between the two. Since the PSR implementation seemed to work well, we decided to keep it in our classification system.

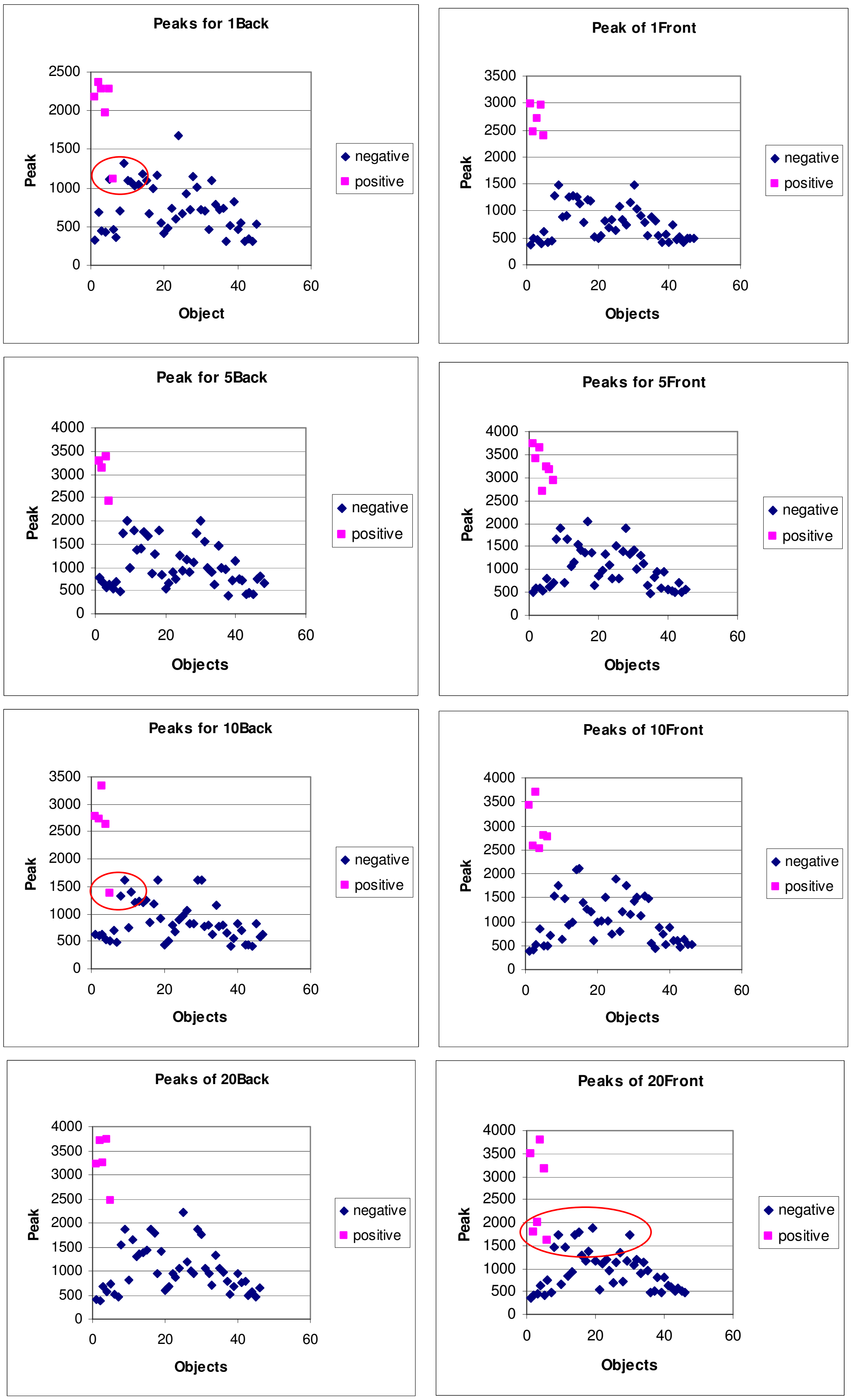


Figure 20: Correlation Peaks against Reference Images

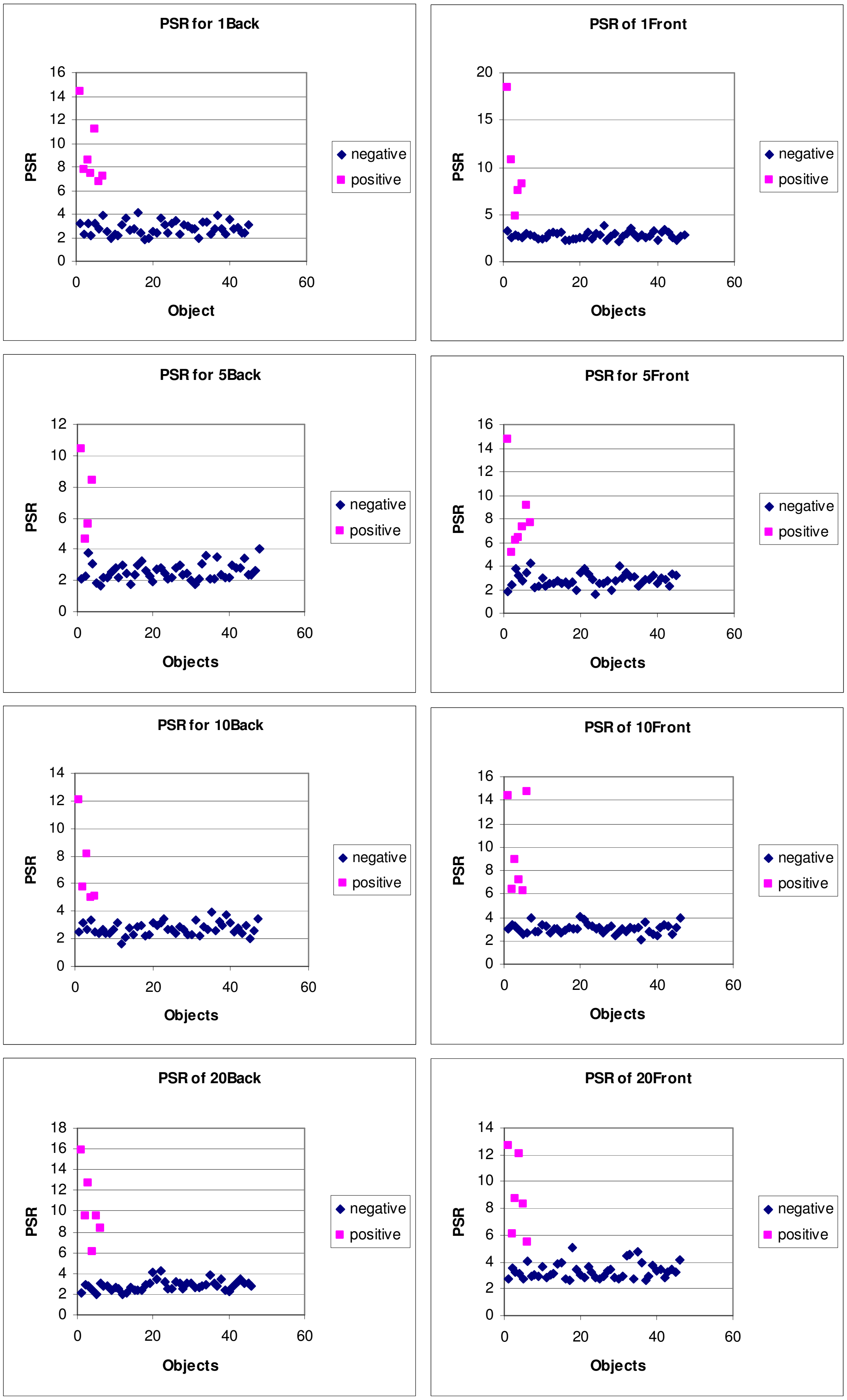


Figure 21: Correlation PSRs against Reference Images

Our next problem was figuring out how interpolation affects our PSR calculations. We created a MATLAB script to show how rotating an image, then un-rotating the image at the same angle would dramatically decrease our PSR values. The following graph shows the data we obtained in digitally rotating and un-rotating our object using bilinear and nearest neighbor interpolation.

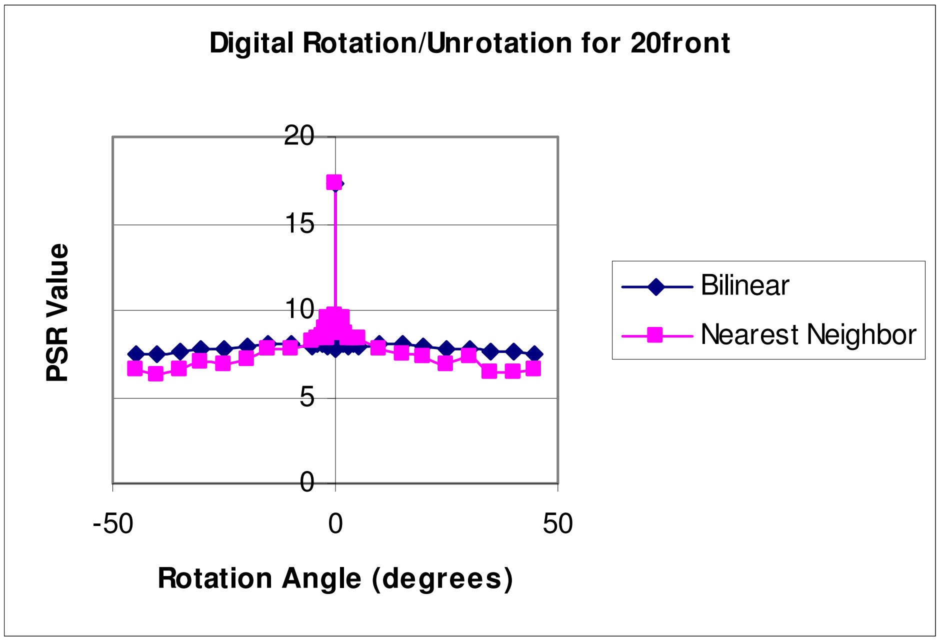


Figure 22: Digital Rotation/Un-rotation Results

Figure 22 shows that nearest neighbor interpolation is only useful for small angle rotations; unfortunately, we cannot depend on the object rotations in our system to be of small angles. Therefore, we decided to use bilinear interpolation for our digital un-rotation.

In order to determine the rotational sensitivity of our correlation filters, we used MATLAB to rotate the reference image by a variety of angles and correlate each rotated image against the original reference image. By placing the PSR threshold computed in the PSR experiment performed beforehand, we can determine how accurate the geographic moments estimation must be to correctly classify the result. The cutoff PSR for a particular reference image is determined according to the experiment performed above (see PSR test) and is used to determine the lowest acceptable PSR for that reference image. The maximum allowable error in rotation angle is the angle corresponding to where the PSR plot intersects the cutoff PSR.

Figure 23 demonstrates that the PSR degrades rapidly as the object rotates away from zero degrees (ideal). We computed that for the back of the $10 bill, the object must be rotated within +/- one degree of the ideal rotation angle to give the correct result when correlating and computing the PSR.

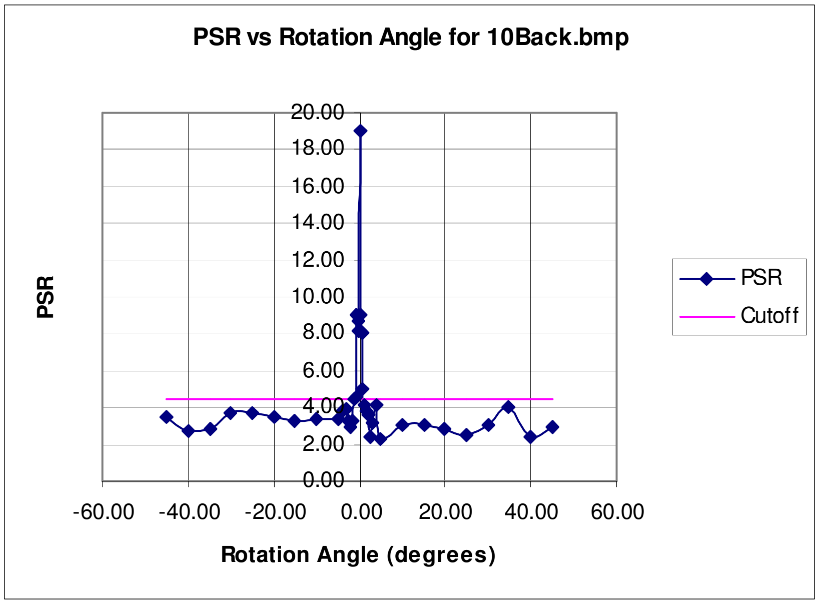


Figure 23: Rotational Sensitivity of PSR

Towards converting from a grayscale to a binary image, we must compare the grayscale intensity as computed by the NTSC formula to a cutoff intensity (out of 255.0). This next experiment assisted us in determining an acceptable background threshold. In MATLAB, we took several images and converted them to black and white according to any of several threshold values. After experimenting with several images, we determined the threshold to be one constant value over all input images, approximately 1.5.

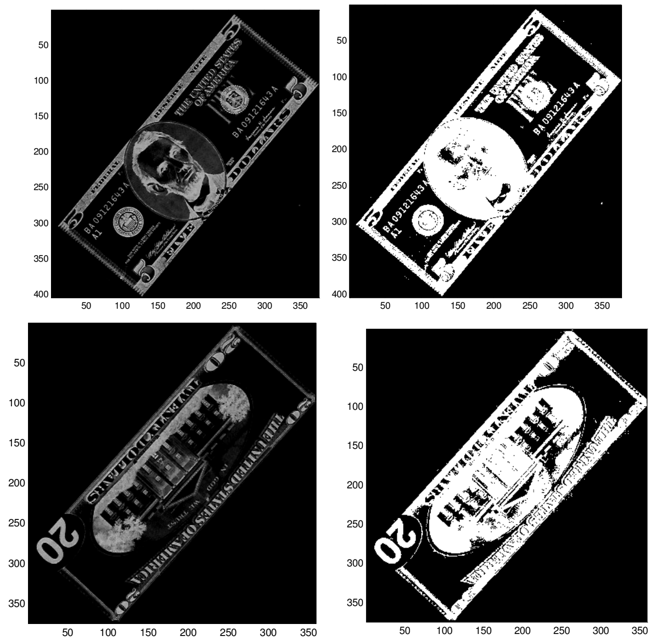


Figure 24: Binary Thresholding of Various Images

In addition to the binary threshold, we also had to determine the size of the structural element used by our morphological closing algorithm. The structural element must fit two criteria: it must be large enough to effectively reunite objects separated during binary thresholding, and it must be small enough not to unite objects far enough away from each other that we would consider them distinct. We experimented with several different kernel sizes:

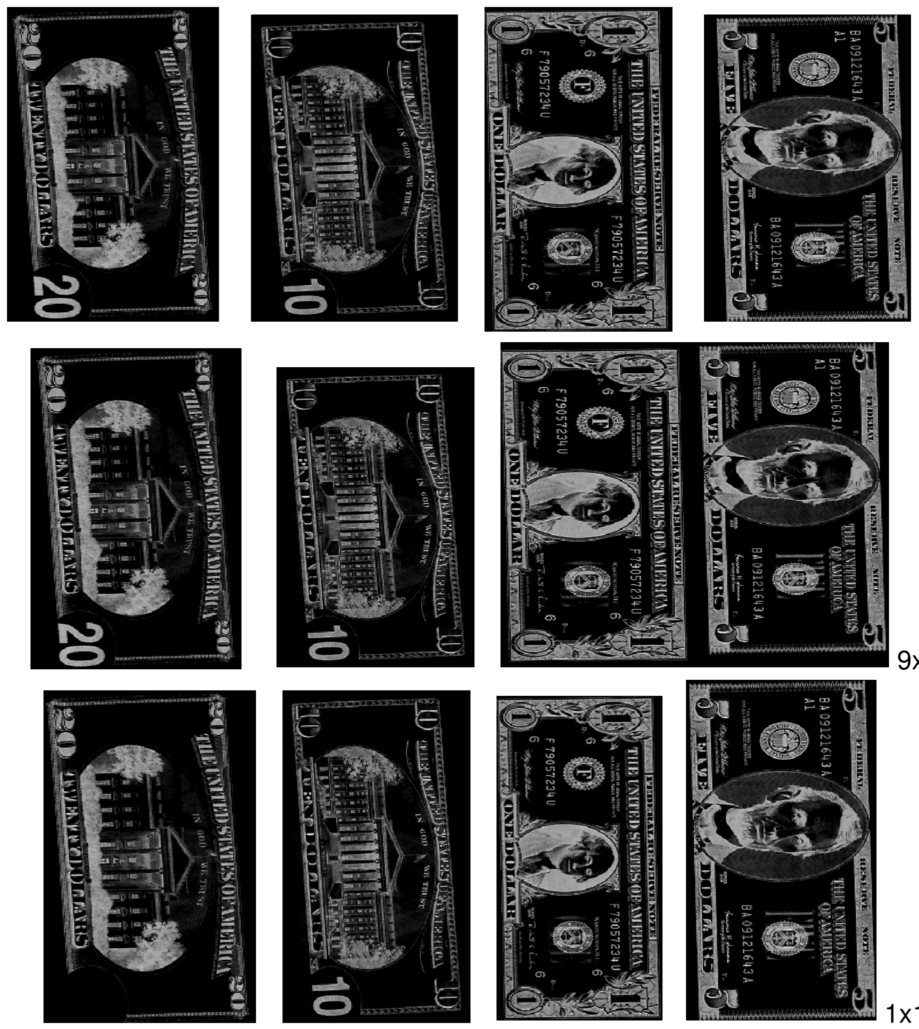


Figure 25: Results of Morphological Closing using Different Structural Elements

Evidently, the 5x5 structuring element fits both criteria in the image we chose. The 9x9 element successfully reunites separated objects (i.e. the number on the bottom right of the back of the $20 and $10 bills) but joins objects that are definitely distinct (the $1 and $5 bills). The 1x1 structuring element does not successfully reunite the number on the bottom right of the back of the $20 bill.

Once we completed these experiments, we were ready to determine whether or not our entire system worked. We ran our set of test images through our software and recorded which images were handled correctly and recorded the results.

Out of the 43 correlations from our test database, there were only two false misses. This shows that our system is approximately 4.65% inaccurate, but it correctly biases towards allowing money through rather than preventing legal images from being scanned or copied. We demonstrated several images, containing both money and legal objects. Tests during our demonstration confirmed the expected functionality; however, we noted that one test involving a group member's ID card failed to produce either correct or consistent results. On two out of three attempts, the card was classified as either the front of the $5 bill or the back of the $10 bill. We found that although the PSR was abnormally high for these false hits, the magnitude of the correlation peak was relatively small; consequently, we modified our software to validate both the correlation peak and the PSR during image classification. This removed the false positives without demonstrably affecting the efficacy of our software.

Conclusion

From our tests, we determined that our system functions properly in the vast majority of cases, erring only on the side of caution, biasing errors towards incorrect misses instead of incorrect hits. However, we note that the speed of our software was not suitable for a real-world implementation. Thus, we would recommend that a real-world implementation possess enough high-speed memory to cache an image of the maximum expected size, to eliminate the need to transfer data frequently. In addition, a real-life implementation would reside on the imaging device itself. Therefore, the only PC-side software necessary would be the standard scanner interface; the details of the implementation of the prohibited-image detection system would be hidden from the user. Although this project would have to clear several hurdles before being ready for distribution to manufacturers, we have adequately demonstrated that the concept is sound, and that such a detection scheme can be implemented on the imaging devices themselves.

References

[1] *Central Bank Counterfeit Deterrence Group*. Available online http://www.rulesforuse.org/. Accessed 9 Oct, 2005.
[2] Ulbrich, Chris. “Currency Detector Easy to Defeat.” *Wired News*. Available online http://www.wired.com/news/print/0,1294,61890,00.html. Accessed 8 Oct, 2005.
[3] Resende, Patricia. “MediaSec seeks more than the eye can see.” *Mass High Tech*. Available online http://www.masshightech.com/displayarticledetail.asp?art_id=59816. Accessed 9 Oct, 2005.
[4] “Know Your Money.” *United States Secret Service*. Available online http://www.treas.gov/usss/money_illustrations.shtml. Accessed 11 Dec, 2005.
[5] Pearson, Chris; Tsao, Amy; Yu, Christina; Theisz, Matthew. “Where’s the Ball?” 18-551, Spring 2004. Available online https://blackboard.andrew.cmu.edu/courses/1/F05-18551/content/_108177_1/stheBallfinalreport.pdf. Accessed 6 Nov, 2005.
[6] Baliga, Avinash; Bang, Dan; Cohen, Jason; Schwicking, Carsten. “Face Detection for Surveillance.” 18-551, Spring 2002. Available online https://blackboard.andrew.cmu.edu/courses/1/F05-18551/content/_108213_1/Group2Final.pdf. Accessed 6 Nov, 2005.
[7] “Technical Solutions Solution Number: 1-1ASCU.” The Mathworks. Available http://www.mathworks.com/support/solutions/data/1-1ASCU.html?solution=1-1ASCU. December 4, 2005.
[8] Umbaugh, Scott E. “Morphology Overview.” *Computer Vision and Image Processing*. Prentice Hall PTR, 1998. Available online http://zone.ni.com/devzone/conceptd.nsf/webmain/8CA8DE2E8881C1AB8625682E0079CE74. Accessed 11 Dec 2005.
[9] Gonzalez, R. C.; Woods, R. E.; Eddins, S. L. *Digital Image Processing Using MATLAB®*. Pearson, 2004.
[10] Hu, Ming-Kuei. “Visual Pattern Recognition by Moment Invariants.” *IEEE Transactions on Information Theory*. IEEE, Feb, 1962.
[11] Yi Li; Zhiyan Wang; Haizan Zeng. “Correlation Filter: An Accurate Approach to Detect and Locate Low Contrast Character Strings in Complex Table Environment.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. IEEE, Dec, 2004.
[12] Savvides, M.; Kumar, B. V. K.; Khosla, P. “Face Verification using Correlation Filters.” Available online http://www.ece.cmu.edu/~kumar/Biometrics_AutoID.pdf. Accessed 11 Dec, 2005.
[13] “Windows Image Acquisition (WIA) 1.0.” Microsoft. Available http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wia/wia/overviews/startpage.asp. November 16, 2005.
[14] “IM – An Imaging Tool.” Tecgraf – Computer Graphics Technology Group. PUC-Rio, Brazil. Available http://www.tecgraf.puc-rio.br/im/. November 16, 2005.