I Spy (and Follow)

18-551, Fall 2005
Final Report

12/14/2005

Group 2:
Khalid Harun (kharun@)
Ramu Bhagavatula (rbhagava@)
Joe Leonelli (jleonell@)

Table of Contents

Section 1. Our Solution

Many places in today's society are seeing an increased reliance on electronic devices and their ability to remove 24/7 human monitors. In the fields of aviation and automotive travel, the importance of constantly monitoring a landing strip or a busy roadway is more important than it has been in the past. A DSP system that is able to detect, identify, and track large objects as they cross a fixed field of view (FOV) is critical to maintaining safe and efficient transportation solutions.

For our project, we will be designing such a system that will be created to detect, identify, and track radio controlled (RC) cars and tanks in the 18-551 lab from a fixed view (on top of the 18-474 cabinet) at a fixed angle. In this scene, the RC vehicles will be moving while the camera remains fixed. However, due to processing limitations, we will not be able to track the targets in real-time. Instead, we will capture a time sequence and then use our DSP system to detect, identify, and track designated targets using correlation filters, the Central Slice Theorem (CST), and Hidden Markov Models (HMMs). The use of the CST and HMMs are unique to 18-551, and should provide a new perspective to the ATR problem. Ideally, our system would be extended and improved to deal with the more complex scenarios described below.

When managing the take-off queue of a busy airport, the volume of aircraft moving through a fixed location can be quite significant. This environment can increase the possibility of aircraft misidentification by the traffic controllers, miscommunication between a pilot and the traffic controllers (regarding which take-off strip to use), and unintentional delays as traffic controllers 'lose track' of aircraft already in the queue. By placing the detection, identification, and tracking roles under the control of a semi-autonomous DSP system, traffic controllers can better utilize their time by efficiently allocating take-off strips. Their decisions will be based on data collected by several of these identical DSP systems, each set up on a different aircraft strip, taking data from a fixed camera at a single view. This will allow the DSP system to deal with detection and identification of aircraft on only a scaling dimension, rather than also on a rotational dimension. This means that on the training phase of the DSP system, data will be provided showing the possible tracking target at different sizes, but all at the same perspective. However, in order to make the system more adaptable, data may be collected of the same tracking target positioned at different orientations to make rotated detection possible.

In the field of automotive travel, a DSP system as described above would serve several purposes. Primarily as an extension of the police force, this system would be used to track and follow a dangerous target down a long roadway, through a busy intersection, or even in an urban setting. This ability to track vehicles, during car chases and for stolen vehicles, is of extreme importance to highway and city police teams. Cameras would have to be set up at key points along a freeway, at intersections, or positioned above the desired streets, but would be limited to a fixed FOV. It is possible that this functionality could be extended to track large-scale traffic patterns in a city, as well as tracking speed limits and traffic violations.

Ultimately, this type of device will be of extreme importance to the military. As today's military is increasingly moving towards a more automated force of highly specialized vehicles and highly trained operators, these vehicles will have advanced controls that are used to perform basic tasks that would otherwise consume valuable man hours and resources. In order for these controls to be effective, they must be able to automatically and independently find potential targets and track their position across the FOV. As a result, operators of offensive vehicles can focus entirely on directing the vehicle and taking necessary actions against any detected and confirmed threats. To be of maximum value, the DSP system would have to integrate both the scaling solution and the rotation solution into a single device that could be attached to a movable camera and assist in tracking and identification of military aircraft, tanks, and naval vessels.

Section 2. Prior 551 Work

The only groups to pursue identification and detection design projects were *Face Verification for ATM Access* (Group 2, 2003) and *Automatic Target Recognition in Synthetic Aperture Radar Images* (Group 2, 2000). The 2000 group used feature extraction along with a classifier, which is very different from our proposed project, which uses correlation for the identification and detection. The 2003 group uses correlation filters for the same purpose as us, but they have no tracking capabilities in their system. Additionally, they were not forced to implement another detection method, such as the Central Slice Theorem, in order to complete their DSP system. However, our system was required to use the CST in order to significantly reduce the number of computations (at the cost of significantly increasing our processing speed, making real-time processing near impossible). The details and necessity of the CST will be demonstrated in *Section 4. Algorithms*.

Beyond these identification and tracking design projects, several previous projects addressed the topic of object tracking. These four projects, *Object Tracking via Optical Flow in Video* (Group 15, 2000), *Face Detection for Surveillance* (Group 2, 2002), *Where's the Ball?* (Group 2, 2004), and *Lights, Camera…Action!* (Group 12, 2004). However, none of these projects required the use of correlation filters, since most were dealing with human faces, in which blob coloring, frame differencing, and optical flow are more reliable and simpler to implement. Since we are dealing with both rotation and scale changes in a highly variable environment (the lab has significant noise due to the lighting, number of other objects, etc), none of these three methods would be appropriate or successful for our solution. Thus, we must use correlation filters to more accurately render the position and identity of our targets. We also use Hidden Markov Models to help predict future detections, using a Gaussian probability to reliably track the targets.

Because we are introducing new algorithms to solve an expanded problem that integrates tracking and identification/detection, our project is certainly unique.

Section 3. Data Sets

For our project we acquired five radio-controlled (RC) models to attempt to detect, identify, and track. Taking into consideration the real-world application of our project for both the military and civilian applications we decided to acquire both military and civilian related RC models. Military related models included an M1A2 Abrams battle tank, a German Leopard tank, and a "Nerf" tank. The civilian models included a Smart Coupe and a Mitsubishi Lancer. Respectively we designated these classes as Classes 1 through 5.

Section 3.1 Training Data

Our primary concern in assembling our training set was to gather a set of data that sufficiently generalized all rotations and a range of scales for each of the targets. The primary problem with this requirement is that there naturally exist an infinite number of rotations. The same is true even for a limited range of scales. However, we cannot possibly hope to capture such a set of data accurately. Also, it is an unrealistic expectation that we will be provided such of data; particularly in our application where our tool is designed to potentially track the vehicles of enemy forces. Therefore, we must limit the size of our training set to the bare minimum that meets some specified threshold of performance. For this we must establish this threshold based on performance expectations w.r.t. our detection scheme (expected probability of detection/miss rate) and w.r.t to our identifications scheme (probability of correct identification).

We first considered the five potential targets we had. The targets psycho-visually (to the human eye) are fairly different to each other except in one case. The models of the M1A2 tank and the German Leopard Tank present similar profiles but do have different coloring schemes and camouflage patterns. The difference in coloring schemes is greatly reduced when compared in the grayscale range but still exists to some degree. The difference in camouflage patterns is far more significant since regardless of colormap it is still present and significantly contributes to frequency-domain profile of each target. With these things in mind, we started our training data collection.

Our training set capture setup consisted of the USB Logitech QuickCam Pro 4000 connected to an IBM G41. We used a camera capture tool for Matlab called vcapg2.dll [1]. This is a Matlab community developed tool that can be found on the Mathworks Central File Exchange. This tool allowed us to capture imagery from a USB enabled capture device directly into the Matlab workspace. Other features of this tool include the ability to specify the colorspace, resolution, zoom, pan, and tilt of the camera where available. In our setup we used the YUV colorspace, a resolution of 320×240 pixels, and default zoom/pan/tilt parameters. Once in the Matlab workspace, we could convert the YUV colorspace images into grayscale images.

Our initial training set consisted of twenty-four angular views of the targets at close range (approximately one yard along line of sight), constant elevation, and depression angle. This resulted in images being taken at every fifteen degree increment views. In order to maximize the profile of the targets, we placed them on white poster board (non-reflective side). This would aid us in the pre-processing of the training data that will be discussed later on. We made sure to note the time of day and place at which the data was take which was between the hours of 7:30 and 9:00 PM and in the 18-551 lab. The importance of this is reflected in the fact that the lighting conditions of the environment strongly dictate in which lighting conditions tools trained with such data will perform reasonably in. Figure 1 are example images demonstrating the setup under which our grayscale training data was generated under both gray and jet colormaps found in Matlab. The jet colormap image is used to demonstrate the isolation/difference of the target from the background. It is also important to note that this image is unprocessed except for conversion from YUV to grayscale.

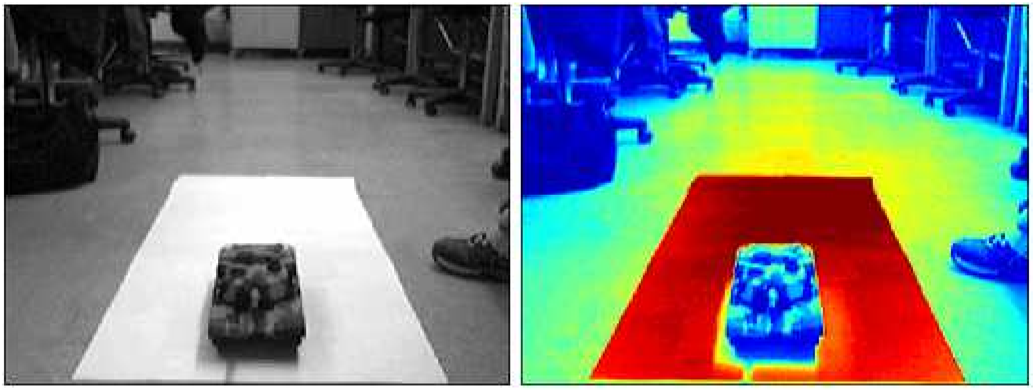


Figure 1: Example on unprocessed grayscale M1A2 training data image (Left) Grayscale colormap (Right) Jet colormap

Once all twenty-four images of each of the targets were acquired, they underwent a set of pre-processing to massage the data into a usable form. Pre-processing of the data began with cropping of the data down to a constant size and positioned bounding box among all the data. This was dictated by the largest horizontal and vertical views of any of the target i.e., the size of both the M1A2 and the German Leopard when they presented forward, backwards, and side profiles. The dimension of this bounding box was determined upon visual inspection to be 205 × 120 pixels. At this point, the targets were now surrounded by only white pixels which are relatively high-valued; grayscale pixel values run from 0 (pure black) to 255 (pure white).

It would have been possible to train on this set of data, but there is a significant problem with doing this. This problem is that difference between the white background and the relatively dark colored targets creates numerous high frequency components in the training data's frequency profile. Our goal in creating this training set is to accurately represent our target's profiles in the spatial and frequency domains. If we introduce previously unrepresented high-frequency components we are artificially changing the data. The best solution to this was to replace the surrounding background with the mean pixel value of only the target. By doing this, we are only contributing to the DC frequency component of the data. The significance of this will be detailed later in the Algorithms section. In order to isolate only the target, we first raised the value of each pixel to a power of 5. This exaggerated the difference between the target and the background. From this we were able to derive a binary mask for the target by first applying a threshold where all pixels above a certain value where marked. These marked pixels made up the 0 values in our mask while the non-marked pixels made up the 1's. Following this masking, a series of erosions and dilations were applied using block-shaped structuring elements. The result of this filled in any accidentally removed pixels from the target itself and any outlier pixels.

At this point we had an image which consisted of mostly 0's except within a tightly bound area around the target only. The mean of the non-zero values (i.e. the target) was calculated and used to replace all the 0 values. The image was also padded up to a square size using the mean value. Figure 2 is the result of all this pre-processing on the image shown above in Figure 1. This pre-processing was applied to all twenty-four images of each target's training set. Figure 3 shows the result of applying this pre-processing to the training set of the M1A2.

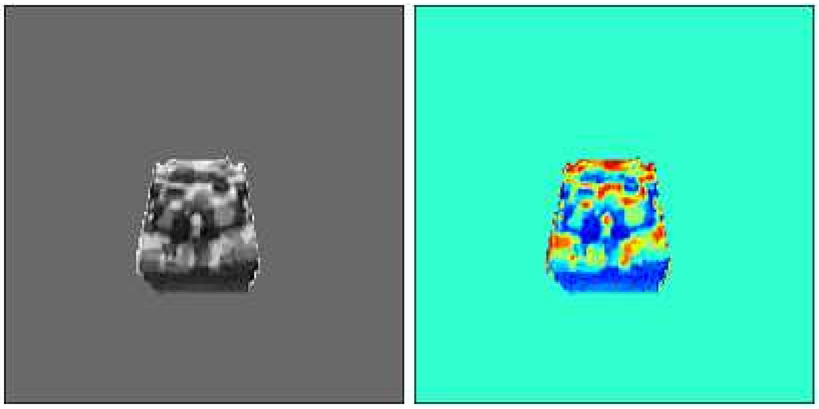


Figure 2: Result of pre-processing on image in Figure(1) (Left) Grayscale colormap (Right) Jet colormap

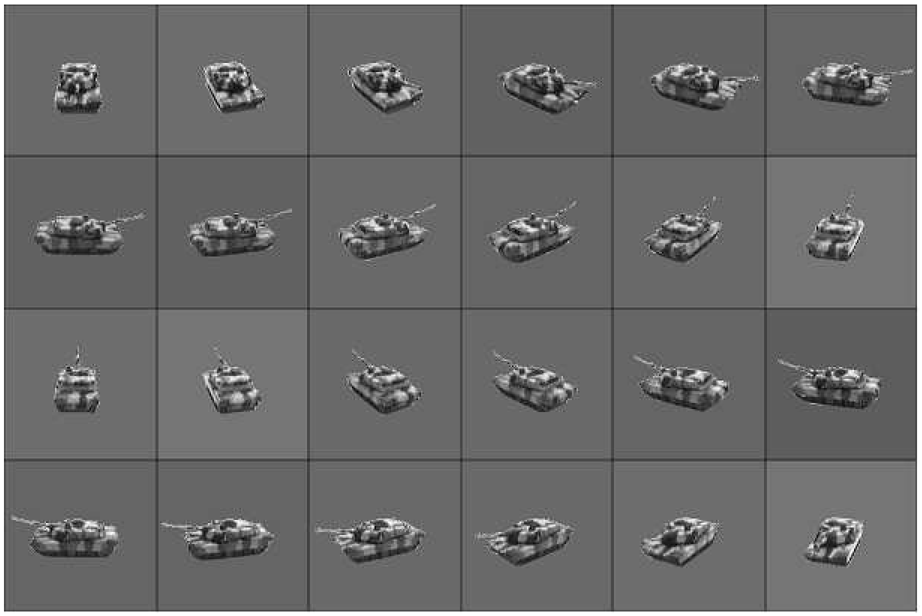


Figure 3: Pre-processed training data for M1A2

Section 3.2 Test Data

Testing data was captured in the same setup as the training data. The first set of testing data was comprised of one-degree increment angular views of the targets taken at the same range and depression angle as the training data. As a result, each class had a testing set consisting of 360 images taken in constant conditions. This testing set would be used to test the identification capabilities of the correlation filters that will be detailed later. The second set of testing data consisted of approximately two minute long video sequences. For each class/target we captured two sequences that consisted of only that target moving throughout the scene performing different maneuvers. This data was used in our Matlab simulations to test the effectiveness of out algorithm. We also captured an additional fifteen sequences that consisted of multiple targets moving throughout the scenes. This data was used to test the multiple target tracking capabilities of our algorithm. We focused the content of this data to contain combinations of classes that presented the strongest confusers for the algorithm. This meant placing both the M1A2 and the German Leopard in a set of sequences together, and in other sequences placing the Smart Coupe and the Mitsubishi Lancer together. The Nerf Tank did not prove to be a significant confuser to any of the classes and was thus included sparsely. We intentionally maneuvered the targets in these sequences to cross each others paths in order to present the possibility of occlusion and target loss. These sets of testing data aided us immensely in assessing the effectiveness of our approach.

Section 4. Algorithms

As stated before, our algorithm is strongly based on work done by Ryan Kerekes, Balakrishnan Narayanaswamy, and Mike Beattie [2]. The primary difference between our approach and theirs is in implementation. When compared to the power and precision of PC using Matlab, the C67 EVM is a severely limited piece of hardware. Incapable of doing two-dimensional FFT's of significant size in floating-point double precision as can be done in Matlab, the EVM forces us to carefully consider many factors when it comes to both precision and as a direct consequence size of data. Such considerations strongly influence how directly we can implement previously done work without introducing new aspects. Since our task is to implement a tracking and identification algorithm, we must consider three tasks. The first task being detection of potential targets, followed by identification of these detected targets, and then tracking of these targets. We will detail each task separately. However, before we can do that we must first go into some detail above certain tools/algorithms we use in our overall approach.

Section 4.1 Correlation Filters

Correlation filters [3] are template-based classifiers that when correlated with an image result in a correlation plane. The correlation plane *C* measures the correlation between the filter and the image. Correlation of a class-specific filter with authentic and imposter data yield very different correlation planes. Figure 4 demonstrates this difference.

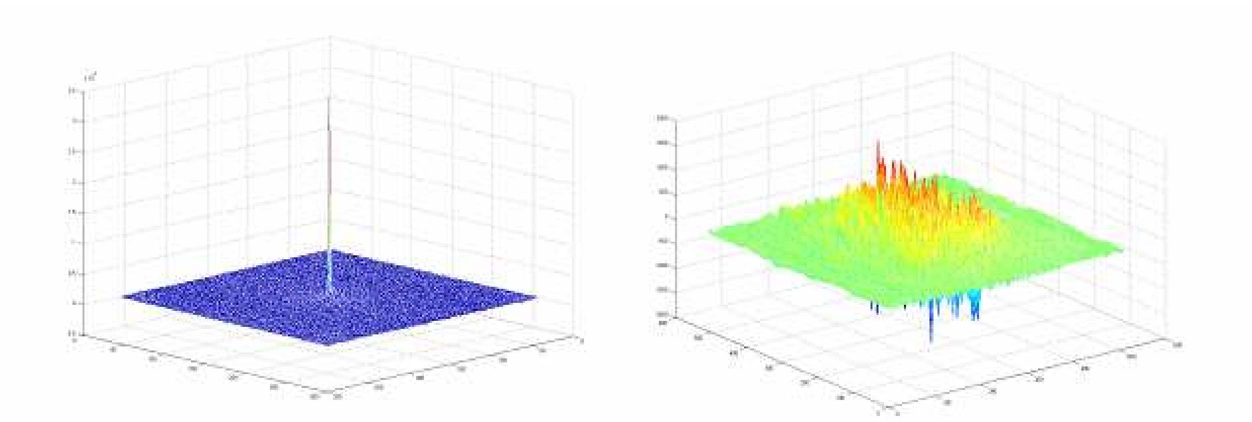


Figure 4: (Left) correlation plane for authentic sample (Right) Correlation plane for imposter sample

To quantify the difference between the two types of correlation planes, we define a measure of recognition called Peak to Sidelobe Ratio (PSR) [4]. This will measure the sharpness of the largest peak in the correlation plane with respect to the immediate area around the peak. PSR is an effective measure of correlation when we expect multiple peaks to be present in one correlation plane or in other words multiple authentic targets.

$$\boldsymbol{PSR} = \frac{\boldsymbol{peak - mean}}{\boldsymbol{std.dev.}} \tag{1}$$

There are various types of correlation filters a set of which we considered for use in our algorithm. We will now describe these filters.

The Minimum Average Correlation Energy (MACE) Filter [5] is designed to minimize the average energy *E* in the correlation plane or Average Correlation Energy (ACE). In the filter we also constrain the value of the correlation peak to be 1. To achieve this we analyze the spectral power density which is placed on the diagonal of the matrix *D*. Our goal is to minimize *E* which is defined as:

$$\boldsymbol{E = h^{+} Dh} \tag{2}$$

where $^{+}$ denotes the conjugate transpose. The constrained minimization of Eq. 2 results in the MACE filter h_{MAC2E}:

$$\boldsymbol{h_{MACE} = D^{-1} X \left(X^{+} D^{-1} X\right)^{-1} u} \tag{3}$$

where *u* is the constrained peak values (vector of ones).

The Unconstrained MACE (UMACE) Filter [6] removes the constraint on the peak value. By removing this constraint, more solutions to the minimization problem are available. We also try to maximize the average value of the peaks or Average Correlation Height (ACH). The closed form solution to the UMACE filter h_{UMACE}:

$$\boldsymbol{h_{MACE} = D^{-1} m} \tag{4}$$

where *m* is the average of the columns of *X*.

We will consider generalizations of the MACE and UMACE filters called the Optimal Tradeoff Synthetic Discriminant Function (OTSDF) filter [7] and the Unconstrained OTSDF (UOTSDF) filter respectively. These generalized filters offer sharp correlation peaks and noise tolerance which are related to the ACE and Output Noise Variance (ONV) of the filter respectively. However, these two qualities are inversely proportional to each other by a constant *a*. Given a desired proportion of peak sharpness to noise tolerance, the filter designs h_{OTSDF} and h_{UOTSDF} are:

$$\boldsymbol{h_{OTSDF} = T^{-1} X \left(X^{+} T^{-1} X\right)^{-1} u} \tag{5}$$

$$\boldsymbol{h_{UOTSDF} = T^{-1} m} \tag{6}$$

where *T* is defined as:

$$T = \alpha D + \sqrt{1-\alpha^2}C \qquad \text{given } 0 \le \alpha \le 1 \tag{7}$$

where *C* is the Gaussian white noise matrix (identity matrix). The primary difference between MACE and OTSDF is the replacement of *D* with *T*.

We analyzed each of these four correlation filter designs to determine which one was best suited to our approach. First we built filters using each design on the training data described above. The first test was to determine which filter was best at identification. Using the testing data that comprised of one-degree increment angular views we tested the identification abilities of each filter. This meant correlating each filter with both authentic and imposter data. The filter that yielded the highest PSR value extracted from the maximum peak within the correlation planes was marked as the determined class. The confusion matrices for each filter build are shown in Table 1, Table 2, Table 3, and Table 4. The alpha parameter for both the OTSDF and UOTSDF filters are chose essentially arbitrarily, but are based on past experience with these filters in non-ATR related situations.

Table 1: Mace filter confusion matrix

		Determined Class				
	MACE	Class 1	Class 2	Class 3	Class 4	Class 5
Testing Class	Class 1	285	56	15	0	4
	Class 2	62	296	2	0	0
	Class 3	21	26	302	6	5
	Class 4	0	0	15	312	33
	Class 5	2	0	20	34	304

Table 2: UMCAE filter confusion matrix

		Determined Class				
	UMACE	Class 1	Class 2	Class 3	Class 4	Class 5
Testing Class	Class 1	263	75	5	4	2
	Class 2	71	273	10	6	0
	Class 3	14	12	314	8	12
	Class 4	4	13	21	287	35
	Class 5	6	3	5	52	292

Table 3: OTSDF filter confusion matrix, alpha = 0.99

		Determined Class				
	OTSDF	Class 1	Class 2	Class 3	Class 4	Class 5
Testing Class	Class 1	312	46	2	0	0
	Class 2	34	323	2	0	1
	Class 3	2	6	322	12	18
	Class 4	0	0	3	336	21
	Class 5	3	4	10	12	331

Table 4: UOTSDF filter confusion matrix, alpha = 0.99

		Determined Class				
	UOTSDF	Class 1	Class 2	Class 3	Class 4	Class 5
Testing Class	Class 1	300	40	11	1	8
	Class 2	51	298	2	0	9
	Class 3	16	11	304	8	11
	Class 4	3	1	15	308	33
	Class 5	1	1	1	36	321

These results demonstrate that the OTSDF filter is the best filter for identification. However, we are also interested in how well these filters separate the classes. During the previously described test of identification capabilities we also made sure to store all calculated PSR values.

Table 5: Mace filter average PSR values

		Determined Class				
	MACE	Class 1	Class 2	Class 3	Class 4	Class 5
Testing Class	Class 1	10.0406	8.7621	1.4325	1.2167	1.1452
	Class 2	9.2912	11.2451	1.8742	1.3542	1.6562
	Class 3	1.2201	1.8641	10.5632	0.9238	1.0023
	Class 4	1.8342	1.9451	1.0342	10.9823	0.7542
	Class 5	1.4231	1.1374	1.1045	1.8652	10.1284

Table 6: UMCAE filter average PSR values

		Determined Class				
	UMACE	Class 1	Class 2	Class 3	Class 4	Class 5
Testing Class	Class 1	9.8417	8.8650	1.7221	1.7214	1.2179
	Class 2	8.5963	10.1727	1.3622	1.2511	1.2410
	Class 3	1.1211	1.7724	9.3671	1.1131	1.2125
	Class 4	1.6211	1.6312	1.6453	9.1244	1.5532
	Class 5	1.8532	1.2321	1.4111	1.2253	8.1317

Table 7: OTSDF filter PSR values, alpha = 0.99

		Determined Class				
	OTSDF	Class 1	Class 2	Class 3	Class 4	Class 5
Testing Class	Class 1	11.9422	9.1632	1.6225	1.1244	1.3342
	Class 2	9.2934	12.6724	1.1922	1.3872	1.4263
	Class 3	1.1154	1.4426	10.7121	1.0227	1.0023
	Class 4	1.7611	1.5112	1.1231	10.7114	0.9113
	Class 5	1.2117	1.0311	1.0765	1.1289	10.7128

Table 8: UOTSDF filter PSR values, alpha = 0.99

		Determined Class				
	UOTSDF	Class 1	Class 2	Class 3	Class 4	Class 5
Testing Class	Class 1	10.5221	8.7122	1.5127	1.3281	1.2117
	Class 2	9.3325	11.1178	1.7653	1.4123	1.4667
	Class 3	1.5612	1.8641	10.8141	1.0041	1.0157
	Class 4	1.9115	1.8672	1.0342	10.1241	0.9213
	Class 5	1.5112	1.5216	1.2217	0.9913	9.9813

These table demonstrate that the OTSDF filter provides the best separation, but also shows that the two tanks (Class 1 and Class 2/M1A2 and German Leopard) are strongly correlated compared to the other classes. However, taking into account our overall algorithm, we felt confident that we would be able to further increase the effective separation of these classes.

These tests of both identification capabilities and effective separation indicated to us that the OTSDF filter was the best suited filter. However, we could have performed further tests to determine what alpha parameter in the OTSDF filter build would have yielded the best performance. However, considering the range of possible alpha values such testing would have occupied too much time with respect to the overall project schedule. As such, we settled on the OTSDF filter built with alpha equal to 0.99.

Section 4.2 Overlap-Save Convolution/Correlation

Due to memory issues on the EVM, which will be detailed in later sections, we were forced to use a technique called overlap-save which we will not detail but provide a reference to, [8]. Using overlap-save we can perform the correlation of a 320 × 240 image by breaking it up into smaller blocks. This will also allow us to perform linear correlation as opposed to circular correlation. It is important to note that we will only perform linear correlation that results in the correlations being generated as a result of the center of the filter being above a valid portion of the image.

Section 4.3 Central Slice Theorem

The Central Slice Theorem (CST) [9] in two dimensions states that the Fourier transform of the projection of a two-dimensional function *f(r)* onto a line is equal to a slice through the origin of the two-dimensional Fourier transform of that function which is parallel to the projection line. The usefulness of the CST is in the fact that if we cane project an image, or in our case a frame of a video sequence, onto a pair of dimensions and do the same for our correlation filters and still retain their identities w.r.t. to the axes upon which they were projected. If we apply our projected one-dimensional correlation filters to the respective one-dimensional projections of the frame, we should still see distinct peaks in each resulting one-dimensional correlations. Suppose we only apply projections along two axes, x and y for simplicity, If we have only one target in our frame, then we should only see one peak in each correlation. By combining the positions of those peaks we will be able to isolate where the target is in the frame. The advantage of this approach is that we are now performing a pair of one-dimensional correlations rather than a fill two-dimensional correlation. This is a huge advantage is speed but comes at the cost of decreased accuracy. If we want to detect more than one target using only two projections we will not be able to exactly isolate the target. Instead, we will get a set of peaks in each one-dimensional correlation where the relation between the peaks in each correlation is not clear. Rather, if we get a set of *N* peaks in each one-dimensional correlation we will only be able to say that we have $N \times N$ possible positions for *N* potential targets. If we increase the number of projections we can reduce the number of possible positions for the *N* targets. However, using two-dimensional discrete data as is the case with images, it is not easy to compute projections along varying axes other than multiples of forty-five degrees. Due to this difficulty, we will consign ourselves to using only two or three projections and being satisfied with the relative isolation of detections. We tested the effectiveness of the filters by taking frames from our testing sequences and calculated probability of detection and miss rate for those frames. We tested a total of 50 frames for each target and each range with Range1 being the closest and Range3 being the farthest and using a threshold of 0.9 × the maximum peak value. These results, show in Table 9, are adequate for our purposes since the overall miss rate is at low enough percentages.

Table 9: Central Slice Theorem analysis results

Class (Range1/Range2/Range3)	Class 1	Class 2	Class 3	Class 4	Class 5
# of Correct Detections	41/36/38	40/37/37	36/33/34	46/44/40	47/45/44
# of False Detections	5/6/9	5/5/5	8/12/9	2/4/7	0/2/5
Probability of Detection	.82/.72/.76	.80/.74/.74	.72/.66/.68	.92/.88/.80	.94/.90/.92
Miss Rate	.08/.16/.06	.10/.10/.10	.12/.10/.14	.04/.04/.14	.06/.04/.10

Section 4.4 Detection

Detection is a difficult task in that it requires us to discriminate a target from its background. In real work applications, the background is often filled with what is collectively called “clutter”. Clutter is unpredictable and cannot be easily modeled statistically. This does not mean that there are not techniques/algorithms designed to identify and isolate clutter. However, such algorithms are well beyond the capabilities of the EVM and the scope of this class. As such, we cannot expect to identify things as being clutter and mark them as such. A more intuitive solution is to mark the object that could be targets.

For this purpose we use the CST as described above to provide us a set of possible detections. Around each of these detections we will place a window of varying size. The size of the window is dictated by the position of the detection relative to the depth of the image. The further back into the scene the detection is, the smaller the window which logically follows from the fact that targets farther away will appear smaller. The closer the detection is the larger the window used will be. The size of the window used is reflective of the maximum target size at those ranges. As stated above, our hope is that the limited number of available targets will created a limited number of possible detections and as a direct result a limited number of windows. We are sure to enforce a minimum distance between potential detections to insure that we do not have overlapping detections. This minimum distance is the maximum size of a target. This may detriment performance by causing us to throw out correct detections, but it is preferable to trying to perform identification and tracking on obscured and overlapping targets. The number and sizes of these windows will directly impact the speed and performance of the identification stage. The end results of the detection stage are a set of windows of varying sizes each centered on the location of a potential detection which are then input into the identification stage.

Section 4.5 Identification

In the windows provided by the detection stage there may possibly be a target of interest. At this point we are interested in not only confirming the presence of a target, but also the identity of the target. Assuming that the user of the system has specified a set of targets that they are interested in tracking, we proceed to apply the correlation filters created to identify those classes. Let us now consider one window containing one potential target. The user has chosen *M* classes of targets and we proceed to apply the corresponding *M* correlation filters. From those *M* correlations, we get *M* correlation planes in which we isolate the peak and calculate the PSR for. At this point we have *M* PSR values for a single window, each one corresponding to a particular class's correlation filter. The class's correlation filter which yields the highest PSR value is designated the class from which the target belongs to. We also consider the possibility that no target or an object that is not actually a target is contained in the window. If this is the case we expect to get a relatively low PSR value. What is considered to be a low PSR value is the result of analysis that we will detail later. We threshold our maximum PSR value to this threshold to limit the number of false identifications. Once we have either confirmed or denied the presence of a target and its identity, we make sure to not the position at which the peak was detected. The position of the peak tells us the suspected location of the target. At the end of this stage we have hopefully detected and identified all targets within a single frame. However, there is a possibility that we have either missed targets or detected objects that are not targets (miss rate and false alarm rate respectively). We now proceed to the tracking stage.

Section 4.6 Tracking

It is important to note that the previous two stages of detection and identification are performed consecutively and sparsely. In other words, we do not perform these stages on each frame. This is due to limitations of the EVM in terms of processing speed and storage capacity. However, if we do not perform these stages, how do we perform tracking? The solution to this is the algorithm proposed in [2]. We relate the processing of consecutive frames in a Markovian sense which will lead us to call the tracking algorithm a Hidden Markov Model [10].

Taking the results from the identification stage we now apply a probabilistic model. We create a probability plane the same size as a single frame, in our case 320 × 240. The probability at each position represents the probability that a target is at that position. Immediately following the identification stage we have a series of target locations that as far as the algorithm is concerned are correct. Therefore, at each of these "confirmed and identified" target positions we have a probability of 1. Expressed formally the probability of p_i target t_i being at pixel x in frame F is

$$p_i\left(x \middle| F\right) \tag{8}$$

Since we assume fill confidence in our initial set of detections and identifications, p_i is equal to 1 for all targets. Thus, our probability plane sums to the total number of detections. This may seem counterintuitive to the notion of a probability distribution function (pdf), but the probability plane is not a pdf. It is a collection of pdf's centered on target positions. Therefore, the total probability or sum of values within the probability should be the sum of all the pdf's present.

Now the question is how do we track these targets across frames? If we assume that p_i is independent across frames the solution is to relate the frames probabilistically using two-dimensional Gaussian distributions that represent the probability of a target moving from one position to another. This distribution should be representative of the speed and direction at which the target is expected to move across the span of a frame during a video sequence. If we expect the target to move slowly and downwards across the frame, we will model that as a non-circular Gaussian with low standard deviation. This will result in the higher probabilities being concentrated around the origin and downwards. However, since in our case we do not expect any predictable movement from our targets we will model our distributions as circular ones. Therefore, we can express the probability of a target moving from pixel x_1 to x_2 in terms of displacement and not position. Thus given a Gaussian distribution $G(d)$ where d is the displacement or distance from a the mean or center of the distribution, we can say that the probability of p_t of a target t_i moving from pixel x_1 in frame F_1 to pixel x_2 in frame F_2 is

$$p_t\left(x_1, x_2 \middle| F_1, F_2\right) = G\left(\left|x_1 - x_2\right|\right) \tag{9}$$

Also, we will assume the target does not move exceedingly far between frames. To this end we can specify the sigma parameter our Gaussians to reflect the expected movement of our circular Gaussians, the larger the expected movement, the larger the sigma parameter. We will call these Gaussians transition probability matrix (TPM). Figure 5 demonstrates the difference between sigma parameters.

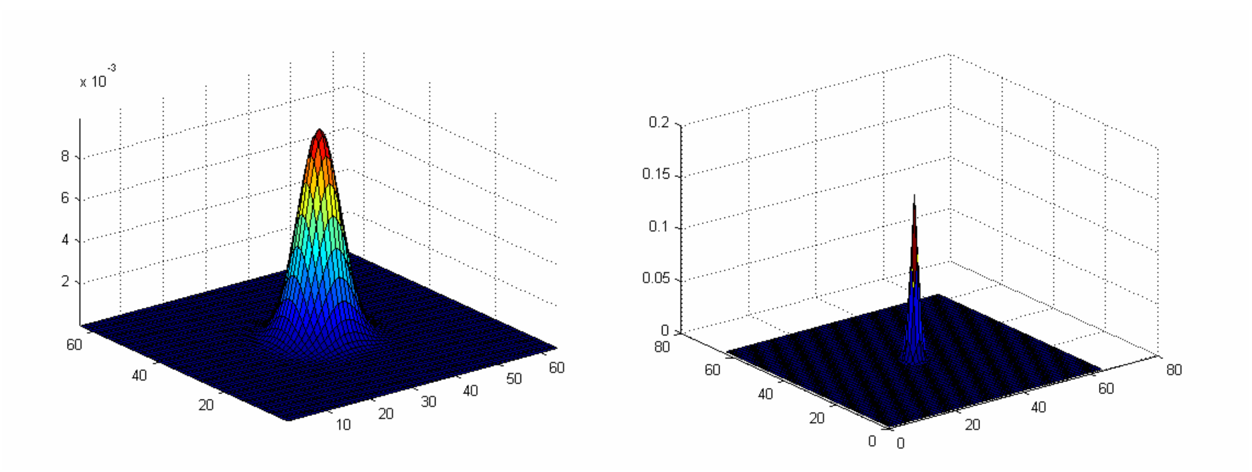


Figure 5: Transition probability matrix (Left) Sigma = 4 pixels (Right) Sigma = 1 pixel

Now that we have established what our TPM's represent, how do they factor into our tracking scheme? Following the detection and identification stages, which span only one frame, we have a set of targets whose initial location and identities have been established. We then process the next frame by extracting windows centered on each target location. As was with the identification stage, the size of the window depends on the range of the target. However, these windows are larger than those in the identification stage to accommodate for target movement. In these windows we apply the correlation filter of the class to which the target belongs to. From the resulting correlation plane we extract PSR of the maximum peak. Now given that we have detected our peak in frame F_f at position/pixel x_i, we compute the probability that it reached x_i from previous positions. Let us suppose that f is 2, in other words we are processing the second frame assuming the first frame has been used in the detection and identification stages. At this point we are interested in determining the probability for each target that they have reached their respective pixels as determined by their correlation planes. Using our Markovian assumption and the independence of the frames, the probability for target t_i being at pixel x_i in frame F_f given we have calculated its PSR value is

$$p_i\left(x_i \middle| F_f\right) = psr_i * G\left(\left|\left(x_i \middle| F_{f-1}\right) - \left(x_i \middle| F_f\right)\right|\right) * p_i\left(x_i \middle| F_{f-1}\right) \text{ given } \mathbf{f} = 2 \qquad (10)$$

Let us now consider even more future frames or for values of *i* greater than 2. Therefore, the probability of target *i* being at position x_i in frame F_f is

$$p_i\left(x_i \middle| F_f\right) = psr_i * G\left(\left|\left(x_i \middle| F_{f-1}\right) - \left(x_i \middle| F_f\right)\right|\right) * \prod_{g=1}^{g=f-1} p_i\left(x_i \middle| F_g\right) \text{ for } \mathbf{f} > 1 \qquad (11)$$

Without any further operations, this equation implies that given that p_i is a true pdf (maximum value is 1) then as we process frames p_i will continue to decrease in value regardless of how strong a PSR value we achieve for each target in each frame. Also, the nature of the tracking problem necessitates the use of a threshold to insure that only true and actual targets are detected and followed. Unless we normalize the probabilities in each frame in a constant manner then any threshold we establish is arbitrary and unjustified. While there are many kinds of normalization, we choose to normalize all p_i to the total number of targets detected, identified, and being tracked. After this normalization, each p_i is now normalized to between 0 and 1. By using this normalization we can use a threshold for p_i to track and identify our targets. Matlab simulations on our captured video sequences containing only one target with varying thresholds have yielded the false alarm/miss rates in Table 10. We make sure to note the false alarm rates because in our scheme a false alarm means a false target being tracked.

Table 10: Matlab simulation false alarm/miss rates

Threshold	Class 1	Class 2	Class 3	Class 4	Class 5
0.60	0.514/0.006	0.522/0.012	0.564/0.080	0.482/0.002	0.498/0.004

0.65	0.428/0.042	0.468/0.062	0.492/0.122	0.416/0.024	0.422/0.036
0.70	0.324/0.094	0.346/0.084	0.386/0.172	0.286/0.054	0.298/0.064
0.71	0.302/0.102	0.322/0.114	0.346/0.264	0.264/0.092	0.282/0.098
0.72	0.286/0.122	0.264/0.182	0.338/0.298	0.242/0.100	0.258/0.106
0.73	0.274/0.156	0.246/0.146	0.312/0.328	0.220/0.108	0.238/0.114
0.74	0.268/0.188	0.238/0.174	0.306/0.364	0.212/0.132	0.228/0.154
0.75	0.254/0.204	0.224/0.218	0.294/0.408	0.204/0.186	0.214/0.198
0.76	0.245/0.216	0.220/0.242	0.286/0.426	0.196/0.202	0.202/0.208
0.77	0.222/0.274	0.218/0.286	0.262/0.484	0.188/0.254	0.194/0.260
0.78	0.216/0.328	0.214/0.334	0.254/0.592	0.182/0.316	0.188/0.322
0.79	0.210/0.412	0.208/0.404	0.248/0.610	0.174/0.364	0.176/0.376
0.80	0.182/0.472	0.186/0.448	0.224/0.632	0.148/0.422	0.156/0.438
0.85	0.108/0.568	0.112/0.554	0.182/0.688	0.092/0.496	0.102/0.512
0.90	0.052/0.640	0.086/0.626	0.128/0.740	0.012/0.522	0.038/0.546

Section 4.7 Overall Algorithm

In order to put all of these stages together with reference to the overall algorithm we will address the role and position of each stage. We have three distinct stages of which only the tracking stage is iterated through more than once. However, we also wish to detect new targets that may have entered the scene and as such we should apply the detection and identification stages again to detect those newly introduced targets. The frequency at which we apply the detection and identification stages can be though of as a “detection increment” in that we will only attempt to detect and identify targets when the frame being processed is the k^{th} frame where k is the size of the increment. In this way, we can limit the processing load of detecting targets and still retain some chance of detecting new ones. An added bonus to the repetition of these stages is that it allows us to confirm or deny the presence and identity of already detected and identified targets.

This should be done because there is a chance that the filters have incorrectly identified the target for reasons of range and orientation. A common difficulty in ATR applications is the idea of pixels-on-target which describes the number of pixels that make up the target in the frame. This number in essence describes how much data is available for analysis. The less data the harder it will be to perform correct analysis. The most common reason for reduced pixels-on-target count is range. The farther away an object is the smaller its profile and the less pixels-on-target it presents. If we had initially identified a target at far range we say that we have less confidence on that identification. However, if the opportunity to identify that target at a closer range is available we would be more confident in the resulting identification. As such, we should try to reconfirm the identities of targets periodically to insure confidence in our identifications. We can only do this because we have confidence in the tracking algorithm which will follow even incorrectly identified targets from far range to close range.

Now that we have addressed the role and position of the detection and identification stages, we can address the tracking stage. The most thorough approach to the problem of tracking would be to analyze all frames in a video sequence. However, we can say with some measure of confidence that the targets in our application will not be moving very far between frames assuming a certain frame rate. The standard frame rate for USB webcams is 30 fps. At such capture speeds, there is unlikely to be significant variation in target position from frame to frame. Thus, it would more efficient for us to analyze frames which are still ordered but now separated by a delay to insure some target movement. The delay or “tracking increment” *d* depends not only on the speed of the camera capture but also the speed of the target. As such, we would only apply the tracking stage to every d^{th} frame. Considering that we are in full control of the targets, we will limit them to be relatively slow moving in a scene that is being captured at 30 fps. Taking into account the digital controls of most of the RC models (models move in fixed increments) we have found through simulation that a tracking increment of five frames works for most cases.

The following is the overall algorithm with respect to the implementation of a GUI:

1. User chooses class(es) of target(s) that he wishes to detect, identify, and track
2. Appropriate filters are loaded
3. User chooses video sequence to analyze/track targets through
4. Detection and identification stages are applied to first frame
5. Tracking stage is applied to every d^{th} frame following
6. Detection and identification stages are applied to every k^{th} frame following

Section 5. Signal Flow

Our DSP system began with a very ambitious signal flow which incorporated a large portion of the ATR solution on the EVM. In this original signal flow, the PC was responsible primarily for capturing each time frame, pre-processing the current frame into windows using the Central Slice Theorem (CST), and displaying the results from the EVM on the Graphical User Interface (GUI), showing the position and identity of all tracked targets.

On the EVM side, the C67 processor unit was responsible for receiving the properly arranged windows, correlating them with specific filters using two-dimensional Fast Fourier Transforms (FFTs) and Inverse FFTs (IFFTs), and then finding the correct position and identity of each target in the window using Peak-to-Sidelobe Ratios (PSRs) and Hidden Markov Models (HMMs).

However, it was quickly realized that not all of this data could possibly fit on the EVM, due to memory constraints. Because of this, only the FFT and IFFT would occur on the EVM, with the peak detection and HMM modeling done on the PC side. Thus, the signal was captured by the camera and arranged on the PC into appropriate windows, which were then sent to the EVM. On the EVM, the windows were correlated with a specific filter (which had been loaded onto the EVM during the initial start-up), which was then sent back to the PC to form part of the frame’s correlation plane. By using priority queues to determine the peak and PSR calculations to determine the best correlation (and the identity), a set of positions and identities for the entire frame was generated.

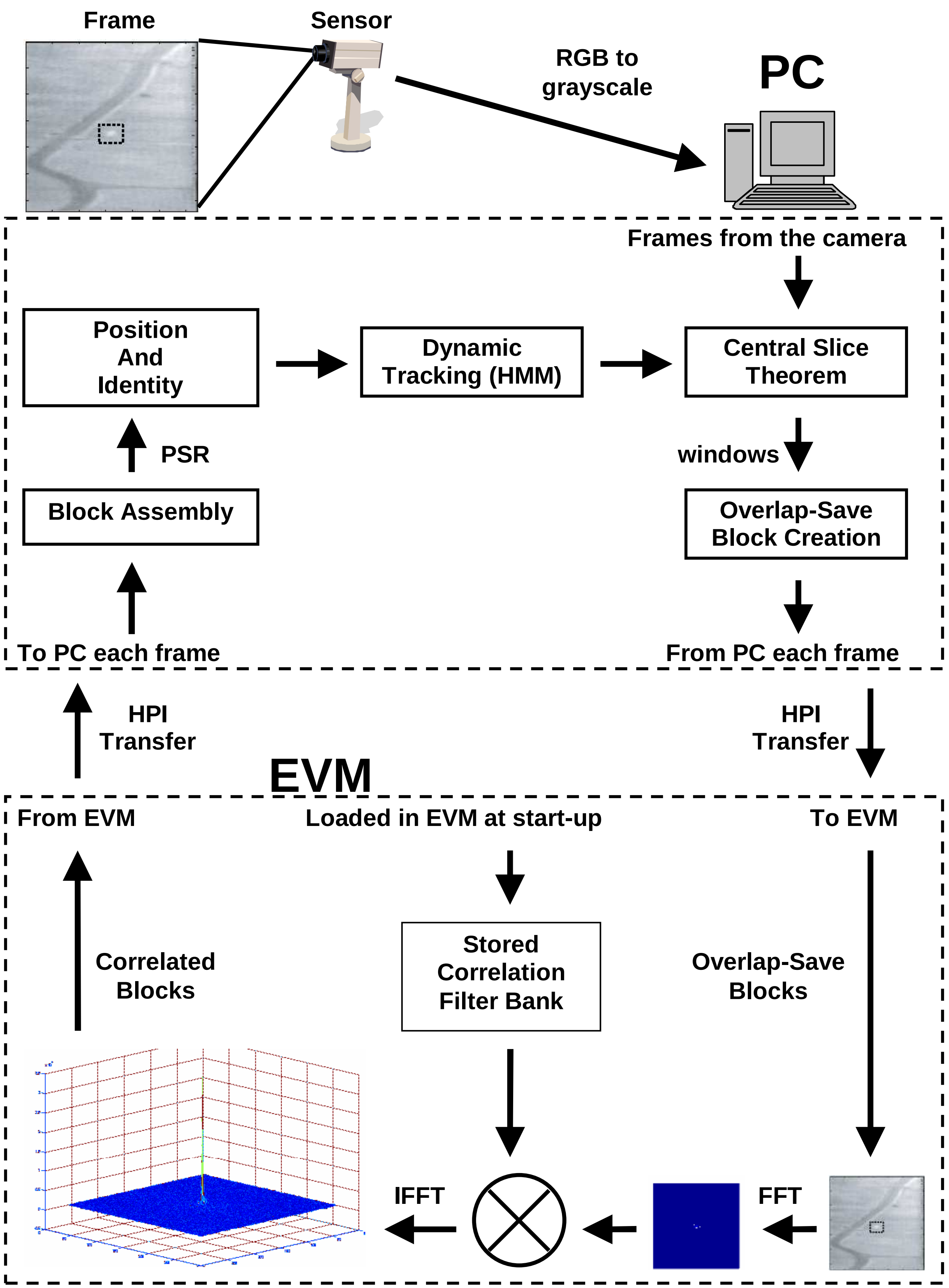


Figure 6: Complete signal flow between GUI, PC side, and EVM.

From this set, later targets were tracked by taking the windows as found by the CST and applying only the filters of the identity determined in the previous detection stage. These correlation planes were then multiplied by the HMM probabilistic distribution to determine the likely position of each target in the current tracking frame.

Section 6. What was done on EVM

Originally we expected the EVM to compute the correlation plane of a set of blocks and only return back to the PC the location and value of the highest peaks from the correlation peak. However, that proved to be unfeasible do to memory constraints. As described in the algorithm, finding a peak involves finding the PSR value as well as making sure that the peak is not adjacent to another peak (suggesting, multiple detections for the same object). We realized that if a detection was found close to the edge of a block, in order to accomplish this task correctly, it will require knowledge about the correlation values in adjacent blocks. Thus, we had to move PSR calculations to the PC where the correlation plane of the image is reassembled. For that reason, the function of the EVM has been reduced to computing the correlation plane of the specified window and returning the entire correlation plane for further processing on the PC.

Section 7. Speed Issues

The rate of data transfers and processing that are required for the project begin with the maximum speed of our targets and continues through to the processing time on the EVM. Originally, we had hoped to do the tracking in real-time however, this was unrealizable due the speed limitations of our devices. The following explains in detail the derivation of the different processing rates necessary to realize this project in real-time. Lastly, we will explain what we decided implement in our design.

Section 7.1 Target Movement Rate (pixels/second)

The maximum speed of the target in terms of pixels changes based on the distance from the camera. The small cars, take between 1 seconds and 3 seconds to go across 320 pixels on at the closer scale and the larger scale, respectively. However, our plan was to drive them slowly to avoid blurring images. The tanks on the other hand, are much slower than the cars. They generally take about 3 and 10 seconds to cross 320 pixels at a closer and further scale, respectively.

Section 7.2 Frame Capture Rate (frames/second)

As a result of the analysis we thought that 30fps would be a good frame capture rate for testing purposes. However, a frame rate of 10fps would be a feasible capture rate for this project. 10 fps would limit the movement of the targets to a maximum of 32 pixels between frames at the largest scale.

Section 7.3 Frame Processing Rate (frames/second)

In order to get real-time results, the frame must be processed at less than 10fps. Frame processing includes several complex steps as is explained above. During each frame, the amount of processing changes based on several factors. The number of targets determines how many windows within the frame we need to look for the targets within. The position of the target determines the size of the window and size of the filters needed for processing.

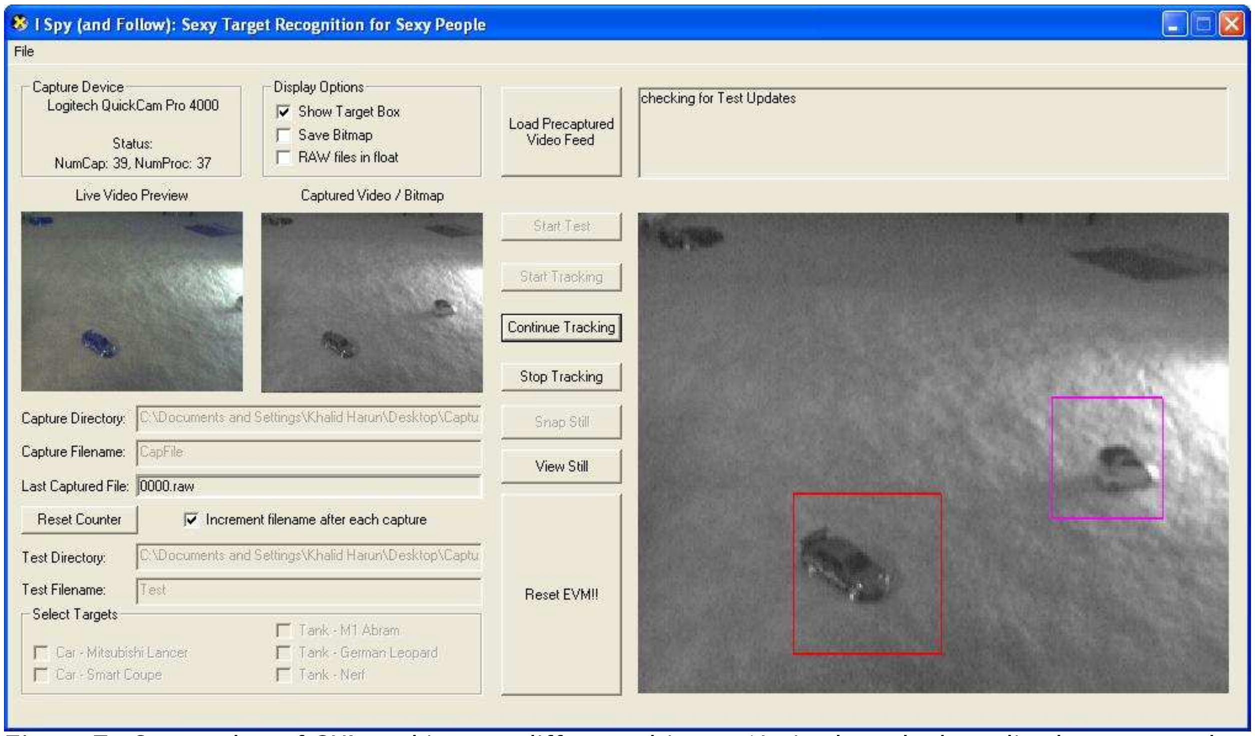
Some rough calculations of the processing time using the timer given in the timer.h gave the following results: HPI transfer PC→ EVM: 3.5 ms/block, HPI transfer EVM →PC: 7 ms/block, 2D Correlation on EVM: 25 ms/block

Section 8. Video Interface

The source of our video stream comes from two sources. Our primary source is a live webcam feed. Our secondary source is a video file in AVI format. In order for the video to be compatible with the targeting process, the video needs to be broken into individual frames. Each frame is a 320 x 240 grayscale image saved in RAW format. The RAW format is similar to format used in Lab 3 except that each pixel is stored as a 1-byte char rather than a 4-byte int. The webcam capture library we are using is the DirectX SDK.

Section 8.1 GUI

The graphical user interface (GUI) has two modes, a tracking mode and a testing mode. The tracking mode captures a live feed from a webcam attached to the computer as the source of the tracking procedures. The test mode uses a prerecorded set of images for the tracking procedures. The user has the option of selecting a directory and a filename for testing and tracking modes.


I Spy (and Follow): Sexy Target Recognition for Sexy People

Capture Device: Logitech QuickCam Pro 4000
Status: NumCap: 39, NumProc: 37

Display Options: [✓] Show Target Box, [ ] Save Bitmap, [ ] RAW files in float

checking for Test Updates

Capture Directory: C:\Documents and Settings\Khalid Harun\Desktop\Captu
Capture Filename: CapFile
Last Captured File: 0000.raw
[✓] Increment filename after each capture
Test Directory: C:\Documents and Settings\Khalid Harun\Desktop\Captu
Test Filename: Test

Select Targets: [ ] Car - Mitsubishi Lancer, [ ] Car - Smart Coupe, [ ] Tank - M1 Abram, [ ] Tank - German Leopard, [ ] Tank - Nerf

Figure 7: Screen shot of GUI tracking two different objects. Notice how the bounding boxes around the two targets are of different. The size of the bounding box denotes the scale and color represent the identity.

Section 8.2 Grayscale

Using the Microsoft DirectX SDK, it captures images as 24-bit RBG Bitmaps. In order to convert it to grayscale, we used the following conversion formula:

$$\text{GRAYSCALE VALUE} = (0.3 \times \text{RED}) + (0.59 \times \text{GREEN}) + (0.11 \times \text{BLUE}) \qquad (12)[13]$$

Section 8.3 Cast to float for processing

Since the saved capture frames are saved as 1-byte pixels and the rest of the processing is done in float, the images are cast as float when they are read into the program that controls the EVM.

Section 9. Synchronization

Synchronizing the GUI with the EVM in one application was a difficult problem due to our lack of experience with Windows Messaging. Therefore, we decided to break the problem into two stages, GUI to PC side, and PC side to EVM. The two sections communicate to each other through files created in a shared directory.

Section 9.1 Naming Standards

As mentioned prior, synchronization is done through files created within a specified directory. Therefore, it is important to be able to distinguish between different file types. There are three different file types we deal with, **.exp**, **.raw**, and **.track** files. Below is a discussion of the format or each file. Before that, a quick word needs to be mentioned about the general naming scheme. Every file related to a particular experiment can be distinguished by the same initial part of the file name. Additionally, information specific to particular frames are related by the same the four-digit number following the filename. This number is zero padded to four digits for consistency.

[experiment name].exp – experiment setup file

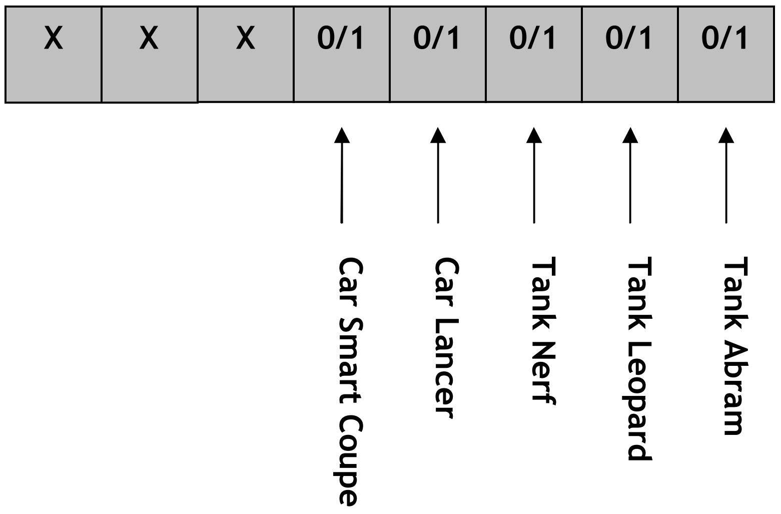


Figure 8: An experiment setup file encodes the targets to look for in the current experiment. Each bit is mapped to a target as the specified in the figure above.

The setup file is a 1-byte file that tells the PC side which targets to track. It is bitmapped for each separate target. If a bit is 1, it means that we are tracking that target and if the bit is 0 then we are ignoring it. Since we track up to five objects, the 5 LSBs are used the represent the target selection and the 3 MSBs are unused. The figure above shows the mapping for our particular case.

[experiment name][####].track – tracking information

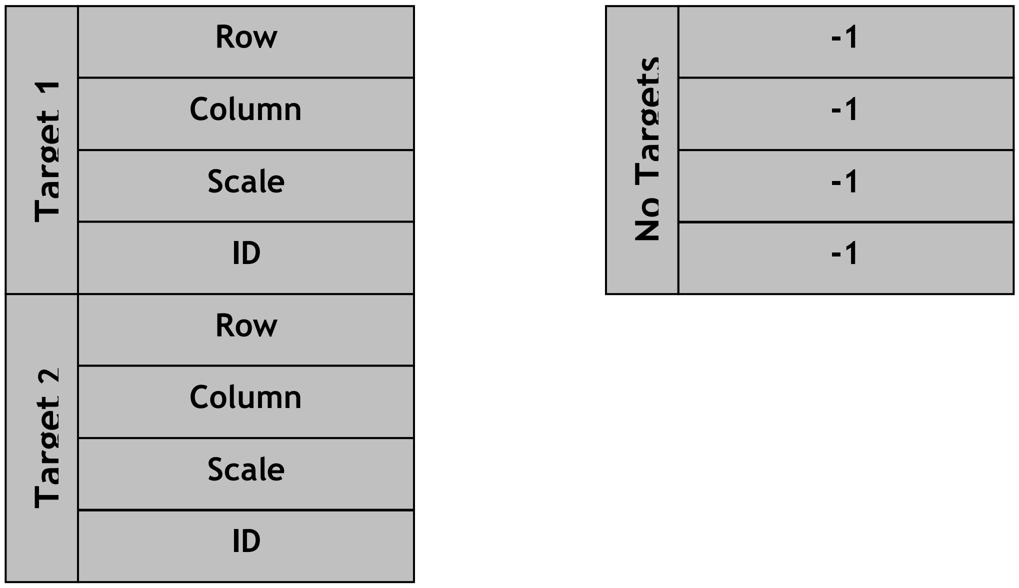


Figure 9: The diagram on the left shows the format of the tracking file with two targets. The diagram on the right shows a tracking file if there are no targets in the window.

The size of the tracking file changes with the number of targets identified by the PC side application. The information for a target is encoded in 4 integer values that are written to the file. For each target, another 4 integer values are appended to the file. The 4 integer values encode for the position of the target in a row index and column index, the scale of the object, and the identity of the object.

[experiment name][####].raw – grayscale image of a frame

This file is setup as explained in Section 8 (Video Interface). There is no header or footer for the file; it is just raw data. We are able to get away with this since all of our images are of the same dimensions and types.

Section 9.2 GUI and PC side

The GUI provides the PC side with an experiment setup file and all of the frames to process. As mentioned above, the images are converted into 320 x 240 chars and saved to a .RAW file. These files are saved to the same directory. The first file is saved as [experiment name]0000.raw. In each frame captured the number increments by one. This number is what we use to properly sync captured frames with the resulting targeting information.

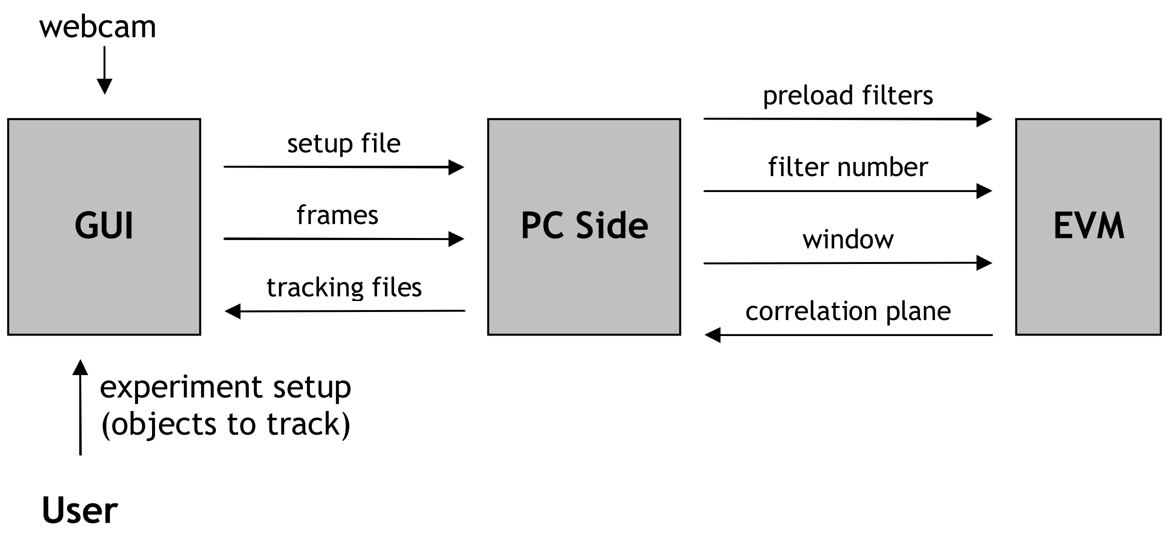


Figure 10: Overall communication scheme between the GUI, PC side, and EVM modules.

Once these images are created, they are ready for processing by the PC side application. The PC side application keeps reading in frames with incrementing filenames starting with [experiment name]0000.raw. When a file is not found, the experiment ends. This works because, the frame capture rate is greater than the frame processing rate. After each frame is processed, a track file is created in the same directory.
While the PC side is processing a frame, the GUI polls every 200 milliseconds for a newly processed tracking file from the PC side. When a tracking file is found, it the GUI loads the corresponding frame from the saved .raw file and uses the information from the tracking file to overlay a box around the target. After this process is complete, the GUI waits for the next track file that is create by the PC side process until the tracking index on the tracking file matches the last captured frame.

Section 9.3 PC side and EVM

On the other end of the PC side, it communicates with the EVM to compute the correlation plane. On initialization, the PC side loads the filters needed for the current experiment based on which target is being tracked. Additionally, the EVM allocates space for the blocks that are to be sent into the EVM. After this initial load, the communication protocol between the EVM and the PC side is detailed below. As you can see below, among the things we had to add several wait stages in order to synchronize the two sides and reduce the risk for error.

Table 11: PC Side ←→ EVM Communication Protocol.

Initialization – Runs only once on initialization

1. The PC side waits until the EVM is initialized and makes a request through a PCI mailbox message.
2. A mailbox PCI transfer is sent to the EVM with an integer value that represents the maximum number of blocks that will be transferred to the EVM. This is used to allocate enough space in the external memory to store future data transfers.
3. The PC side waits until the EVM makes a request through a PCI mailbox message.
4. A mailbox PCI transfer is sent to the EVM with an integer value that represents the total number of filters to store in external memory. This is used to allocate enough space in the external memory to store all of the filters needed for the particular experiment in external memory.
5. The PC side waits until the EVM makes a request through a PCI mailbox message.
6. An HPI transfer is initiated to transfer the entire bank of filters to the EVM.

Main Routine – Repeats until the end of the experiment

7. The PC side waits until the EVM is done processing and sends a message through a PCI mailbox message that it is ready to receive a new set of blocks to process.
8. A mailbox PCI transfer is sent to the EVM with an integer value that represents which of the preloaded filters to use on the current blocks.
9. PC waits until the mailbox has been read.
10. A second mailbox PCI transfer is sent to the EVM denoting the number of blocks to use this filer on.
11. PC waits until the mailbox has been read.
12. The window that we want a correlation plane for is sent to the EVM in blocks of 64 by 64 pixels through an HPI transfer.
13. After the EVM processes each block, the results are returned to the PC sided through another HPI transfer.
14. PC waits until the HPI transfer completes.

Section 10. PC side

Taking into account the complexity of our algorithm, we soon realized that it was unfeasible to implement a significant number of the operations on the EVM. As such, the PC implemented most of our algorithm. Here we will briefly describe what the PC was responsible with respect to pre-processing and algorithm implementation.

Section 10.1 Algorithm functions

The PC was responsible for all of the algorithm functions except for the correlation of the individual blocks used in the overlap-save correlation as described above. As a result, the first part of algorithm addressed by the PC was the detection stage. In this step, the PC not only projected the frame being processed onto the axes being used but also performed a time-domain correlation of the frame projections with the detection filter projections. We chose to use a time-domain correlation for two reasons. The first was to insure that we performed a linear correlation as opposed to a circular correlation. The second was to prevent a circumvention of the EVM responsibilities. To implement a FFT on the PC would undermine the use of the EVM at all.

After detection of potential targets is complete, the next step would be to use the correlation filters to determine the identities of PC uses the locations of these detections to position a window of the appropriate size around each size. In each window all available filters should be applied to determine the identity of the target. The correlation being performed by the EVM, the PC must prepare the blocks needed for overlap-save correlation. The specifics of the block preparation are discussed in following sections. Once the blocks are prepared they are transferred to the EVM. Once the EVM is finished performing the correlation the resulting correlated blocks are transferred back to the PC. The PC then takes these blocks and extracts the valid portions and assembles them into the correlation plane. Using priority queues, the PC extracts the five largest peaks in the correlation plane that not only meet a certain PSR threshold but are also separated from each other and the edges of the image by a certain distance to insure that only peaks are detected and not the sidelobes of peaks. The peak yielding the highest PSR is determined to be the location of the target. This process of maximum PSR extraction is repeated for each filter application and the class of the filter yielding the highest PSR value is labeled as the target's class. This process is repeated for all detections following which we enter the tracking stage.

Using the detection and identification results, in subsequent frames we applied appropriate correlation filters to windows centered on the detections and current tracking positions. Using the probabilistic model described above, we tracked the targets through the sequence while applying the detection and identification stages at appropriate frames to confirm the identities and presence of existing targets and to detect new targets.

Section 10.2 EVM communication

EVM communication was primarily concerned with correct transfer of overlap-save blocks to and from the EVM. This was achieved via HPI and PCI transfers. HPI transfers were used for the blocks themselves while PCI was used to inform the EVM of parameter information concerning the number of blocks, filters to be used, and timing synchronization between the EVM and the PC.

However, before this could occur the filters that would be used first had to be transmitted. This meant notifying the EVM via PCI how many filters were to be used overall or in other words how many classes were going to be tested for. After receiving the filters via HPI, the EVM would be ready to perform overlap-save correlation. This first involved notification of the EVM via PCI about the number of blocks to receive and how filters to apply to a set of blocks. We are concerned with applying more than one filter to one set of blocks during the identification stage where we apply all filters to one window or set of blocks. Then the EVM was informed of which filters to apply for a certain set of blocks. Following the correlation of the blocks with a filter, the EVM notified the PC to be ready for reception of the correlated blocks. The correlated blocks would be sent back to the PC via HPI.

Section 10.3 Filter preparation

Preparation of the filters was done using the OTSDF filter with an alpha parameter of 0.99. Due to the single precision native to the EVM, we built our filters using single precision. The Matlab code in later sections has the specific details of our filter build.

Section 10.4 Block preparation

Block preparation was done in the way detailed in [8] but in a two-dimensional sense. We made sure to zero-pad the original image in such a way that when blocks where extracted, their valid portions would contain correlation values resulting from correlation of the filter with the image when the filter's center is above a valid portion of the image. This portion of the PC code occupied the vast majority of the processing time since it essentially required us to clone the image and separate it into the appropriately overlapping blocks necessary for overlap-save correlation.

Section 10.5 EVM Emulation

In order to expedite our debugging of our algorithm, we developed an EVM emulator in C to be used on the PC. This emulator included emulation of both HPI and PCI transfers to insure complete similarity of the emulator to the actual EVM. As such, no major changes to the PC code were necessitated except for the change in name of some basic function calls involving the EVM. One primary difference between the emulator and the actual EVM was the specific FFT being used. While the EVM utilized a radix-4 single precision FFT, the emulator utilized a mixed-radix FFT whose source is listed in the references.

Section 11. EVM

In the original draft of our project, the TMS320C6701 Evaluation Module (C67 EVM, or EVM, for short) was responsible for a whole host of processes relating to the integrated ATR system. In an effort to maximize utility and efficiency of our system, the EVM was going to compute a frequency domain correlation with a given identification filter. This entailed performing a two-dimensional FFT of the current frame, multiplying this frame by a specific identification filter to create a correlation in the frequency domain, and finally compute the two-dimensional IFFT of this correlated frame to compute the correlation plane in the time domain. The original plan then also called for peak detection and PSR calculations, which were computed by sorting the frame and finding the maximum correlation plane values.

However, it soon became apparent that all of this data could not fit on the EVM. Since we planned on using Radix 4 FFTs over Radix 2 FFTs (due to the smaller number of cycles, meaning Radix 4 is faster), we needed to choose a row size and column size that were factors of 4. It became obvious that a 256 x 256 (4^4 x 4^4) block would be much too large for the EVM, and a 16 x 16 block (4^2 x 4^2) would be too small to correlate with our filters, which were 50 x 50 at the largest scale (meaning we would have to page in both the input and the filter, which was undesirable). We therefore decided on 64 x 64 (4^3 x 4^3) blocks to FFT and correlate on the EVM.

Since we wanted to perform all of the additional operations on the EVM, we originally tried a fixed point iteration of the FFT on the EVM, requiring half as much memory as the floating point FFT. This soon proved untenable though, as we discovered that the fixed point representation was incompatible with the PC side, and the FFT was not reliably invertible.

This lead to our final decision to use the floating point Radix 4 FFT on 64 x 64 blocks, and to move all of the non-correlation functions to the PC side.

Section 11.1 Data Memory and Code Size Needed

The specific values for each of these sections of memory were extracted from the .map file that was created when the project was built.

11.1.1 Internal Memory, on-chip

ONCHIP_DATA:	0x0de06 used out of 0x10000
ONCHIP_PROG:	0x0fc20 used out of 0x10000

Table 12: Internal Memory, on chip

Data Element	Size (bytes)	Total	Location
currentBlock	64 x 64 x 2 x4	32768	ONCHIP_DATA
Filter	64 x 32 x 2 x 4	16384	ONCHIP_DATA
rowTemp	64 x 2 x 4	512	ONCHIP_DATA
currentNumFilters	4	4	ONCHIP_DATA
currentNumBlocks	4	4	ONCHIP_DATA
currentFilter	4	4	ONCHIP_DATA
Twiddle	64 x 2 x 3 / 4 x 4	384	ONCHIP_DATA
Revtable	64 x 4	256	ONCHIP_DATA
rev_j	4	4	ONCHIP_DATA
I	2	2	ONCHIP_DATA
J	2	2	ONCHIP_DATA

maxNumBlocks	4	4	ONCHIP_DATA, stack
totalNumFilters	4	4	ONCHIP_DATA, stack
Done	4	4	ONCHIP_DATA, stack
Reinitialize	4	4	ONCHIP_DATA, stack
Request	1	1	ONCHIP_DATA, stack
Function variables	<24	<24	ONCHIP_DATA, stack

11.1.2 On-board Memory, off-chip

SBSRAM_DATA:	0x00000 used out of 0x2c000
SBSRAM_PROG:	0x001a0 used out of 0x14000

Table 13: On-board Memory, off-chip

Program Element	Location
Mkrevtable	SBSRAM_PROG, “FAST”
Fillwtable	SBSRAM_PROG, “FAST”

11.1.3 External Memory, off-board

SDRAM0:	0x400000 used out of 0x400000
SDRAM1:	0x320000 used out of 0x400000

Table 14: External Memory, off-board

Data Element	Size (bytes)	Total	Location
filterBank	<15 x 64 x 64 x 2 x 4	<491520	SDRAM0
evmInput	<128 x 64 x 64 x 2 x 4	<4194304	SDRAM0
evmOutput	100 x 64 x 64 x 2 x 4	3276800	SDRAM1

Section 11.2 Memory Allocation

Originally, in the fixed point implementation of the FFT, we had implemented paging in each frame to compute the correlation over the entire frame, since there was no way that we could store a 320 x 240 image in the on-chip memory of the EVM. This was accomplished using DMA transfers from external memory, where the blocks had been HPI transferred from the PC side, which had properly arranged them according to their 64 x 64 block within the larger window. The hope was to ping-pong DMA transfer two blocks on the EVM at a time, similar to lab 3, greatly increasing our efficiency.

Unfortunately, we were unable to fit this amount of data on the EVM using floating point without significantly reducing the processing ability of our system. With a 64 x 64 block, we would need two of these blocks at once for a ping-pong DMA transfer, which would require 64 kB of on-chip memory just for the blocks. Since other data needed to be stored in the on-chip memory (such as the twiddle and digit reversal tables), this was not a possibility. The other option to use ping-pong DMA transfers on the blocks would be to make the blocks 16 x 16; however, this would be significantly inefficient both when correlating the filters and when paging in and out of the frame. Because of this, we abandoned the idea of ping-pong DMA transfers for the blocks, and merely planned on paging one block of 64 x 64 complex interleaved data points.

11.2.1 Use Paging and parallelizing

The aforementioned memory allocation issues forced us to do only minimal paging, rather than ping-pong DMA transfer paging. Unfortunately, one DMA channel was always idle because of this, and the other channel was idle during all of the correlation calculations.

A solution to this problem was devised, however, when we thought about ping-pong DMA transferring in the filter, since only half of it could be stored in on-chip memory as it was. Rather than waiting for the half filter to finish correlating with half of the frequency domain block, transferring in the second half, and completing the correlation, we would create a ping-pong DMA transfer that split the filter into quarters. While one quarter of the filter was calculating the correlation, the next quarter was being transferred into on-chip memory in preparation of the next portion of the correlation calculation. Although this was never implemented, we are confident that it would represent a significant speed improvement in the overall system and would be an easy correction.

Section 11.3 Table of profile results

Table 15. Table of Profile Results

	Avg. Cycles	Time
1D FFT	1,506	0.06 ms
2D FFT	2.45 million	98.00 ms
Complex matrix transpose	101,840	4.07 ms
Filter multiplication	97,365	3.89 ms
2D correlation/block	12.1 million	484.00 ms

Section 12. Comparisons of Data and Results

Unfortunately, we are not able to show the results of our final running design because we were not able to fully debug our PC side code. However, there is extensive data shown in the algorithms section of our report.

Section 13. Analysis of Results/Future Improvements

Unfortunately, due to the complexity of our project we were unable to produce results for our EVM implementation. However, results using Matlab are detailed in the Algorithms section. Analysis of these results demonstrates a relatively acceptable level of performance among all classes but could be vastly improved on. False alarm and miss rates for the M1A2 and German Leopard tanks are strongly dependent on the separation achieved by the correlation filters. While there exists some separation, the current filter design and build lead to the system being consistently confused when presented with both tanks.

The alpha parameter of 0.99 chosen for the OTSDF filter did not prove to be a significant value in the sense that it provided markedly profound results. As such, there remains much analysis into the exact relationship between the alpha parameter and the performance of the algorithm. Perhaps more noise tolerance would benefit the algorithm in the presence of few pixels-on-target or the poor lighting conditions of the lab. The speculars presented by some of the models could be adapted for by modeling them as Gaussian white noise.

The parameters of the Gaussian distributions used in the Matlab simulations were chosen as a direct result of observation of maximum possible position displacement of each class. However, this in no way suffices for a scientific analysis and as such leaves room for more work. Closer analysis of the nature of the target classes will yield a better modeling of their movement via these Gaussian distributions. These distributions could and should be adaptable to target range.

Section 14. Problems encountered

During our exploration of methods to efficiently and effectively perform large size correlations on the EVM we considered the use of fixed point precision and associated functions such fixed point FFT's.

Section 14.1 Fixed point precision

Fixed point precision offers many advantages such as reduced data size and improved processing speed due to smaller data size. However, it comes at the cost reduced precision. Taking this into account we considered the use of DSP functions from the C62x library [11] which included a radix-4 fixed point FFT. This function necessitated the use of 16-bit data in Q15 format. Q15 format [12] refers to the use of 16 bits to represent numbers between -1 and 1. This meant that the most significant bit (MSB) represented the sign bit while the remaining fifteen bits represented fractional bits. Such a format requires conversion from standard floating point formats to Q15. In essence this meant that all of our floating point numbers should be between -1 and 1 to insure that they would be properly represented in Q15. However, conversion between these two formats is not intuitive although we eventually determined a reliable conversion. Still, such a conversion caused quantization error in terms of both the FFT and the application of the filter which we found to be unacceptable.

Another issue concerning fixed point precision is whether the C67x DSP is capable of performing proper fixed point operations. Specifically we were concerned with the multiplication of Q15 numbers. In typical integer multiplication, the necessary bit growth does no affect the value that the least significant bits (LSB) represent since the bit growth grows from the MSB. However, in Q15 or fractional multiplication the bit growth occurs from the LSB and does change the values that the other bits represent. This was not an issue on the C62x DSP which is a fixed point processor designed for Q15 format and as such is designed to accommodate for fractional bit growth. Our concern with multiplication mainly addressed the application of our correlation filters to the resulting FFT of a block. This could be addressed by conversion of Q15 numbers into float and allowing the EVM to properly grow the resulting product. However, this does not address the internal working of the fixed point FFT being used from the C62x DSP library. Since the fixed point FFT was designed to work on the fixed point processor of the C62x DSP, we could not be sure of its proper functioning on the C67x DSP considering the integral part that multiplication plays in calculation of the FFT.

Repeated attempts at solving these problems yielded much insight into the nature of finite precision and quantization effects along with some innovative solutions to some of these problems. However, we were not able to achieve satisfactory results with respect to the FFT and application of the correlation filters. The end result of our attempts with finite precision was the conclusion that it was not a feasible option in our application concerning the precision required. After we reached this conclusion, we returned to using floating point functions which required us to rework our EVM implementation.

Section 15. Code Referenced online

We had references several sources online for different parts of our code. The GUI was mainly done using the help pages in MSDN. Other code that we implemented are a mixed-radix FFT in our EVM emulator, an FFT shift that works for odd as well as even sized blocks, and a priority queue from Visual C++ libraries.

Section 15.1 Webcam

The base code for the GUI was a sample program, StillCap, in the DirectX SDK in the Direct Show API. Its original function was to capture feed from webcam in the form of a single 24bit RGB bitmap frame or to capture a stream in an AVI file. [14] Some of the added benefits of this code were that it had a preview screen which continuously updated the view from the webcam and the fact that the application runs in a modal dialog box. The advantage of running a modal dialog box is that the appearance and user control interfaces is easily changeable in Visual Studios using the resource editor.

This code had to be adapted to include the additional controls we wanted to implement such as the ability to choose targets, starting and stopping experiments, and various other options. Additionally, we added a frame to display the tracking resulting on the right side of the GUI.

Section 15.2 FFT Shift and Mixed Radix FFT

Originally we tried to create our own FFT shift code; however, it didn’t work for a correlation plane with odd dimensions. We found an algorithm for the FFT shift online which worked correctly for both odd and even dimensions. [15]

For our EVM emulator, we needed to implement an FFT to do the correlation. Since the code runs on the PC, we needed to find a good FFT online. No changes were made to the code. [16]

Section 16. References

[1] http://www.ikko.k.hosei.ac.jp/~matlab/matkatuyo/vcapg2.htm

[2] Kerekes, R., Narayanaswamy, B., and Beattie, M. "Efficient target tracking with correlation filters." Spring, 2005.

[3] B.V.K. Vijaya Kumar, “Tutorial Survey of Composite Filter designs for Optical Correlators,” *Applied Optics*, vol. 31, pp. 4773-4801, 1992.

[4] B. V. K. Vijaya Kumar and L. Hassebrook. “Performance measures for correlation filters,” *Appl. Opt.*, vol. 29, pp. 2997- 3006, 1990.

[5] A. Mahalanobis, B.V.K. Vijaya Kumar, and D. Casasent, “Minimum average correlation energy filters,” *Applied Optics*, vol. 26:3633-3640, 1987.

[6] A. Mahalanobis, B.V.K. Vijaya Kumar, S. Song, S.R.F. Sims, and J.F. Epperson, “Unconstrained correlation filters,” *Applied Optics*, vol. 33:3751-3759, 1994.

[7] B. V. K. V. Kumar, D. Carlson, and A. Mahalanobis, “Optimal tradeoff synthetic discriminant function (OTSDF) filters for arbitrary devices,” *Opt. Lett.*, vol. 19, pp. 1556–1558, 1994.

[8] Bracewell, R.N. (1990). Numerical Transforms, Science, 248: 697-704

[9] Oppenheim, A. V. and Schafer R. W., *Discrete-Time Signal Processing*.

[10] Rabiner, L. R. (1989). “A tutorial on hidden Markov models and selected applications in speech recognition.” *Proc. IEEE,* 77 (2), 257-286.

[11] http://focus.ti.com/docs/apps/catalog/resources/appnoteabstract.jhtml?abstractName=spru657b

[12] http://www.mathworks.com/access/helpdesk/help/toolbox/tic6000/

[13] http://www.bobpowell.net/grayscale.htm

[14] DirectX SDK. http://www.msdn.microsoft.com

[15] http://www.dsprelated.com/showmessage/20790/1.php

[16] http://hjem.get2net.dk/jjn/fft.htm

Section 17. Code

Section 17.1 MATLAB Scripts

```
function H = otsdf(tr_set, alpha)

% H = otsdf(tr_set, alpha)
%
% Author: Ramu Bhagavatula
% Data Created: April 5, 2005
% Last Modified: April 9, 2005
%
% Purpose:
% Creates otsdf filter based on training set of images and alpha value
%
% Inputs:
% 1. tr_set = Cell array of training images
% 2. alpha = Number between 0 and 1 describing the ratio of noise tolerance
% to peak sharpness in the otsdf and uotsdf filters.
%
% Outputs:
% 1. H = ostdf filter that results from using the tr_set and alpha

[d1,d2] = size(tr_set{1});  % Dimensions of training images
N = length(tr_set);         % Number of training images
d = d1*d2;                  % Total dimensionality of images

X = zeros(d,N); % Initializes matrix whose columns are the FFTs of the training images

for(i = 1:N)                % Iterates through training images
   img = single(tr_set{i});
   buf = fft2(img);
   X(:,i)=buf(:);           % Stores FFT of each training images as columns in X
end

D = zeros(d,1);    % Initializes average spectral power density matrix
for(i = 1:N)       % Iterates through FFTs of training images
   D = D+X(:,i).*conj(X(:,i));   % Calculates total spectral power density of training images as a column
rather than a diagonal
end
D = D/N;    % Calculates average spectral power density matrix

C = ones(d,1);                              % Noise covariance matrix
beta = (alpha*D)+(sqrt(1-alpha^2)*C);       % Tradeoff between noise tolerance and peak sharpness
u = ones(N,1)*d1*d2;                        % Normalizes output of correlation planes to be 1

tmp = zeros(d,N);             % Initializes temp matrix for inv(T)*X
for(i = 1:N)                  % Iterates through FFTs of training images
   tmp(:,i) = X(:,i)./beta;    % Element by element division works as inverse because D is stored as a
column
end

V = X'*tmp;
h = tmp*inv(V)*u;      % Calculates otsdf filter
H = reshape(h,d1,d2);  % Reshapes filter to match dimensions of training images
```

Section 17.2 EVM Code

```
#include <stdio.h>
#include <stdlib.h>
#include <common.h>
#include <board.h>          /* EVM library */
#include <pci.h>            /* PCI communication library */
#include <dma.h>
#include <mathf.h>

typedef float T;

#define N_POINTS 64
#define HALF_BLOCK N_POINTS*N_POINTS
#define FULL_BLOCK N_POINTS*N_POINTS*2
#define ROW_SIZE N_POINTS*2
#define ROW_BYTES ROW_SIZE*sizeof(T)
#define HALF_SIZE_BYTES HALF_BLOCK*sizeof(T)
#define BLK_SIZE_BYTES FULL_BLOCK*sizeof(T)

//#pragma CODE_SECTION(loadFILTER, "FAST"); // probably don't want, the other three we
probably do
#pragma CODE_SECTION(fillwtable, "FAST");
#pragma CODE_SECTION(mkrevtable, "FAST");
#pragma DATA_SECTION(evmOutput, "NOTFAST");

unsigned short int i, j;
unsigned int currentNumFilters, currentNumBlocks, currentFilter;

// FFT CODE
T twiddle[N_POINTS*3/2];
int revtable[N_POINTS];           // Look-up table for digit-reversing
int rev_j;
int count = 0;

// I/O blocks, evmOutput was corrBlocks, evmInput was blocks
T evmOutput[FULL_BLOCK*100];  // arbitrary big length, to hold max number of blocks per
'frame', can be up to 128
T *evmInput;
T *filterBank;

// manipulation blocks
T rowTemp[ROW_SIZE];
T currentBlock[FULL_BLOCK];       // full block
T filter[HALF_BLOCK];           // half a full block
//T *fwdFilter = filter1(?);      // for future rolling DMA filters?
//T *transFilter = filter2(?);

/* Function prototypes */
int requestTRANSFER(void *buf, int size, int command);
int waitTRANSFER();
int dmaCopyBLOCK(void *src, void *dest, int numBytes, int chan);
int waitBlockTRANSFER();

// FFT CODE
void cfftr4_dif(float* x, float* w, short n); /* Prototype for FFT routine */
void Do2dCorrelationFloat(int numBlocks);
void fillwtable(int n);
void complexMatrixTranspose(float *x, int rows, int cols);
void mkrevtable(int n);
void loadFILTER(T *newFilter);
void loadHalfFILTER(T *newFilter);

/*
 * DMA transfer a new Filter
 *  Each filter is a 64 x 64 block of float numbers
 *  SIZE_FILTER_BYTES = BLOCK_SIZE_BYTES = 64*64*sizeof(float)
 */

int main(void)
{
	int maxNumBlocks, totalNumFilters;
	int done, reinitialize;
	char request[1];

	evm_init();		/* Initialize the board */
	pci_driver_init();	/* Call before using any PCI code */

	DMA_AUXCR = 0x00000010;	/* Set priority of HPI over CPU to avoid crashing */

	fillwtable(N_POINTS);
	mkrevtable(N_POINTS);

	while(1)
	{
		printf("\nStart\n");

		// Poll PC for maximum number of blocks to expect
		requestTRANSFER(request, 0, 0x01);
		maxNumBlocks = waitTRANSFER();

		// Allocate evmOutput and evmInput in external memory; evmOutput already
allocated for 100 blocks
		evmInput = (T *)malloc(maxNumBlocks*BLK_SIZE_BYTES);
//		evmOutput = (T *)malloc(maxNumBlocks*BLK_SIZE_BYTES);

		if(evmInput == NULL || evmOutput == NULL) {
			printf("\nError #1!\n");
			exit(1);
		}

		// Poll PC for total number of filters available
		requestTRANSFER(request, 0, 0x01);
		totalNumFilters = waitTRANSFER();

		// Allocate memory for filters
		filterBank = (T *)malloc(totalNumFilters*BLK_SIZE_BYTES);
		if(filterBank == NULL) {
		     printf("\nError #2!\n");
		    exit(1);
		}

		// Ask PC to send filters
		requestTRANSFER(filterBank, totalNumFilters*BLK_SIZE_BYTES, 0x02);
		waitTRANSFER();

		// Poll PC as to whether or not process is done
```

```
                requestTRANSFER(request, 0, 0x05);
                done = waitTRANSFER();

                while(!done)
                {
                        // Poll PC about how many filters to apply to incoming set of blocks
                        requestTRANSFER(request, 0, 0x01);
                        currentNumFilters = waitTRANSFER();

                        // Poll PC about how many blocks are being sent
                        requestTRANSFER(request, 0, 0x01);
                        currentNumBlocks = waitTRANSFER();

                        // Ask PC to send block
                        requestTRANSFER(evmInput, currentNumBlocks*BLK_SIZE_BYTES, 0x03);
                        waitTRANSFER();

                        // Iterate over each filter being applied to current set of blocks
                        for(i = 0; i < currentNumFilters; i++)
                        {
                                //Poll PC as to which filter to apply currently outt of total
number to apply
                                requestTRANSFER(request, 0, 0x01);
                                currentFilter = waitTRANSFER();

                                // Load specified filter
                                loadFILTER(filter);

                                // Perform 2D overlap-save correlation on current set of blocks
using current filter
                                Do2dCorrelationFloat(currentNumBlocks);

                                // Ask PC to retrieve correlated blocks using current filter

                                requestTRANSFER(evmOutput, currentNumBlocks*BLK_SIZE_BYTES,
0x04);
                                waitTRANSFER();
                }
        }
        // Poll PC if user wishes to start entire process over again, reinitialize the tool
        requestTRANSFER(request, 0, 0x06);
                reinitialize = waitTRANSFER();

                if(!reinitialize)
                        break;
        }
        // Free allocated memory
        free(evmInput);
        free(filterBank);
}

/* Use mailbox 1 for address, 2 for size, and 3 for command */
int requestTRANSFER(void *buf, int size, int command) {
    amcc_mailbox_write(2, size);
    amcc_mailbox_write(3, command);
        pci_message_sync_send((unsigned int)buf, FALSE);
        return(0);
}
```

```
/* The PC will send a message when the transfer is complete.  Wait
       for that to happen */
int waitTRANSFER() {
       unsigned int value;

       pci_message_sync_retrieve(&value);
       return(value);
}

/* dma_copy_block: Copies numBytes from src to dest using DMA channel
       chan.  chan can be 0 or 1.  this function is ASYNCHRONOUS!  You must
       poll DMA0_TRANSFER_COUNT (or DMA1_TRANSFER_COUNT) to see when the
       transfer is complete */
/* Only valid for numBytes < 4 * 0xFFFF */
int dmaCopyBLOCK(void *src, void *dest, int numBytes, int chan) {
       unsigned int dma_pri_ctrl=0;
       unsigned int dma_tcnt=0;

  /* Give DMA priority over CPU, and increment src and dest
      after each element */
       dma_pri_ctrl = 0x01000050;

  /* One frame, and we're using 4 byte elements */
       dma_tcnt = 0x00010000 | (numBytes/4);

  /* Write to DMA channel configuration registers */
       dma_init(chan,
           dma_pri_ctrl,
           0,
           (unsigned int) src,
           (unsigned int) dest,
           dma_tcnt);

       DMA_START(chan);
       return(OK);
}

void loadFILTER(T *newFilter) {
       // Start DMA transfer for new filter
       dmaCopyBLOCK(&(filterBank[currentFilter*FULL_BLOCK]), newFilter, HALF_SIZE_BYTES, 0);
       while(DMA0_XFER_COUNTER);
       // Done!
}

void loadHalfFILTER(T *newFilter) {
       // Start DMA transfer for new filter
       dmaCopyBLOCK( &(filterBank[currentFilter*FULL_BLOCK+HALF_BLOCK]), newFilter,
HALF_SIZE_BYTES, 0);
       while(DMA0_XFER_COUNTER);
       // Done!
}


// Perform 2D overlap-save correlation using the current set of blocks and varying number of
filters
void Do2dCorrelationFloat(int numBlocks)
{
	int blockNum;
	float temp;

	// Iterate through current set of blocks
	for (blockNum=0; blockNum<numBlocks; blockNum++)
	{
		// DMA copy in the current block from external memory
		dmaCopyBLOCK(&(evmInput[FULL_BLOCK*blockNum]), currentBlock, BLK_SIZE_BYTES,
0);
		while(DMA0_XFER_COUNTER);
		count++;

		// Begin 2D FFT
		// 1. FFT along rows
		// 2. Digit Reverse
		// 3. Transpose
		// 4. FFT along rows (essentially columns of original matrix)
		// 5. Digit Reverse
		// 6. Divide by N

		// FFT along rows
		for(i=0;i<N_POINTS;i++)
			cfftr4_dif(&(currentBlock[i*ROW_SIZE]), twiddle, N_POINTS);

		// Digit reverse after first set of FFT's
		for(i=0; i<N_POINTS; i++)
		{
			for(j=0;j<N_POINTS;j++)
			{
				rev_j = revtable[j];      // rev_j is now the digit-reversal of
j

				rowTemp[2*j  ] = currentBlock[i*ROW_SIZE+2*rev_j];
				rowTemp[2*j+1] = currentBlock[i*ROW_SIZE+2*rev_j + 1];
			}
			memcpy(&(currentBlock[i*ROW_SIZE]),rowTemp,ROW_BYTES);
		}

		// Complex interleaved float transpose
		complexMatrixTranspose(currentBlock,N_POINTS,N_POINTS);

		// FFT along rows (essentially FFT along columns of original matrix)
		for(i=0;i<N_POINTS;i++)
			cfftr4_dif(&(currentBlock[i*ROW_SIZE]), twiddle, N_POINTS);

		// Digit reverse after second set of FFT's
		for(i=0; i<N_POINTS; i++)
		{
			for(j=0;j<N_POINTS;j++)
			{
				rev_j = revtable[j];      // rev_i is now the digit-reversal of
i

				rowTemp[2*j  ] = currentBlock[i*ROW_SIZE+2*rev_j];
				rowTemp[2*j+1] = currentBlock[i*ROW_SIZE+2*rev_j + 1];
			}
			memcpy(&(currentBlock[i*ROW_SIZE]),rowTemp,ROW_BYTES);
		}
		// Divide by N (premepts divide by N necessary later and also helps to prevent
overflow)
		for(i=0;i<N_POINTS;i++)
		{
			for(j=0;j<N_POINTS*2; j++)
			{
				currentBlock[i*ROW_SIZE+j] /= N_POINTS;
			}
		}

		// Begin correlation
		// 1. Load first/top half of filter
		// 2. Apply first/top half of filter
		// 3. Load second/bottom half of filter
		// 4. Apply second/bottom half of filter

		// Load first/top half of current filter
		loadFILTER(filter);

		// Apply first/top half of current filter to current block
		for (i=0; i<N_POINTS/2; i++)
		{
			for (j=0; j<2*N_POINTS; j+=2)
			{
				// Foil and conjugate for IFFT
				temp                         = currentBlock[i*ROW_SIZE+j] *
filter[i*ROW_SIZE+j  ] + currentBlock[i*ROW_SIZE+j+1] * filter[i*ROW_SIZE+j+1];
				currentBlock[i*ROW_SIZE+j+1] = currentBlock[i*ROW_SIZE+j] *
filter[i*ROW_SIZE+j+1] - currentBlock[i*ROW_SIZE+j+1] * filter[i*ROW_SIZE+j  ];
				currentBlock[i*ROW_SIZE+j  ] = temp;
			}
		}

		// Load second/bottom half of current filter
		loadHalfFILTER(filter);

		// Apply second/bottom half of current filter to current block
		for (i=0; i<N_POINTS/2; i++)
		{
			for (j=0; j<2*N_POINTS; j+=2)
			{
				// Foil and conjugate for IFFT
				temp                                    =
currentBlock[i*ROW_SIZE+j+HALF_BLOCK] * filter[i*ROW_SIZE+j  ] +
currentBlock[i*ROW_SIZE+j+1+HALF_BLOCK] * filter[i*ROW_SIZE+j+1];
				currentBlock[i*ROW_SIZE+j+1+HALF_BLOCK] =
currentBlock[i*ROW_SIZE+j+HALF_BLOCK] * filter[i*ROW_SIZE+j+1] -
currentBlock[i*ROW_SIZE+j+1+HALF_BLOCK] * filter[i*ROW_SIZE+j  ];
				currentBlock[i*ROW_SIZE+j+HALF_BLOCK  ] = temp;
			}
		}


		// Begin 2D IFFT
		// Note: Due to conjugation done in application of the filter, it is unecessary
to conjugate before applying FFT to achieve IFFT
```

```
		// 1. FFT (IFFT) along rows
		// 2. Digit Reverse
		// 3. Divide by N
		// 4. Transpose
		// 5. FFT along rows (essentially columns of original matrix)
		// 6. Digit Reverse

		// FFT (IFFT) along rows
		for(i=0;i<N_POINTS;i++)
			cfftr4_dif(&(currentBlock[i*ROW_SIZE]), twiddle, N_POINTS);

		// Digit reverse after first set of FFT's (IFFT's)
		for(i=0; i<N_POINTS; i++)
		{
			for(j=0;j<N_POINTS;j++)
			{
				rev_j = revtable[j];	// rev_i is now the digit-reversal of
i

				rowTemp[2*j  ] = currentBlock[i*ROW_SIZE+2*rev_j];
				rowTemp[2*j+1] = currentBlock[i*ROW_SIZE+2*rev_j + 1];
			}
			memcpy(&(currentBlock[i*ROW_SIZE]),rowTemp,ROW_BYTES);
		}

		// Divide by N
		for(i=0;i<N_POINTS;i++)
			for(j=0;j<N_POINTS*2; j++)
				currentBlock[i*ROW_SIZE+j] /= N_POINTS;

		// Complex interleaved float transpose
		complexMatrixTranspose(currentBlock,N_POINTS,N_POINTS);

		// FFT (IFFT) along rows (essentially columns of original matrix)
		for(i=0;i<N_POINTS;i++)
			cfftr4_dif(&(currentBlock[i*ROW_SIZE]), twiddle, N_POINTS);

		// Digit reverse after second set of FFT's (IFFT's)
		for(i=0; i<N_POINTS; i++)
		{
			for(j=0;j<N_POINTS;j++)
			{
				rev_j = revtable[j];	/* rev_i is now the digit-reversal of
i */

				rowTemp[2*j  ] = currentBlock[i*ROW_SIZE+2*rev_j];
				rowTemp[2*j+1] = (currentBlock[i*ROW_SIZE+2*rev_j + 1]);
			}
			memcpy(&(currentBlock[i*ROW_SIZE]),rowTemp,ROW_BYTES);
		}

		// DMA copy the correlated block into external memory
		dmaCopyBLOCK(currentBlock, &(evmOutput[FULL_BLOCK*blockNum]), BLK_SIZE_BYTES,
1);
		while(DMA1_XFER_COUNTER);
	}
}
```

```
// Fill in twiddle table needed for FFT
void fillwtable(int n) {

	int k;
	float constant=2.0*(float)PI/(float)n;

	for (k=0; k<3*n/4; k++)
	{
		twiddle[2*k] = cosf(constant*(float)k);
		twiddle[2*k+1] = sinf(constant*(float)k);
	}
}

// Complex interleaved matrix float transpose
void complexMatrixTranspose(float *x, int rows, int cols)
{
	float tempReal, tempImag;
	int i, j;

	// Iterate through rows
	for(i=0;i<rows;i++)
	{
		// Iterate through columns
		for(j=i+1;j<cols;j++)
		{
			// Store current complex value
			tempReal=x[i*2*cols+2*j];
			tempImag=x[i*2*cols+2*j+1];
			// Transfer in transposed complex value into current (row, column)
			x[i*2*cols+2*j]=x[j*2*cols+2*i];
			x[i*2*cols+2*j+1]=x[j*2*cols+2*i+1];
			// Transfer out current complex value in transposed (row, column)
			x[j*2*cols+2*i]=tempReal;
			x[j*2*cols+2*i+1]=tempImag;
		}
	}
}

/* This function creates the lookup table for digit reversing.  After
   it is run, revtable[n] equals the pairwise digit-reversal of n.
   n is the size of the FFT this table will be used for.*/
void mkrevtable(int n) {
  int bits, i, j, r, o;

  bits= (31 - _lmbd(1, n))/2;  /* _lmbd(1,n) finds leftmost 1 bit in n */
  for(i=0; i<n; i++) {
    r=0; o=i;
    _nassert(bits>=3);
    for(j=0; j<bits; j++) {
      r <<= 2;
      r |= o & 0x03;
      o >>= 2;
    }
    revtable[i] = r;
  }
}
```

Section 17.3 PC algorithm

```
int main(int argc, char* argv[])
{
 int programExit = 0;
 int i = 0;
 int done = 0;
 int reinitialize = 0;
 TS ts;     // why was this commented out?

 colThresh = 4*100000000;
 rowThresh = 4*100000000;
 corrThresh = (float) 0.9;

 totalNumTargetsDetected = 0;

 initializeEVM();

 loadEXPERIMENT();

// loadGAUSSIANS();

 totalNumFilters = NUM_SCALES*totalNumClasses;
 filterBank = (EVM_TYPE *)
malloc(BLK_SIZE*2*totalNumFilters*sizeof(EVM_TYPE));
 filterDims = (unsigned int *)
malloc(totalNumFilters*NUM_SCALES*sizeof(int));


 loadFILTERS();

 maxNumBlocks = ((((MAX_WINDOW_SIZE)/(BLK_DIM-
MAX_FILT_SIZE+1))*((MAX_WINDOW_SIZE)/(BLK_DIM-MAX_FILT_SIZE+1)))+1);
 blocks = (EVM_TYPE *)
malloc(maxNumBlocks*2*BLK_SIZE*sizeof(EVM_TYPE));
 corrBlocks = (EVM_TYPE *)
malloc(maxNumBlocks*2*BLK_SIZE*sizeof(EVM_TYPE));

 waitREQUEST(&ts);
 fprintf(stderr, "Transfer request: CMD %x, SIZE %i, ADDRESS %x\n",
ts.command, ts.size, ts.buffer );
 if(!evm6x_send_message(hBd, (PULONG)&maxNumBlocks)) // #1
 {
  fprintf(stderr, "Send message error!\n");
  exit(1);
 }

// PCItransfer(myEVM.maxNumBlocks , maxNumBlocks);  // EVM
communication
// EVM_malloc_blocks(&myEVM);

 waitREQUEST(&ts);
 fprintf(stderr, "Transfer request: CMD %x, SIZE %i, ADDRESS %x\n",
ts.command, ts.size, ts.buffer);
 if(!evm6x_send_message(hBd, (PULONG)&totalNumFilters)) // #2
 {
  fprintf(stderr, "Send message error!\n");
  exit(2);
 }
```

```
// PCItransfer( myEVM.totalNumFilters , totalNumFilters);
// EVM_malloc_filter_bank(&myEVM);

 waitREQUEST(&ts);
 fprintf(stderr, "Transfer request: CMD %x, SIZE %i, ADDRESS %x\n",
ts.command, ts.size, ts.buffer);
 if(ts.command==0x02)
 {
  if(ts.size != (BLK_SIZE*2)*totalNumFilters*sizeof(EVM_TYPE))
   fprintf(stderr, "Wrong size!!!\n");

  START_TIMER;
  sendDATA(&ts, &(filterBank[0]));
  STOP_TIMER;
  fprintf(stderr, "Elapsed Time for Send Filter Bank: %f\n",
elapsed_time());
 }

// HPItransfer(myEVM.filterBank, filterBank,
(BLK_SIZE*2)*totalNumFilters*sizeof(EVM_TYPE));
 frameNum = 0;

 while(1)
 {
  if(!loadFRAME())
   done = 1;
  else
   done = 0;

  waitREQUEST(&ts);
  fprintf(stderr, "Transfer request: CMD %x, SIZE %i, ADDRESS %x\n",
ts.command, ts.size, ts.buffer);
  if(!evm6x_send_message(hBd, (PULONG)&done))
  {
   fprintf(stderr, "Send message error!\n");
   exit(19);
  }

//  PCItransfer( myEVM.done , done);
//  EVM_check_done(&myEVM);

  if(done)
   break;

  //if(frameNum%DETECT_STEP==0)
  if((frameNum == 0) || (!targets))
  {
   fprintf(stderr, "Entering Detection Stage...\n");
   detectionSTAGE();
   //applyGAUSSIAN();
   writeFrameINFO();
  }
  else if(frameNum%FRAME_STEP == 0)
  {
   fprintf(stderr, "Entering Tracking Stage\n");
   trackingSTAGE();
   //applyGAUSSIAN();
   writeFrameINFO();
  }
  //writeFrameINFO();
  frameNum++;
 }
 waitREQUEST(&ts);
 fprintf(stderr, "Transfer request: CMD %x, SIZE %i, ADDRESS %x\n",
ts.command, ts.size, ts.buffer);
 if(!evm6x_send_message(hBd, (PULONG)&reinitialize))
 {
  fprintf(stderr, "Send message error!\n");
  exit(20);
 }
// PCItransfer( myEVM.reinitialize, reinitialize );
 free(blocks);
 free(corrBlocks);
 free(filterBank);
 free(filterDims);
 if(targets)
  freeTargets(targets);

 fprintf(stderr, "Freeing allocated memory\n");
 /* Clean up and exit */
 if (!evm6x_hpi_close(hHpi))
 {
  fprintf(stderr, "Error closing connection to EVM!\n");
  exit(13);
 }
 if (!evm6x_close(hBd))
 {
  fprintf(stderr, "Error closing connection to EVM!\n");
  exit(14);
 }
 fprintf(stderr, "Closed connection to EVM\n");
 return(0);
}
```