An Implementation of Vehicle Tracking Using Motion Layers

Rachel A. Kitzmann

Masters Project 1051-840
08 May 2013

ABSTRACT

A vehicle tracking algorithm was applied to airborne surveillance imagery taken using the ITT Exelis Wide Area Airborne Surveillance (WAAS) sensor. The algorithm utilizes a Kinade-Lucas-Tomasi (KLT) algorithm for detecting and tracking the "good" features, which are defined as features that remain similar over time, in the image. These "good" features are then added to motion layers that are constrained by size, shape, and consistency of motion to be "vehicle-like." The motion layer algorithm is more robust than the KLT algorithm alone because individual KLT points can be lost due to occlusion, but as long as some points remain in the motion layer, the layer will continue to be tracked.

Keywords: KLT, motion layer, vehicle detection, feature tracking

1. INTRODUCTION

Aerial photography first made an appearance in 1858, prior to the Wright era of modern flight, when Parisian photographer Gaspard-Félix Tournachon (pseudonym, "Nadar") used a tethered balloon to photograph Val De Bievre, near Paris. Airplanes first carried cameras in 1908, approximately 5 years after the Wright Brother's first flight, when a photographer accompanied Wilbur Wright and obtained the first aerial motion pictures over Le Mans, France. The military took interest soon after, and aerial photography was used to capture over one million reconnaissance images during World War I [1].

Today, the field of remote sensing has expanded greatly due to advances in both sensors, and aircraft and spacecraft. Google Earth [2] has become a household name, allowing the public access to 2-D and 3-D imagery for navigation and education. Military applications of remote sensing now include defense mapping [3], surveillance, reconnaissance, and target detection, tracking, and persecution.

There are many sensor solutions available to both the military and law enforcement that will achieve quality target detection and tracking. Active sensors, such as radar, can provide excellent target ranging ability. However, with the advent of modern radar jamming techniques [4], alternative passive solutions are growing in popularity. Passive sensor solutions often have multiple sensor bands available, such as panchromatic (visible) and infra-red, allowing for both day and night sensing. Targeting pods, such as the Northrop Grumman LITENING [6] and Lockheed Martin SNIPER [5] pods are optimized for target detection, tracking, and persecution. However, due to their relatively small fields of view (FOV) (1024 x 1024 pixel FOV for the LITENING CCD sensor [6]), the sensor line of sight must be continuously updated to stay located on the target of interest. This prevents the user from being able to stay situationally aware, since they are primarily focused on a single target. Targeting pods such as these cannot compete with a large area surveillance sensor, such as Wide-Area Airborne Surveillance (WAAS) sensor offered by ITT Exelis, when it comes to situational awareness.

The WAAS sensor has the ability to operate as a high-resolution visual band panchromatic sensor, as well as a mid-wave infra-red sensor. Multiple cameras operate in each band, which allows multiple small, high resolution images taken by each camera to be combined together by an on-board image processor into one large mosaicked image. Depending on altitude, the final image can span several ground kilometers, allowing for detection and track of multiple targets over a large area. This enables the user to track specific threats over time, to understand the history of the threat's movements, and to possibly predict the future movements of the threat based on the best ingress/egress routes to the area of interest.

This project focuses on detection and track of vehicles present in imagery obtained by the ITT Exelis WAAS sensor. The tracking algorithm implemented utilizes a Kinade-Lucas-Tomasi (KLT) tracker [8] optimized to detect and track the "good" feature points in the image. The theory behind the KLT tracker is discussed in Section 2. Once good features are detected they are assigned to "motion layers." This essentially turns the KLT point tracker into an area tracker using the

method of Cao *et al.* [9], which is described in Section 3. The specific details behind this project's implementation are described in Section 4, with a discussion of the results and conclusions provided in Sections 5 and 6.

1.1 Project Objectives

This project was chosen with the following goals in mind:

- To apply the techniques investigated in this project to real-life sensor applications encountered in industry.
- To apply the knowledge gained through work experience to an academic project.
- To gain a deeper understanding of image processing techniques and their applications to specific problems.

2. KLT TRACKER

Lines, edges, and corners in an image can be identified using many well-studied algorithms. Some detectors, such as the Laplacian line detector, use sudden changes in local intensity to identify the edges of an object. Other edge detectors such as the Roberts, Prewitt or Sobel operators utilize a gradient operator in order to find both the edge strength as well as the edge's direction [7]. However, even if a region in an image is rich in texture (*i.e.*, many lines, edges or corners available), the region could still contain few features good for tracking. For example, consider the photograph of barren tree branches shown in Figure 1. Corners are created by the crossing tree branches (circled in yellow), however these are non-physical features formed by the current camera-target orientation. If these crossings were tracked across several frames (at times t_1, t_2, etc.) using a moving camera, the features could disappear due to the change in geometry. Additionally, detectors that primarily use changes in intensity can be prone to errors caused by changes in shadows, and reflections off of shiny surfaces.

Figure 1: Crossing Tree Branches Indicating "Corners" Across a Depth Discontinuity. Photo courtesy of www.photos-public-domain.com

In order to avoid these potential problems, the KLT tracker [8] measures a feature's dissimilarity over time. Dissimilarity is defined as the feature's RMS residual between the first and current frame. Once the dissimilarity grows too large, the feature is discarded. Using dissimilarity in this manner allows the KLT feature tracker to be insensitive to image rotation, changes in illumination, and image noise.

According to [8], changes in intensity within a window in an image can be described as motion of the window:

$$I(x, y, t + \tau) = I(x - \xi(x, y, t, \tau), y - \eta(x, y, t, \tau)) \quad (1)$$

So, at time t+τ, the location of every point in the new image can be determined by moving every point $\boldsymbol{x} = (x, y)$ in the original image taken at time t, by the displacement, $\delta = (\xi, \eta)$. The displacement δ is represented as an affine motion field that can account for both translation as well as warping of the image:

$$\delta = D\boldsymbol{x} + \boldsymbol{d}, \tag{2}$$

where

$$D = \begin{bmatrix} d_{xx} & d_{xy} \\ d_{yx} & d_{yy} \end{bmatrix} \tag{3}$$

is a deformation matrix, and **d** is the translation of the feature window's center. Therefore, every point **x** in the first image *I* moves to point A**x**+**d** in the second image *J*, where A=**1**+D, and **1** is a 2x2 identity matrix. This motion can then be written as:

$$J(A\boldsymbol{x} + \boldsymbol{d}) = I(\boldsymbol{x}). \tag{4}$$

The KLT tracker finds the affine parameters A and **d** that will minimize the dissimilarity, given by

$$\epsilon = \iint_W [J(A\boldsymbol{x} + \boldsymbol{d}) - I(\boldsymbol{x})]^2 w(\boldsymbol{x}) d\boldsymbol{x}, \tag{5}$$

where W is the feature window, and w(x) is an optional weighting function.

Minimizing the residual in Equation (5) reduces to creating the linear system

$$\mathrm{T} = \iint_{\mathrm{W}} \begin{bmatrix} \mathrm{U} & \mathrm{V} \\ \mathrm{V^T} & \mathrm{Z} \end{bmatrix} \mathrm{w}\, \mathrm{d}\mathbf{x}, \tag{6}$$

which is derived in detail in reference [10]. The matrix Z can be derived as:

$$Z = \begin{bmatrix} g_x^2 & g_x g_y \\ g_x g_y & g_y^2 \end{bmatrix},$$

where g_x and g_y represent the spatial gradient of the image intensity with respect to x and y, respectively.

In order to track the window of interest from frame to frame, the eigenvalues of matrix Z must be large enough to be above the image noise level, and cannot differ by several orders of magnitude. Two small eigenvalues would indicate an approximately constant intensity across the image, and therefore no edges or corners. One large and one small eigenvalue represent unidirectional texture. Two large eigenvalues represent corners, or other patterns that can be tracked reliably [8]. According to [8], if the smallest of the two eigenvalues is large enough to meet the noise threshold then, in practice, a single threshold, λ, for the two eigenvalues can be used, and we accept the feature if

$$\min(\lambda_1, \lambda_2) > \lambda. \tag{7}$$

The KLT algorithm described above has been implemented in public domain source code written in the C-programming language by Stan Birchfield [11], and was used in this project for detection and track of the KLT features. The Birchfield implementation is very flexible in the routines and parameters that can be called in order to tune the tracker output to the imagery of interest. This flexibility was utilized by this project in order to separately optimize the KLT output for the image registration step and the motion layer detection step as described in Section 3. The selectable parameters altered during the course of this project include:

- Maximum number of desired features
- Minimum distance (in pixels) between selected features
- Minimum eigenvalue (λ) allowed for feature selection

3. MOTION LAYERS

Compared to fixed camera surveillance, airborne video surveillance is highly mobile, allowing the user the freedom to view large areas, and to change the area of interest in real-time. However, detection and tracking of objects during airborne surveillance can be challenging; motion of the airborne platform causes the entire image, including background, to move from frame to frame, and objects within the image can change throughout the tracking process due to illumination changes and changes to the object's orientation.

Detection of moving vehicles within a large image frame is further complicated by the large amount of background clutter present in the scene. The moving vehicles of interest comprise only a small portion of the entire image, and often the intensity of the vehicle pixels is not significantly higher than the background. This results in a low signal-to-clutter ratio[1] (SCR) between the target of interest and the background. There are several techniques available to detect moving vehicles, such as [12], however these can cause delays in real-time processing due to the long image sequences necessary for target declaration.

The motion layer method for vehicle detection and tracking proposed by Cao *et al.* [9] has two major benefits over existing algorithms. First, use of the KLT algorithm for feature point detection coupled with creation of motion layers combines the best of two worlds; the KLT algorithm ensures that only "good" features (as described in Section 2) are detected and assigned to motion layers, and creation of these motion layers improves the robustness of the KLT point tracker by turning it into an area tracker. This allows the motion layer algorithm to maintain track on an object even if some of the KLT detected features are lost due to partial occlusion, or variations in intensity caused by shadow, or glare off of glossy objects. The second major benefit is that the motion layer algorithm claims to combine vehicle detection and tracking into a single step in order to speed the image processing in order to run in real-time. A flowchart of the method is shown in Figure 2.

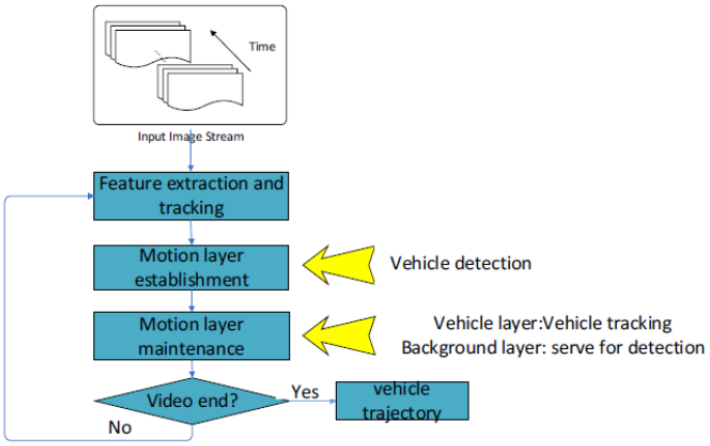

Figure 2: Flowchart of Cao *et al.* Motion Layer Tracking Method

In the method of Cao *et al.* [9], an initial set of features are found using the KLT method described in Section 2. The scene is divided into regions, with the KLT algorithm run separately in each region in order to avoid selecting too many feature points from a single region. The moving vehicle points are separated from the background points by examining differences in the motion[2] of the points between the image at time t_0 (i_{t0}) to the image at time t_1 (i_{t1}). It is assumed in [9] that the majority of the feature points found will lie on the background, due to the small target density within the total

[1] Similar to a signal-to-noise ratio, the term "signal-to-clutter ratio," or SCR, is often used by the radar community to describe the ratio of return signal power from the target of interest to the power of the return signal from the background. For our passive case we will use SCR to describe the intensity of the target with respect to the background.

[2] At this stage, it is not specified by Cao *et al.* [9] exactly how the background points are separated from moving target points. A coarse method based on examination of the histograms of the feature motion was used in this implementation and is described in Section 4.

scene. Therefore, the majority of the KLT points in the region will share the same background motion, with moving targets being outliers. These outliers are eliminated such that the remaining points can be used as control points for the image registration. The average motion of the background features is then used to compute the affine transformation needed, per the KLT method, to register the image i_{t0} to the image i_{t1}.

Once the images are registered, temporal differencing is used to find the moving targets. Temporal differencing (TD) is the result of subtracting two images taken from different times, t_1 and t_0:

$$\mathrm{d_{t1}(x,y) = i_{t0}(x,y) - i_{t1}(x,y) \quad for\ x = 0 \ldots N\ and\ y = 0 \ldots M,} \tag{8}$$

where x is the horizontal pixel location ranging from 0 to N, y is the vertical pixel location ranging from 0 to M, and NxM pixels represent the total image dimensions. If the image registration step is perfect, background pixels that have not moved will occur in the same (x,y) location in both i_{t0} and i_{t1}, so their resulting difference d_{t1} will be exactly zero. Pixels located on moving targets will have a non-zero difference, and can be easily identified.

Moving target motion layers are then established by assigning each non-zero differenced region that meets a certain size criteria to a coarse selection rectangle. The vehicle size used will vary based on sensor altitude and sensor aspect with respect to the ground, and must be tailored for a particular application. The value used for this implementation is given in Section 4. The KLT points that fall within the coarse selection rectangle are initially assigned to that particular motion layer.

Each motion layer is then refined by applying a consistency of motion measure. The KLT feature occurring closest to the center of the coarse selection rectangle is assumed to lie on the tracked vehicle. Because vehicles are rigid objects, all KLT points lying on the vehicle should share the same motion as the center point. The magnitude S and direction θ of motion for each pixel is given by the equations:

$$S = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2} \tag{9}$$

$$\theta = \frac{y_1 - y_0}{x_1 - x_0} \tag{10}$$

where (x_0,y_0) and (x_1,y_1) represent the pixel locations in i_{t0} and i_{t1} respectively. The motion similarity computed by [9] is then given as

$$MS = e^{|s_1 - s_0| + |\theta_1 - \theta_0|}, \tag{11}$$

where s_1 and θ_1 are the motion magnitude and direction for the pixel of interest, and s_0 and θ_0 are the motion magnitude and direction of the pixel closest to the coarse rectangle center. If the value of *MS* is large, the point is eliminated from the motion layer.

At this point, both the background motion layer and an initial set of moving vehicle motion layers have been established. In order to speed processing, these image registration and TD steps for detecting and establishing new motion layers are only repeated every several frames. The frequency of the image registration and TD steps is not specified by [9] because it is dependent on the particular implementation, and can be tuned to meet the detection requirements of a particular application. The required frequency of new vehicle detection depends on the rate at which the sensor captures images, as well as the velocity of the moving targets of interest. If the image capture rate is low, or the moving target velocity is high, one could increase the frequency of the image registration and TD steps in order to ensure a high velocity target is present in the scene at least one time (and preferably several times) this detection process is performed.

Although the image registration and TD steps for target detection are only performed occasionally, the KLT algorithm for detection and tracking of individual features is performed every frame. Until the detection process is repeated, the motion layers must be maintained in order to add or remove KLT features that are either newly detected or not observed (*i.e.*, "lost") in the current frame. In order to maintain the motion layers, the average motion (S, θ) of the features of each layer is computed and the coarse motion layer rectangle is shifted by this average motion. The motion similarity measure is then used to test all KLT points found within the bounds of each coarse rectangle against the motion of the pixel closest to the center of the rectangle. This adds any newly detected points to the layer, while removing any points that were lost.

4. METHODOLOGY

The KLT and motion layer algorithms were applied to the electro-optic (EO) visual band imagery collected from the Wide-Area Airborne Surveillance Sensor (WAAS), developed by ITT Exelis. The full EO band imagery has dimensions of 4872 x 3248 pixels. In order to speed algorithm development, and to better view the resulting imagery, image subsets of 950 x 1200 pixels were created from the center of the full frame. The initial establishment of the motion layer and subsequent motion layer maintenance are described in detail in sections 4.1 and 4.2.

This project tracked moving vehicles through three images at times t_0, t_1, and t_2. These are hereafter referred to as images i_{t0}, i_{t1}, and i_{t2}. The following WAAS image files were used as images i_{t0}, i_{t1}, and i_{t2} respectively:

- 12AUG109Z0005900ZXEO0000NA0000004C643622.ntf
- 12AUG109Z0005901ZXEO0000NA0000004C643622.ntf
- 12AUG109Z0005902ZXEO0000NA0000004C643623.ntf

Each file was opened in ENVI, and a central 950 x 1200 pixel subset of each image was saved as a JPEG file in order to perform the KLT and motion layer processing.

4.1 Initial Motion Layer Establishment

The initial establishment of the motion layer was performed roughly in accordance with the theory described in Section 3, but was modified in order to utilize existing algorithms where possible. A flow chart of the actual implementation is shown in Figure 3. The main difference between this flow, and the one proposed by Cao *et al.* (Figure 2) is that we perform the KLT feature detection as two distinct phases; one which is optimized to select background features to perform image registration, and one which is optimized to select clustered target features. Both implementations, however, are designed to only perform the TD detection process once every several frames[3].

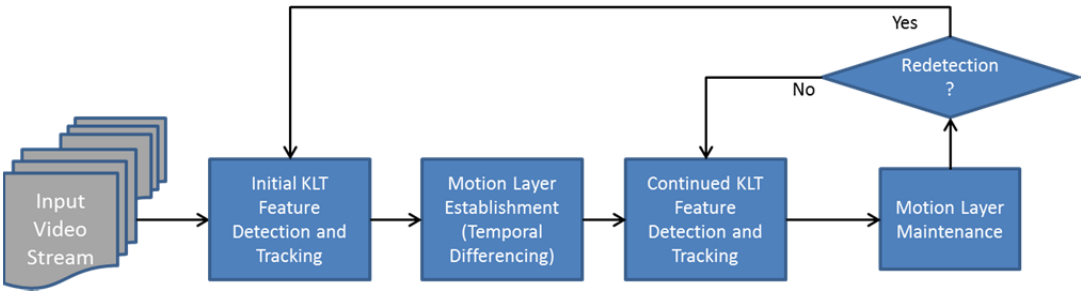

Figure 3: Flowchart of motion layer algorithm, modified to suit the described implementation

The initial establishment of the motion layer begins with running the KLT algorithm two different times, on images i_{t0} and i_{t1}. The first time, the minimum allowable eigenvalue was set to 150, and the minimum distance between features was set to 2 pixels. This allowed the KLT points to cluster on vehicles and other bright objects as shown in Figure 4. The second time, the minimum allowable eigenvalue was set to 300 and the minimum distance between features was set to 30 pixels. These settings prevent too many points from clustering on bright objects, and instead cause the points to spread throughout the background as shown in Figure 5.

[3] For this implementation, vehicles were only tracked through one iteration of the entire detection/maintenance cycle in order to prove the concept. The frequency at which the TD step is required to run in order to achieve the best detection performance was not investigated.

The motion of each returned KLT feature was then calculated assuming Euclidean geometry of the two scenes, per equations (9) and (10). Two histograms, one for motion magnitude and one for motion direction, were computed for the list of features in order to determine the most commonly occurring motion direction and magnitude[4]. It was assumed that the most often occurring motion would belong to background points. These histograms are shown in Figure 6 and Figure 7.

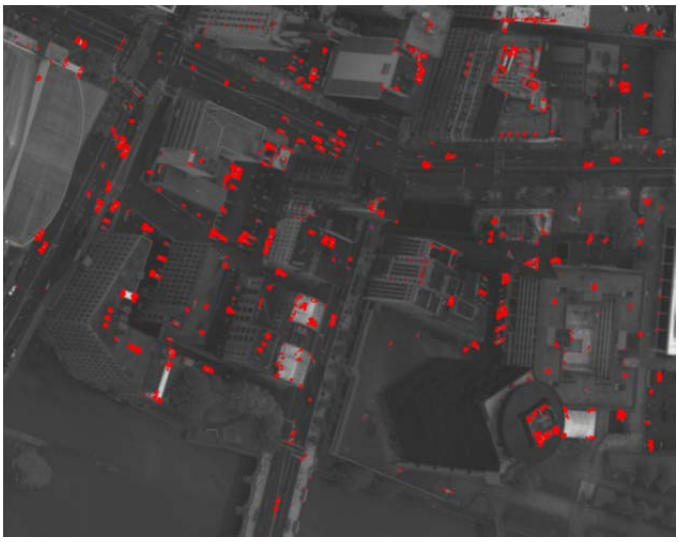

Figure 4: KLT results tuned for feature detection, with a 2-pixel minimum spacing between features and minimum allowable eigenvalue = 150. (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

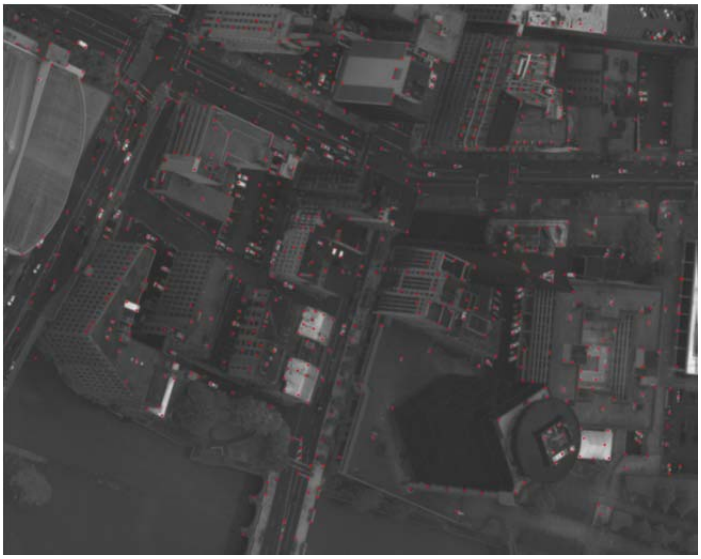

Figure 5: KLT Results tuned for background detection, with a 30-pixel minimum spacing between features and minimum allowable eigenvalue = 300. (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

[4] It should be noted that the large occurrence of "zero" for both the direction and motion of magnitude histograms is an artifact due to computing the histogram on a generically sized list that contained many empty rows. If the point array was sized to eliminate these zero rows, the true occurrence of zero values in the image would be small. Therefore, for this illustration, zero value can be ignored.

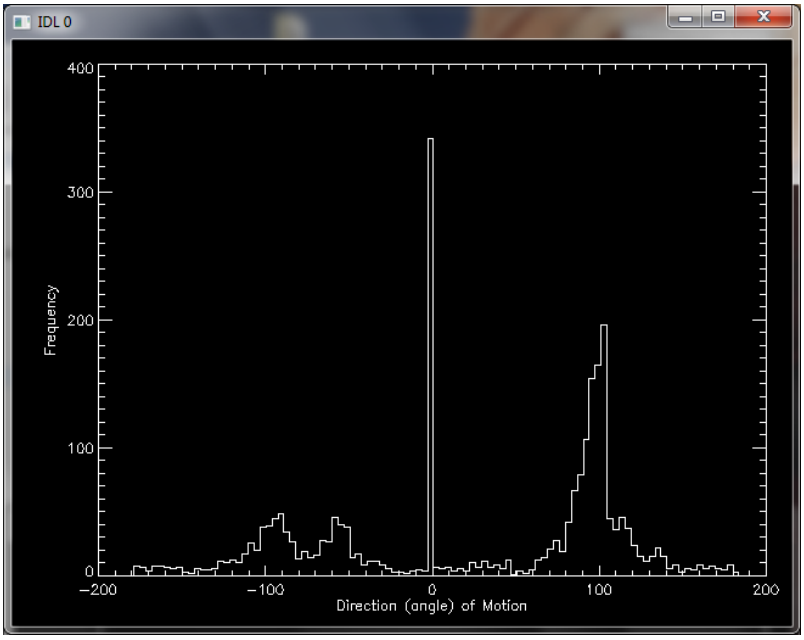

Figure 6: Histogram of motion direction for KLT features tuned for background detection

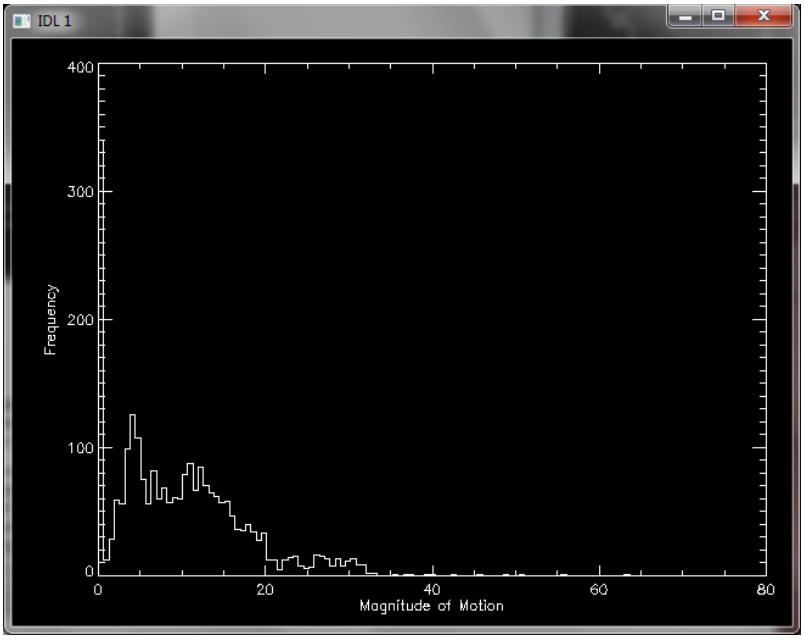

Figure 7: Histogram of motion magnitude for KLT features tuned for background detection

Based on the direction histogram, the initial KLT point list was reduced in order to coarsely eliminate movers that did not share the global background motion. For the example discussed here, because most of the KLT points were expected to lie on background, and because the histogram shows the most frequency occurring direction of motion was between 90^o and 110^o, all points having a direction of motion between 90^o and 110^o were categorized as background points. These points were then added to a ground control point (GCP) list.

The GCP list was then used, along with a built-in image registration routine in the ENVI software suite, to perform automatic registration and warping of image i_{t0} to i_{t1}. A second-order polynomial mapping was chosen in order to account for changes in translation and rotation, and also to approximate changes in variable scale (or perspective) between the two images[5]. Once i_{t0} was warped to match i_{t1}, the temporal differencing step was performed by subtracting the i_{t0} warp from i_{t1}. The differenced image was then scaled such that a grayscale value of 128 was assigned to pixels whose difference was zero. Pixels with negative differences were scaled between 0 and 128, while positive differences were scaled between 128 and 255. This resulted in the majority of background being displayed in mid-gray, the initial moving target location being displayed darker than mid-gray, and the new target location being displayed brighter than mid-gray. The resulting temporal differenced image is shown in Figure 8.

Figure 8: Temporal differenced image produced by subtracting image i_{t1} from the warp created when image i_{t0} was registered to i_{t1}. The image is scaled such that a difference of zero results in a mid-level gray intensity. (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

Next, the temporal differenced image was turned into a binary (black and white) image. The goal was to turn all pixels that were background and prior-frame target locations to black, while the target location in the current image, i_{t1}, would remain white. Otsu's method of automatic image thresholding was attempted, but produced an unacceptable threshold. According to Gonzalez and Woods [7], "The basic idea [behind Otsu's Method] is that well-thresholded classes should be distinct with respect to the intensity values of their pixels and, conversely, that a threshold giving the best separation between classes in terms of their intensity values would be the best (optimum) threshold." However, because there were relatively few bright areas compared to the image as a whole, and the targets of interest were not drastically brighter than the surroundings, Otsu's method produced a threshold unsuitable to this implementation. So, instead, a manually defined threshold was applied based on inspection of the digital count values of moving vehicles. A threshold digital count of 150 was selected[6] in an attempt to balance between removal of background and false elimination of potential motion layers. A morphological opening operation was then applied in order to improve the binary image by removing some of the noisier areas, such as the ones circled in red in Figure 9. The original thresholded image and morphologically opened image are shown in Figure 9 and Figure 10, respectively.

[5] A second-order polynomial only approximately accounts for changes in scale, and works best if the scale changes are small. See section 5.2 for further discussion.

[6] Note, this threshold value is particular to this set of imagery. It is likely that imagery acquired on a different day, or imagery of a different scene would require a different threshold. The algorithm could be improved by using an automatic thresholding method.

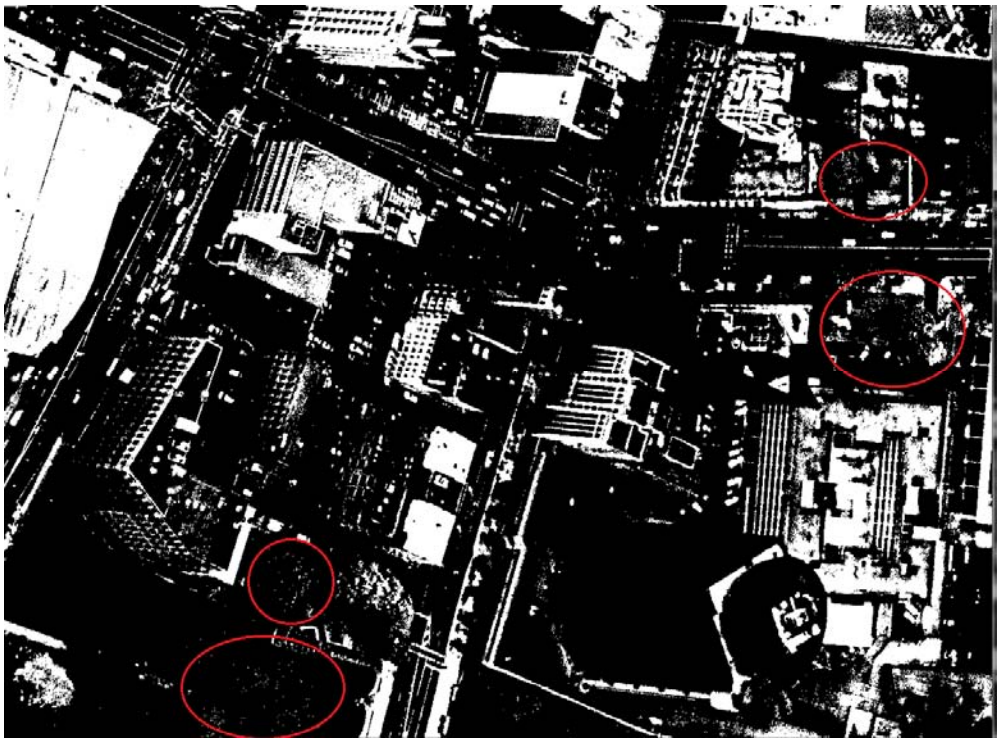

Figure 9: Binary image produced after manually thresholding the temporal-differenced image

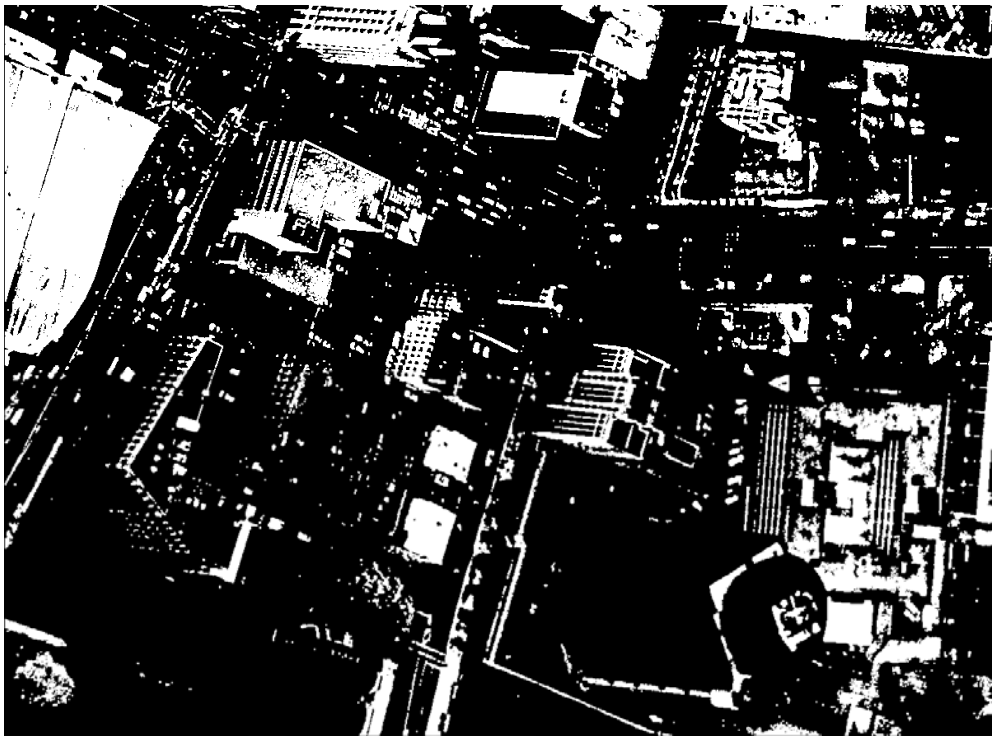

Figure 10: Binary image produced after morphologically opening the original thresholded image

Each white blob in the binary image was then assigned to a region, and each region was assigned a color for display. Very large and very small regions were eliminated, leaving approximately vehicle-sized blobs ranging between 12 and 375 pixels[7]. The reduced region array is shown in Figure 11.

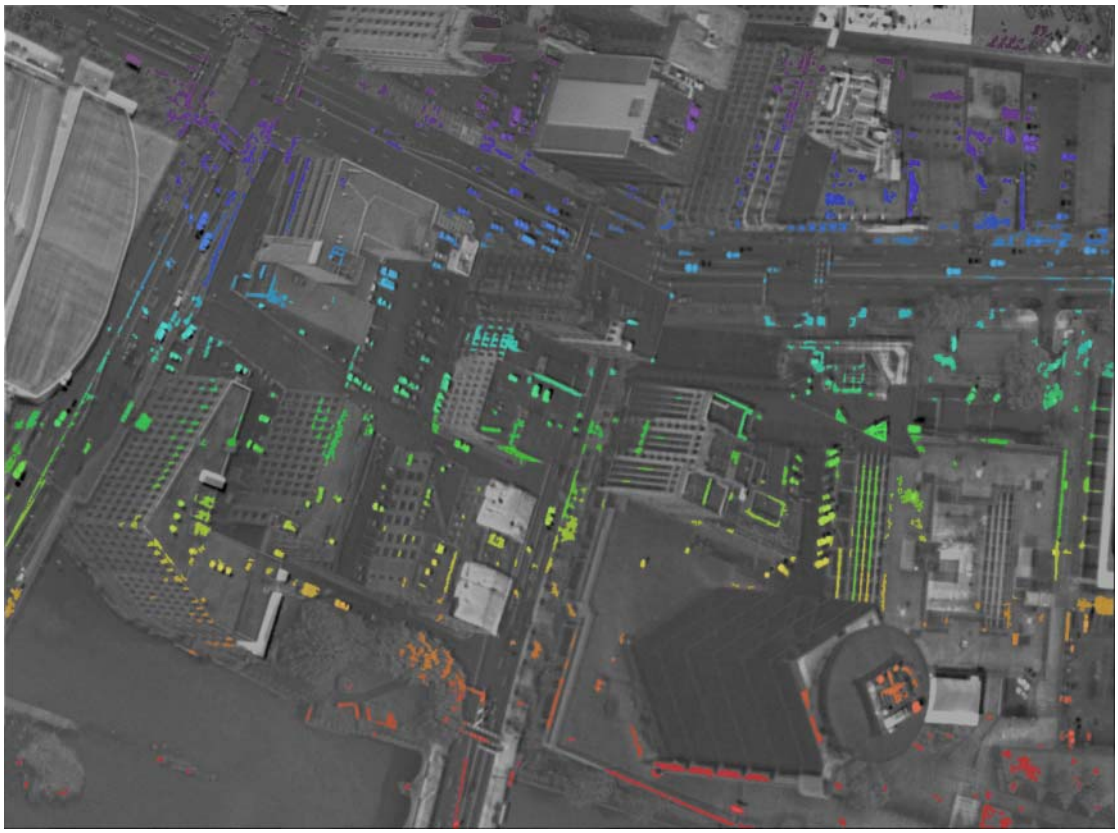

Figure 11: Region array produced by assigning a unique region number to each colored region. These are the regions remaining after large and small regions not meeting a size constraint were eliminated. (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

The shapes of these vehicle-sized regions were then tested in order to eliminate regions created by edges of buildings [13]. In order to do this, a covariance matrix of pixel locations contained in each region was computed. The eigenvectors of the covariance matrix can be used as the x' and y' axes of a new reference frame, rotated such that the x' axis aligns in the direction of maximum spread of the region blob. See Figure 12. The ratio of the first and second eigenvalues of the covariance matrix can then be used as an indication of the length to width ratio of the blob. Eliminating all regions whose covariance eigenvalue ratio is greater than a threshold will eliminate very long regions, such as edges of buildings or roadway markings, while leaving square and rectangular regions that are roughly "vehicle shaped." Once the region passes the shape test, an initial coarse selection rectangle was drawn around the region.

[7] It should be noted that while 12 pixels is too small for a full vehicle size, the manual thresholding method was seen to "split" some vehicles into two smaller segments. Using a small value for minimum vehicle size prevents the algorithm from throwing away real targets that were poorly thresholded. Similarly, 375 pixels is a very conservative estimate for maximum vehicle size (that is, overly large). However, this conservative maximum allowed the algorithm to eliminate grossly large regions (buildings, etc) without throwing away potential items of interest (*i.e.*, large semi-trucks). The purpose of the size constraint is primarily to reduce processing throughput by eliminating regions of no interest.

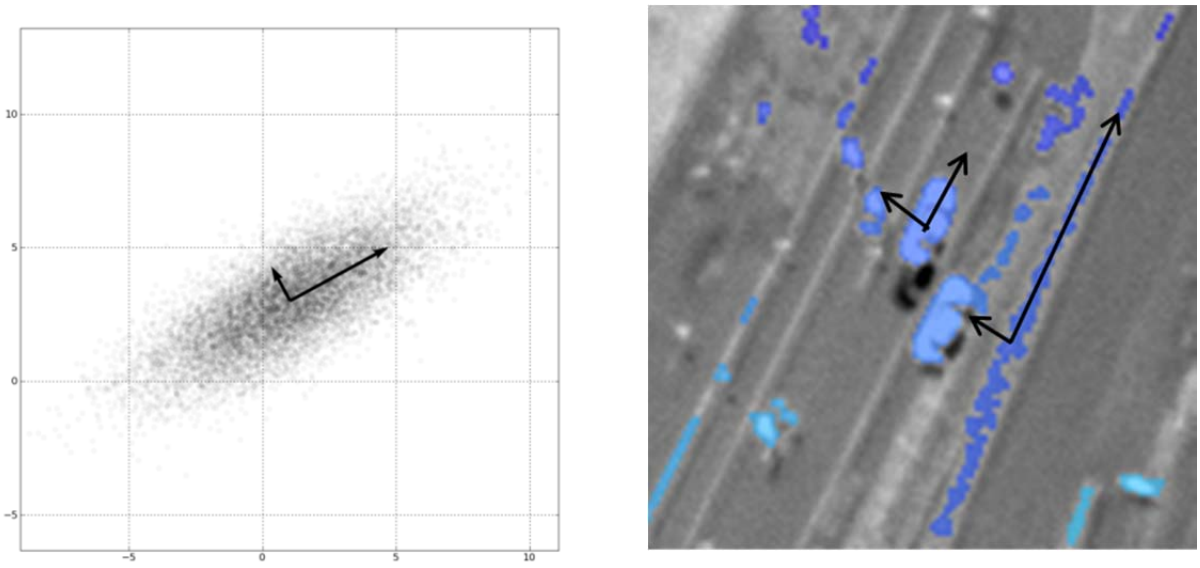

Figure 12: Example multivariate distribution (left) showing covariance in the x and y directions. The directions of the arrows represent the eigenvectors of this distribution's covariance matrix, and the lengths of the arrows correspond to the square roots of the eigenvalues of the covariance matrix. (Image courtesy of Wikimedia Commons, http://commons.wikimedia.org/wiki/Main_Page) The image on the right is an example of a vehicle region that is retained and a road edge region that is eliminated due to the eigenvalue ratio threshold.

Next, a more refined method for eliminating background points from true movers was applied to the KLT list that was optimized for vehicular clustering. Here, instead of examining the histogram of the motion direction and magnitude, a z-score was computed for both the magnitude and direction of the tracked point's motion [13]. The z-score of an observation x can be computed as

$$z = \frac{x-\mu}{\sigma} \tag{12}$$

where μ is the mean of the population containing observation x, and σ is the standard deviation of the population. The z-score of an observation represents the distance between the observation and the population mean in units of standard deviation. So, for example, if an observation has a z-score of magnitude 1, that observation is one standard deviation away from the mean. Because the z-score measures distance from the population mean, the z-score can be used to separate outliers from a population. In our case, points with a high z-score are assumed to be movers, while points with a low z-score are classified as background. This is based on the assumption that the majority of KLT tracked points will fall on background, with relatively few moving vehicles in a given scene.

It was found that moving targets were identified best by first checking the z-score of the point's motion direction for extreme outliers. For any points not passing the direction check, the z-score of the point's magnitude was checked. If the point's motion was an outlier in either magnitude or direction, it was classified as a mover, and added to a coarse motion layer. If it was not an outlier, it was classified as background and not displayed. The results of eliminating background coarse layers are shown in Figure 14. As a reference, the coarse layers calculated without performing this background elimination step are shown in Figure 13.

There are still some points that pass both the size check and the motion consistency check but are not actually moving vehicles. Some of these erroneously assigned motion layers are circled in Figure 14, and are an example of false detections, or false alarms. The high contrast regions shown in the right-most image in Figure 15 passed the eigenvalue criteria of the KLT tracker, allowing points to be detected. The detected KLT points fell within regions that pass the vehicle size and shape check, as can be seen in the left-most image in Figure 15. The overall image motion due to the airborne camera caused these points to have motion different enough from the rest of the background (that is, a high z-

score for both direction and magnitude), so they also passed the moving target check. At this stage of the process it is best to be conservative when eliminating regions as non-movers, at the expense of additional false alarms, because it is undesirable to inadvertently throw away potential tracks of interest. The trade-offs between false alarm rate and missed detections, as well as potential ways of reducing false alarms, are further discussed in Section 5.4.

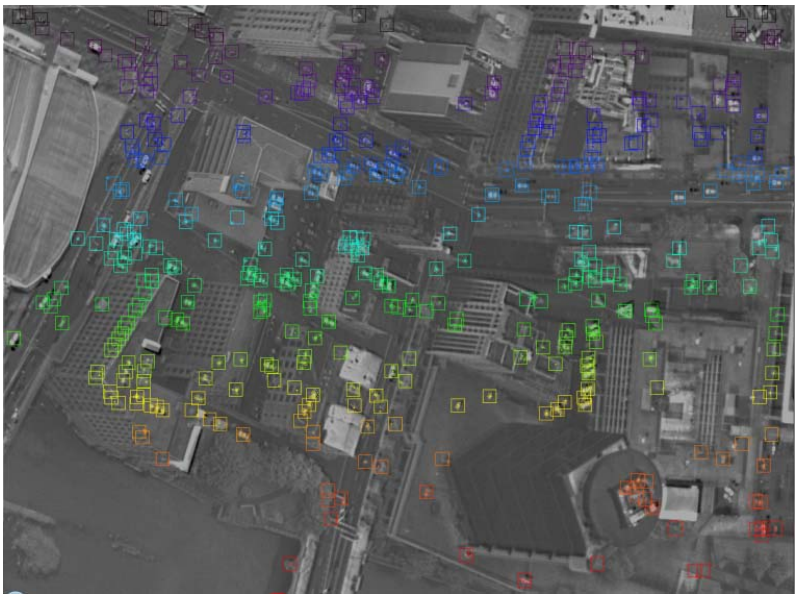

Figure 13: Original set of coarse motion layer and assigned KLT points prior to removing layers determined to have motion consistent with the background. (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

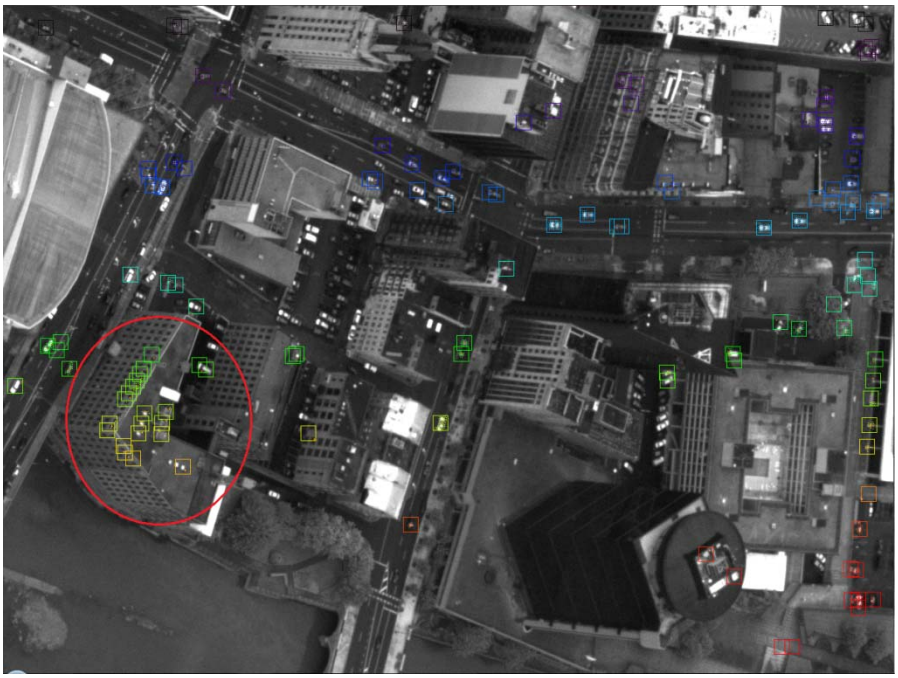

Figure 14: Coarse motion layers and assigned KLT points remaining after elimination of background layers (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

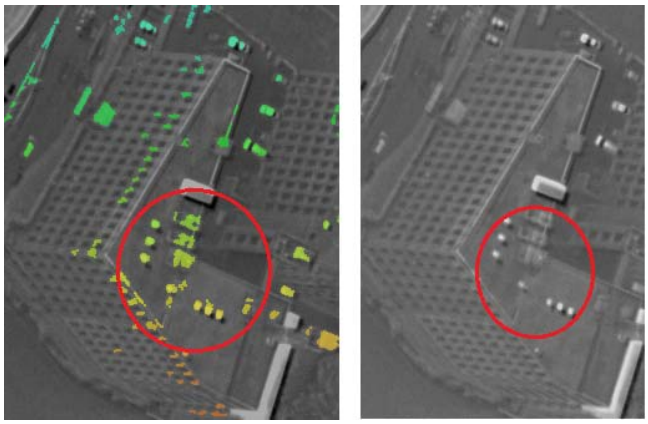

Figure 15: Regioned image (left) showing areas passing the vehicle size and shape checks. TD image (right) showing high-contrast rooftop vents, allowing for KLT point detection.

Once the KLT points have been assigned to a coarse selection rectangle, and the extraneous background layers have been eliminated, the motion of each point within a rectangle is compared to the motion of the KLT point closest to the center of the rectangle using the motion similarity metric [9] given in Equation (11). By comparing all points within the rectangle to the motion of the point closest to the center, points lying within the rectangle but not on the target can be eliminated from the motion layer. This final set of points is shown in Figure 16, and completes the initial motion layer detection phase.

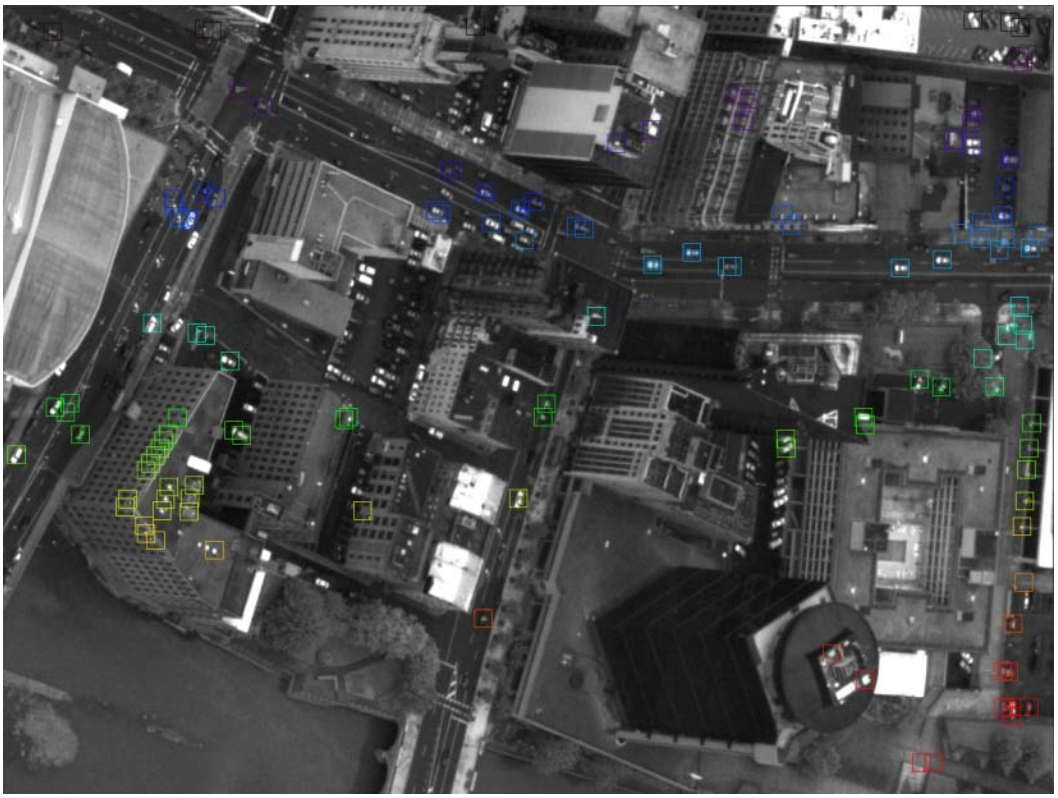

Figure 16: Completed detection of motion layers for frame $i_{tl.}$ (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

4.2 Motion Layer Maintenance

After the initial set of motion layers are detected, the layers must be maintained for several subsequent frames until the next detection/creation cycle. The KLT algorithm attempts to replace any points that have been lost between frames with new points, so it is possible that the layers can both lose and gain points from frame to frame. The motion layer algorithm must be able to add new points to the layer, maintain tracked points within the layer, remove points that have been lost, and eliminate the layer entirely if all points in the layer are lost.

In order to maintain the motion layer in i_{t2}, the first step is to place the coarse selection rectangles from i_{t1} into their proper position in i_{t2}. This was done by approximating the motion between i_{t1} and i_{t2} as the motion between i_{t0} and i_{t1}. This method works best if the vehicle's change in velocity is small. In order to compensate for inaccuracies in positioning the coarse rectangle, the size of the coarse rectangle could be increased if desired. An example of where one might decide to increase the coarse rectangle size is illustrated in Figure 17. In general, the position errors for this implementation were determined to be small enough that growing the rectangle was not required.

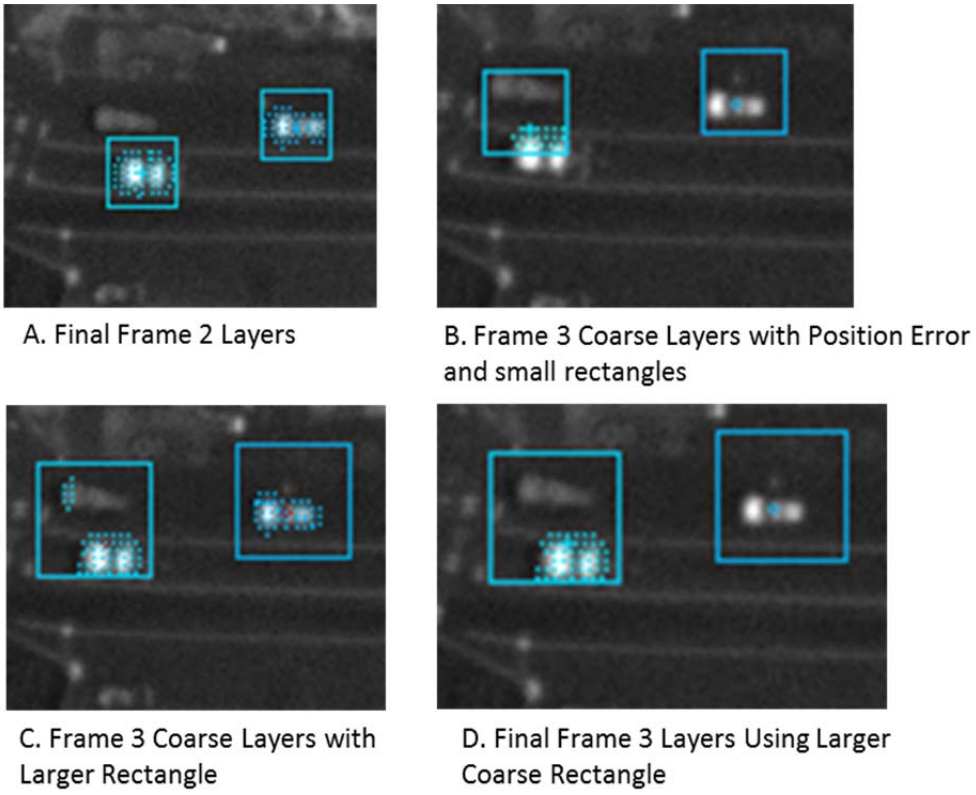

Figure 17: Vehicle subset illustrating inaccuracies in coarse rectangle placement

Once the coarse selection rectangles are repositioned, all of the KLT points tracked into i_{t2} are compared to the coarse rectangle boundaries. If any points are located within the boundaries of the rectangle they are added to the coarse layer. These results are shown in Figure 18.

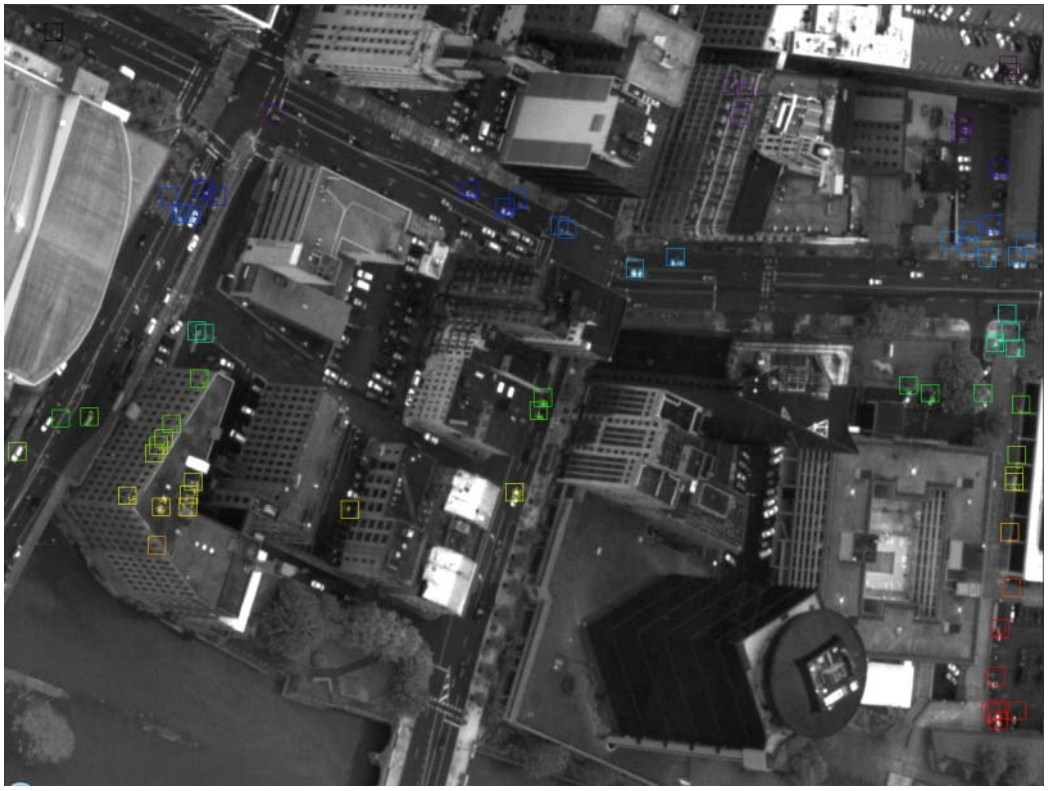

Figure 18: Coarse motion layers maintained into frame $i_{t2.}$ (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

Once the new set of points is available, a new point closest to the center of the rectangle is computed. The motion of each point in the layer is then compared to the motion of the new point closest to the center, using Equation (11). Points that do not share the motion of the point closest to the center are removed from the layer. The final set of motion layers computed for i_{t2} are shown in Figure 19. This phase of motion layer maintenance can then be repeated for several frames before repeating the motion layer detection phase. This speeds processing of the established tracks, albeit at the expense of real-time new target detection.

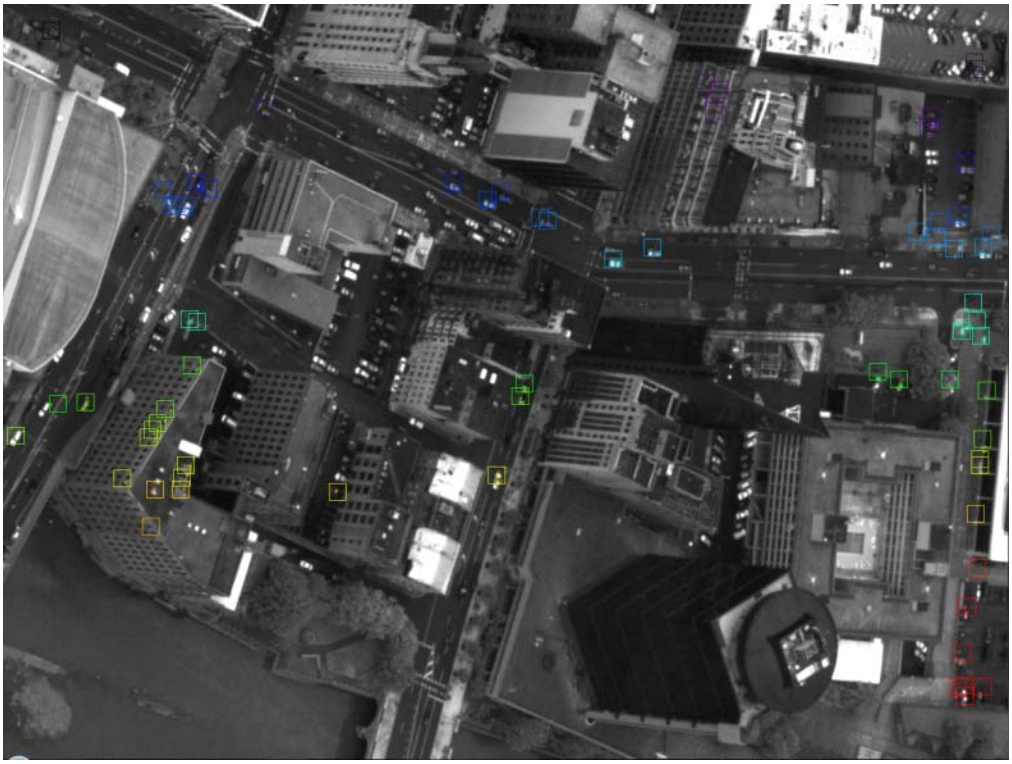

Figure 19: Completed motion layer maintenance for i_{t2}. (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

5. RESULTS

Overall, algorithm performance was per design. The algorithm was designed to detect and track moving vehicles by assigning them to motion layers, and maintaining the motion layer through several frames. The algorithm is expected to be insensitive due to changes in intensity caused by shadow or glare due to the nature of how the KLT algorithm chooses "good features" to track. Adding the KLT tracked features to motion layers allows the algorithm to be robust to partial object occlusions that may occur after initial detection due to the design of the motion layer maintenance phase; lost points can be removed from the layer and new points can be added once detected, provided the layer has been established. One known limitation of this algorithm is that no state estimation is being performed in order to maintain track on fully occluded objects. An object might be re-detected after being fully occluded, but it would be classified as a new target.

An example where the algorithm performed exactly as expected is shown in Figure 21. The upper two images of Figure 21 show the coarse motion layers (left) and final layers after motion consistency is applied (right) for frame i_{t1}. The lower two images show the coarse motion layers (left) and final motion layers (right) for the vehicles maintained into frame i_{t2}. The areas circled in red depict stationary vehicles that were initially assigned to a motion layer, but were correctly rejected in frame i_{t2} as non-movers. This is discussed further in section 5.3. The vehicle identified by area A was split into two separate motion layers due to the thresholding process discussed in 5.3, but the algorithm was able to successfully maintain track on the two layers into frame i_{t2}. While this may not be optimum behavior, it did not appear to negatively affect the final result.

The algorithm did behave poorly in one area where it was expected to behave well. Figure 20 shows two vehicles with good contrast against the surrounding background, and a large number of initial KLT points assigned to the coarse motion layer. When the motion of the KLT points was compared to the motion of the point closest to the center of the rectangle, all of the KLT points with the exception of the point closest to the center were eliminated from the layer. This can occur if the point closest to the center is new, with no motion history, in frame i_{t1}. This behavior could be corrected

by error-checking the point closest to the center of the coarse rectangle. If the point is new, then the algorithm should be designed to select the next closest point that has a motion history.

When the coarse selection rectangles are moved to their new position in frame i_{t2}, the magnitude and direction of the motion were based on the history of motion between frame i_{t0} and i_{t1}. So, if the point is new for frame i_{t1}, there will be no motion history available to use as a motion estimate for moving the rectangle. This should be corrected by using the true motion of the point closest to the center between frame i_{t1} and i_{t2}, since it is known when processing frame i_{t2}, without estimating the motion. If the point closest to the center is lost in frame i_{t2}, then the next closest point to the center of the rectangle with a motion history between frame i_{t1} and i_{t2} should be chosen.

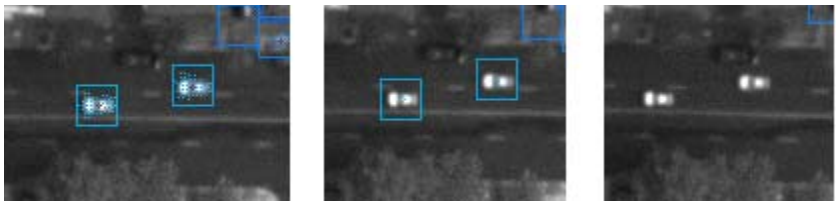

Figure 20: Most KLT points present in the coarse layers detected in i_{t1} (left) are eliminated when motion consistency is applied (center), and are lost completely in frame i_{t2}.

The performance of the combined KLT plus motion layer algorithm was highly dependent on a few major factors. The signal-to-clutter ratio of the targets of interest was the major influence in whether both the KLT algorithm and the temporal differencing to establish the initial motion layers performed well. Effects of the signal-to-clutter ratio of detected vehicles on algorithm performance are discussed in Section 5.1. Another major influence of algorithm performance is input image size. Size of the imagery affects the choice of good features by the KLT algorithm, as well as affecting the image registration and temporal differencing steps of initial motion layer establishment. The effect of size on algorithm performance is discussed in Section 5.2. The spacing of targets (both moving and non-moving) also influences the overall algorithm performance, as discussed in Section 5.3. Section 5.4 concludes the discussion of algorithm performance with a discussion of detection performance versus false alarm rate.

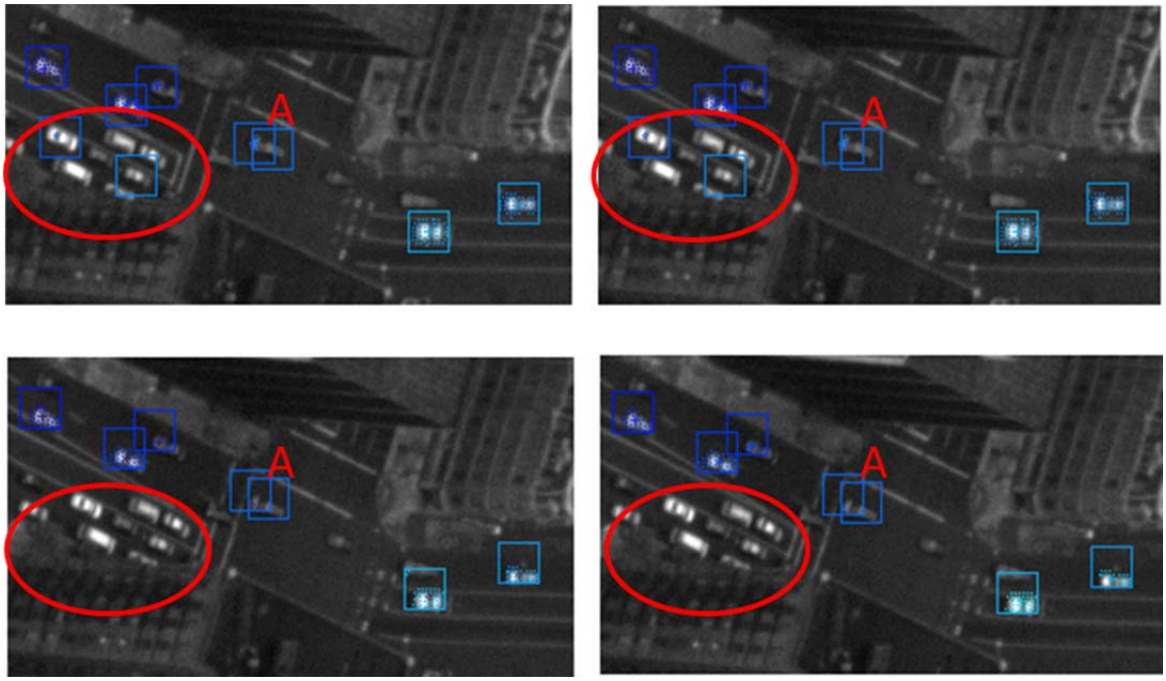

Figure 21: Algorithm Performance: Coarse motion layers identified in i_{t1} (top left) are refined into the final layers for i_{t1} (top right). These are then tracked as coarse layers into frame i_{t2} (bottom left) and refined into final layers (lower right).

5.1 Effects of Signal-to-clutter Ratio on Algorithm Performance

The performance of the motion layer portion of this detection and tracking algorithm is heavily dependent on the initial output of the KLT algorithm. Figure 23 shows a subset of the points output by the KLT algorithm. This subset was part of a 950x1200 pixel image. The yellow circles highlight vehicles that only have a few pixels meeting the criteria of the KLT tracker. Comparing Figure 23 to Figure 22, it can be seen that the highest contrast vehicles shown in Figure 22, were most easily detected by the KLT tracker in Figure 23.

The temporal differencing portion of the motion layer algorithm had a similar difficulty in detecting lower contrast moving vehicles. It can be seen in Figure 24 that the vehicles circled in yellow were not assigned to a region, which means they would not have been assigned a motion layer during initial motion layer detection. Since the coarse selection rectangles are first created from the temporal differenced image, and KLT points are assigned once the coarse regions are created, it is possible that the temporal differencing step could miss a region that the KLT tracker actually detects. In this case, the KLT point would never be assigned to a motion layer even though it is a valid tracked point. This could potentially increase the percentage of missed targets. Therefore, it is very important to establish good thresholding criteria tuned to the particular scene of interest in order to establish the initial coarse selection rectangle regions.

Figure 22: Frame i_{t1} subset showing high-contrast vehicles. (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

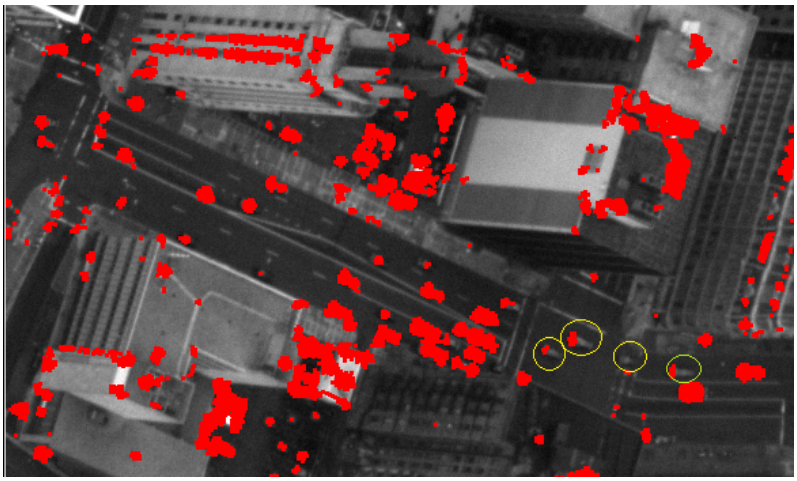

Figure 23: Initial KLT tracked points (red dots), optimized for finding features clustered on moving targets in frame i_{t1}. (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

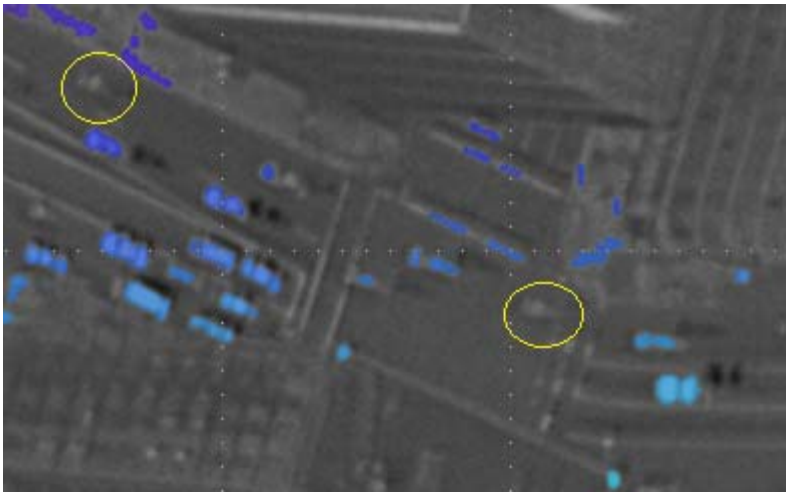

Figure 24: Regioning performed on temporal differenced image, showing where low-contrast targets are not detected. (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

5.2 Effects of Image Size on Algorithm Performance

Since the KLT algorithm is optimized to find the best features in an image, it is natural for the algorithm to choose the high contrast features over low contrast features. Unfortunately, this means that for a large image, such as the full-sized WAAS output shown in Figure 25, the KLT algorithm can select the best features to track, but they are not necessarily the objects of interest.

Figure 26 is a closer view of the area identified by the yellow square in Figure 25. Parked vehicles, as well as windows and edges of buildings can all meet the criteria for a “good track” when running the KLT algorithm. Because the KLT algorithm will select a specified number of the “best” features by design, this causes it to exclude many of the moving vehicles that are really the targets of interest, if they had a lower contrast against the background.

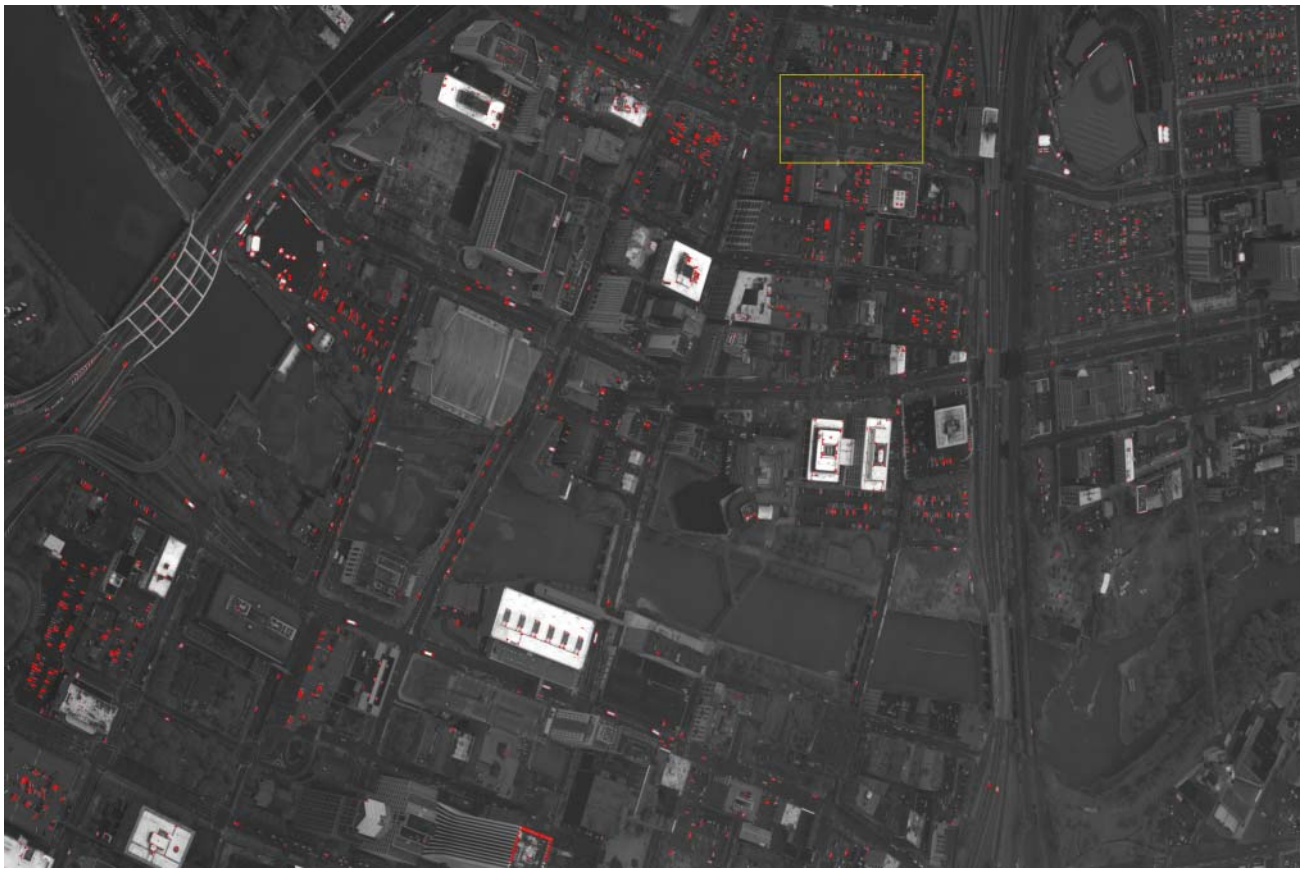

Figure 25: Full size KLT tracker output, optimized for finding points clustered on moving vehicles in frame i_{t1}. Yellow square identifies an area where "good KLT features" are not the targets of interest (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

Figure 26: Zoomed region of full-sized image, highlighting parked vehicles as good features to track. (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

A large image size can also induce perspective changes that cause errors in algorithm performance. The WAAS sensor is oriented to collect imagery approximately 40 degrees off nadir, causing the ground path between the sensor and a point at the top of the sensor FOV to be significantly different than the ground path between the sensor and points at the bottom of the FOV. This induces a perspective change throughout the imagery, because the ground spot size of pixels located further from the sensor will be larger than the ground spot size of pixels that are closer to the sensor. This means that objects far from the sensor will appear smaller in the imagery (*i.e.*, will span fewer pixels,) than identical objects close to the sensor. At the center of the image this change in perspective should be small, but the perspective change will become larger towards the edges of the image, and will grow with the size of the image.

When the perspective change across the image is large, errors are induced in the image registration step. In order to warp the image at i_{t0} to exactly match the image at i_{t1}, the mapping transform would have to account for this change in perspective. If the image were smaller, the change in scale would be less prominent, and the 2nd-order polynomial mapping algorithm used for image registration would more closely approximate the actual transformation between the two images. Improving the image registration would then also improve the temporal differencing step, because the differences between locations of background pixels would be reduced. This would help eliminate some of the "ghosting" that occurs at the edges of buildings, as highlighted in Figure 27.

Effects of image size on the overall algorithm performance can be mitigated by processing the image as several smaller "blocks." The algorithms can be run independently on each image block, with the results combined at the end of a frame.

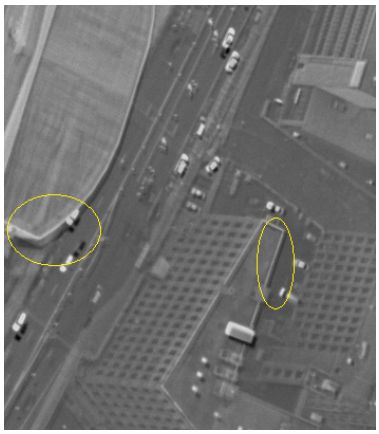

Figure 27: Temporal differenced image highlighting areas where errors in image registration cause misaligned building edges to appear as "ghosts" or shadows. (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

5.3 Effects of Target Spacing on Algorithm Performance

Spacing between objects was a critical factor in the algorithm's ability to detect the initial coarse motion layer regions. If a vehicle's intensity is similar to that of the background, with no contrasting pixels in between, the thresholding process cannot separate the vehicle from the background. This is illustrated in Figure 28, by the partially occluded vehicle circled in red. There is no separation between the vehicle and the building in the thresholded image, so when the regioning step occurs, the entire large, blended region is rejected as being too large. Use of a more advanced morphological operation might help improve this, however it is necessary to balance between eroding an area in order to separate vehicles from background, and closing areas where vehicles might be split in two. This is shown in Figure 29.

Even when the thresholding step produced cleanly separated vehicles, if the vehicles were too closely spaced the coarse selection rectangles overlapped as shown in the lower left image in Figure 30. The lower left image was created prior to eliminating regions that shared the same motion as background. The lower right image correctly shows fewer regions in the lower half of the image because it is believed that those vehicles were stopped at a traffic light, and therefore "moving" as background. There were no areas of the image where vehicles believed to be in motion were clustered such that the algorithm rejected an unacceptable number of KLT points, or eliminated entire motion layers. Therefore it is unknown whether overlapping coarse selection rectangles will induce unacceptable algorithm performance.

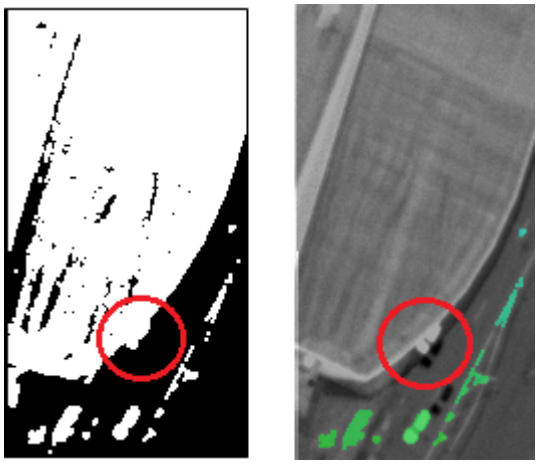

Figure 28: Thresholded image (left) illustrating where a partially occluded vehicle is not assigned to a region (right) due to blending with a large region that does not meet the imposed size constraints.

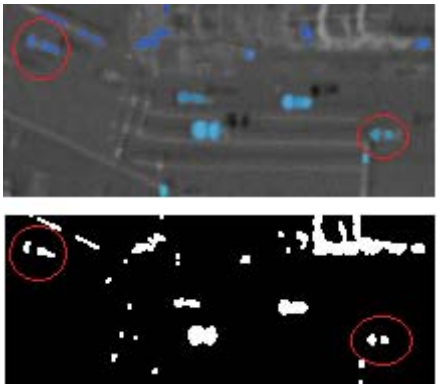

Figure 29: Regioned Image (top) and thresholded image (bottom) showing an example where portions of vehicles are split into separate regions.

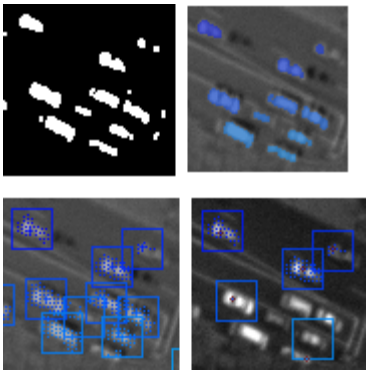

Figure 30: Cleanly thresholded vehicles (top left) producing distinct vehicle regions (top right). The close spacing of the vehicle regions produce overlapping coarse rectangles (lower left). Some of these regions were then correctly rejected by the z-score motion constraint method as moving too much like background (lower right).

5.4 Missed Detections and False Alarms

In scoring algorithm performance, the two most important factors are missed detections and false alarms. The most common causes of missed detections are low signal-to-clutter ratio, and poor thresholding due to closely spaced target and background, which were discussed in section 5.1 (Figure 24) and section 5.3 (Figure 28). Since the goal of the motion layer algorithm was to detect and track *moving* vehicles, a false track would be defined as reporting to the end user the detection and track of objects that share the same motion as the background. While the z-score method described in section 4.1 was effective in separating movers from non-movers, some non-movers were still present in the final set of motion layers as shown in Figure 31. If these sets of motion layers, which include windows, parked vehicles, and rooftop vents, were reported to the end user then the false alarm rate would be high.

These non-movers could be eliminated in several ways. One way is to ensure that over time the motion layer moves like a vehicle. A velocity gate could be established such that the motion layer is only reported after it satisfies the velocity gate for M out of N frames. However, this method could be complicated due to variations in traffic flow that would occur in urban areas; motion would be quite different between vehicles idling at a stoplight and vehicles cruising down an open highway.

The fact that the vehicular motion is not constant, and the vehicles can sometimes be stopped altogether, poses a challenge in balancing detection performance against false alarm rate. There is not much that can be done for missed detections due to low signal-to-clutter ratio (*i.e.*, target contrast); however "missed" detections can be improved by not rashly eliminating potential targets because of their size or motion. It would be best to verify the motion of the layer is the same as the background motion for several frames before eliminating the layer as background in order to prevent an early discard of a stopped vehicle, such as the one stopped at a traffic light (Figure 30). This can, however, cause the target declaration time to be very long, since a motion history would have to be established prior to declaring any targets.

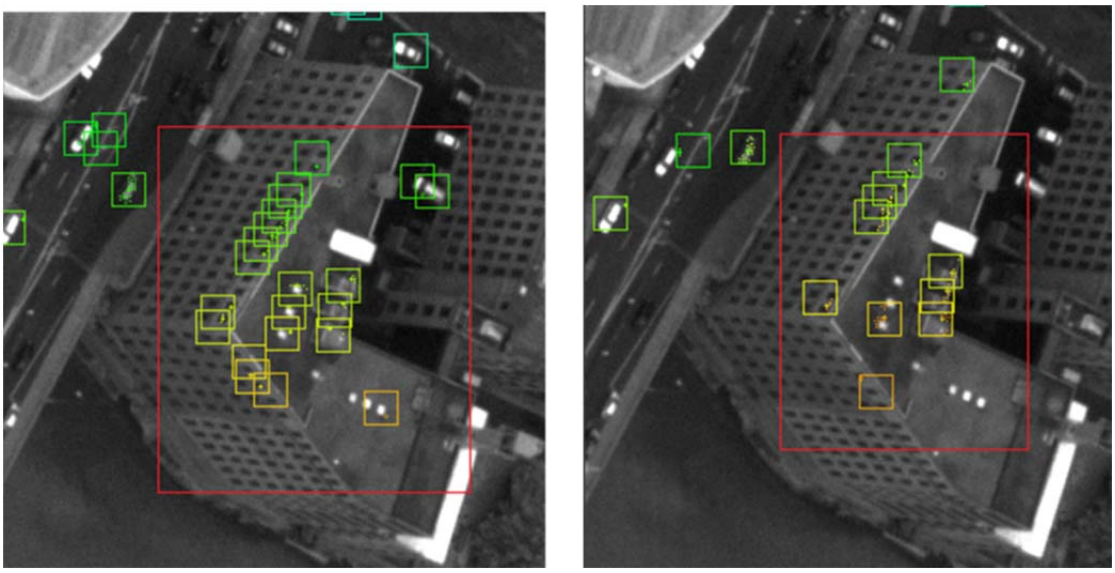

Figure 31: Non-moving background points remaining in final motion layers of frame i_{t1} (left) and i_{t2} (right). (Original image courtesy of ITT Exelis WAAS sensor flight test, August 2010.)

6. CONCLUSIONS

Adding KLT features to motion layers for tracking moving vehicles allows for a robust algorithm that combines the best features of both the KLT tracker and motion layer concepts: the algorithm is robust against changes in illumination due to the nature of the KLT tracker, and is also robust against partial occlusions due to the manner in which the motion layers are maintained. While this particular implementation does not share the benefit of real-time target tracking that the original motion layer implementation boasts, the concept of vehicle tracking using motion layers has been demonstrated using imagery provided by the ITT Exelis WAAS sensor.

As expected, signal-to-clutter ratio of the target against the background was a driving factor in the ability to detect, and subsequently to track, the targets of interest. The large ground coverage area of the WAAS imagery posed some unique issues, but these can likely be mitigated by breaking the image into sections and processing the algorithm on these individual sections.

Tuning the algorithm to the imagery of interest was particularly important in order to balance missed detections against false alarms. Specialized tuning was performed in several areas, including establishing the initial threshold criteria for creation of the coarse rectangle regions, establishing size and shape criteria in order to only accept motion layers that could be vehicles, and establishing the motion constraints used to determine if KLT feature motion matched that of the background, or the pixels closest to the centers of the coarse selection rectangles. While the basic theory of the motion layer tracker can be used to satisfy many different customers, the specific tuning required in order to meet a particular false alarm rate, or target declaration timeline, might cause the same algorithm to perform very differently. Ultimately, it is the customer requirements that would drive the particular tuning required.

6.1 Evaluation of Objectives

The three stated objectives of this project were:

- To apply the techniques investigated in this project to real-life sensor applications encountered in industry.
- To apply the knowledge gained through work experience to an academic project.
- To gain a deeper understanding of image processing techniques and their applications to specific problems.

I have been working as a professional engineer in the aerospace industry for the past nine years. Every year I return to my undergraduate alma mater to talk with the students about working in industry. This year, one of the students asked me “How do you know when you’re finished with a project?” My tongue-in-cheek, but completely truthful answer was that you are finished when you run out of schedule, or you run out of money. At the time, I did not realize that one day I would have to convince myself that my answer would be acceptable in ultimately determining the scope of this project.

Initially, I began this project with the desire to create a “new” tracking algorithm by improving upon existing techniques. I’ve worked with many sensors that use Kalman filters for state estimation of moving targets (*i.e.*, estimating the target’s position and velocity in order to predict its future movements), in order to track targets through a missed detection, or through a full occlusion. So naturally, my first thought was to implement an algorithm using a Kalman filter to further my understanding on the topic. Ironically, this project quickly diverged from that original desire when I realized through literature review that I couldn’t simply start with tracking; I had to start with vehicle *detection*. This increased the initial scope of the project, and by the time I had the detection and tracking piece of the project working satisfactorily there was not time to improve the implementation by adding a state estimation algorithm.

Overall, however, I feel that I did meet the three stated objectives of this project. Using temporal differencing as a method for moving target detection is common in industry. By performing an image registration and attempting to temporally difference two images that didn’t line up perfectly, I can now more fully appreciate the challenges this approach poses for the sensor suppliers. My work experience has enabled me to identify techniques used in industry, such as examining motion history over time as a method to reduce false alarms, which could be useful in improving the overall performance of the motion layer tracker. Work experience has also taught me that customer requirements must ultimately drive performance, and it is likely to be the tuning performed during algorithm development that will generate the correct performance to meet the customer requirements. Finally, this project has exposed me to some new image

processing tricks, such as using eigenvalues/eigenvectors to roughly determine shape, and using statistics such as the z-score to eliminate outliers, that I can keep in my back pocket for use the next time I face a similar problem.

I've also learned a few intangible lessons through the course of this project, the most important of which is that in academics, as opposed to industry, it is okay to let the scope of the project develop as things progress. I spent a lot of time investigating different methods of tuning the algorithm for improved performance. Even though I did not have an opportunity to implement some things that would have been very cool, such as investigating the infrared WAAS imagery, or implementing a Kalman filter, I learned quite a lot about the individual steps used in the creation of this algorithm, and what worked well, and not so well, when applied to the WAAS imagery.

ACKNOWLEDGEMENTS

Thank you to my husband, Brian, who has been with me throughout the course of this degree. I am sure you are looking forward to life after school as much as I am.

A big thank you also goes to Carl Salvaggio. While it sounds like a small thing, your willingness to answer e-mail on the weekends has truly been the difference maker. You've helped make the coding bearable, and I've very much enjoyed working with you.

REFERENCES

[1] Lillesand, T., Keifer, R., Chipman, J., [Remote Sensing and Image Interpretation], John Wiley & Sons, Inc., 62-64 (2008).
[2] Google Earth, http://www.google.com/earth/index.html, 2013.
[3] "National Geospatial-Intelligence Agency", Wikipedia, http://en.wikipedia.org/wiki/National_Geospatial-Intelligence_Agency, 2013.
[4] "Radar Jamming and Deception", Wikipedia, http://en.wikipedia.org/wiki/Radar_jamming, 2013.
[5] "Sniper Pod", Lockheed Martin, http://www.lockheedmartin.com/us/products/Sniper.html, 2013
[6] AN/AAQ -28(V) LITENING Product Sheet, Northrop Grumman Corp., http://www.northropgrumman.com/Capabilities/LITENING/Documents/litening.pdf, 2013.
[7] Gonzalez, R., Woods, R., [Digital Image Processing], Pearson Prentice Hall, 689-787 (2008).
[8] Shi, J., Tomasi, C., "Good Features to Track," IEEE Conference on Computer Vision and Pattern Recognition, 593-600 (1994).
[9] Cao, X., Lan, J., Yan, P., Li, X., "KLT Feature Based Vehicle Detection and Tracking in Airborne Videos," *Image and Graphics (ICIG), 2011 Sixth International Conference on* , vol., no., pp.673-678, 12-15 August,. 2011
[10] Shi, J., Tomasi, C., Good Features to Track. TR 93-1399, Cornell U., 1993.
[11] Birchfield, S., "KLT – an Implementation of the Kinade-Lucas-Tomasi Tracker", http://www.ces.clemson.edu/~stb/klt/
[12] Yu, Q., Medioni, G., "Motion Pattern Interpretation and Detection for Tracking Moving Vehicles in Airborne Video," CVPR pp. 2671-2678 (2009).
[13] Salvaggio, C., personal communication, July 2012 – May 2013.

APPENDIX A – IDL SOURCE CODE

```
;+
; :NAME:
;     READ_GCP_File_subset1
;
; :PURPOSE:
;     Reads a Trackfile (text file) and compute motion vector for each tracked target.
;
; :CATEGORY:
;     Image Processing.
;
; :CALLING SEQUENCE:
;     result = READ_GCP_File_subset1( filename )
;
; :INPUTS:
;     filename:
;        The ENVI header filename to be read.
;
; :KEYWORD PARAMETERS:
;     NONE
;
; :RETURN VALUE:
;     A structure containing the tracked points
;     Record contains the values:
;      [HorizPosIm1, VertPosIm1, ValueIm1, HorizPosIm1a, VertPosIm1a, ValueIM1a,
;       HorizPosIm2, VertPosIm2, ValueIm2, HorizPosIm3, VertPosIm3, ValueIm3]
;
;
; :SIDE EFFECTS:
;     NONE
;
; :REQUIRES:
;   read_gcp_file_subset1
;   A KLT trackfile containing points clustered on moving vehicles
;
;
; :MODIFICATION HISTORY:
;     Written by:       Rachel Kitzmann
;     May, 2013        Original code
```

```
;
;
; :DISCLAIMER:
;    This source code is provided "as is" and without warranties as to performance
;    or merchantability. The author and/or distributors of this source code may
;    have made statements about this source code. Any such statements do not
;    constitute warranties and shall not be relied on by the user in deciding
;    whether to use this source code.
;
;    This source code is provided without any express or implied warranties
;    whatsoever. Because of the diversity of conditions and hardware under which
;    this source code may be used, no warranty of fitness for a particular purpose
;    is offered. The user is advised to test the source code thoroughly before
;    relying on it. The user must assume the entire risk of using the source code.
;-

FUNCTION READ_GCP_File_subset1, FILE, MAXREC=MAXREC

; Check arguments of file
if (n_elements(file) eq 0 ) then $
  message, 'Argument FILE is undefined'
if (n_elements(maxrec) eq 0) then $
  maxrec = 100000L

; Open Input file
openr, lun, file, /get_lun

; define record structure and create array

fmt = '(12(1x, e15.12))'

record = {HorizPosIm1:0.0E+0, VertPosIm1:0.0E+0, ValueIm1:0.0E+0, $
   HorizPosIm1a:0.0E+0, VertPosIm1a:0.0E+0, ValueIm1a:0.0E+0, $
   HorizPosIm2:0.0E+0, VertPosIm2:0.0E+0, ValueIm2:0.0, $
   HorizPosIm3:0.0E+0, VertPosIm3:0.0E+0, ValueIm3:0.0};, $
   ;HorizPosIm4:0.0E+0, VertPosIm4:0.0E+0, ValueIm4:0.0}

data = replicate(record, maxrec)

; Read records until end of file reached
nrecords = 0L
recnum = 1L
```

```
while(eof(lun) ne 1) do begin
  ; read this record (jumps to bad_rec: on error)

  on_ioerror, bad_rec
  error = 1
  readf, lun, record, format = fmt
  error = 0

  ; store data for this record
  data[nrecords] = record
  nrecords = nrecords + 1L

  ; check if max record count exceeded
  if (nrecords eq maxrec) then begin
  free_lun, lun
  message, 'Maximum Record Reached: increase MAXREC'
  endif

  ; check for bad input record
  bad_rec:
  if(error eq 1) then $
    print, 'Bad Data at record ', recnum
    recnum = recnum + 1

endwhile

; close input file
free_lun, lun

; Trim Data array and return it to caller
data = data[0 : nrecords -1]
return, data

end
;-------------------------------------------------------------
;+
; :NAME:
;     MOTION_VECTOR
;
; :PURPOSE:
;     Reads a Trackfile (text file) and compute motion vector for each tracked target.
;
```

```
; :CATEGORY:
;     Image Processing.
;
; :CALLING SEQUENCE:
;     MOTION_VECTOR
;
; :INPUTS:
;     Feature_Clusters_14April.txt
;
; :KEYWORD PARAMETERS:
;     NONE
;
; :RETURN VALUE:
;     A text file containing the tracked points and their motion
;     File contains the columns:
;      [HorizPosIm1, VertPosIm1, ValueIm1, HorizPosIm1a, VertPosIm1a, ValueIM1a, MotionDirection1a,
;       MotionMagnitude1a, HorizPosIm2, VertPosIm2, ValueIm2, MotionDirection2, MotionMagnitude2
;       HorizPosIm3, VertPosIm3, ValueIm3, MotionDirection3, MotionMagnitude3
;       HorizPosIm4, VertPosIm4, ValueIm4, MotionDirection4, MotionMagnitude4]
;
; :SIDE EFFECTS:
;     NONE
;
; :REQUIRES:
;   read_gcp_file_subset1
;   A KLT trackfile containing points clustered on moving vehicles
;
; :MODIFICATION HISTORY:
;     Written by:       Rachel Kitzmann
;     May, 2013        Original code
;
;
; :DISCLAIMER:
;    This source code is provided "as is" and without warranties as to performance
;    or merchantability. The author and/or distributors of this source code may
;    have made statements about this source code. Any such statements do not
;    constitute warranties and shall not be relied on by the user in deciding
;    whether to use this source code.
;
;    This source code is provided without any express or implied warranties
;    whatsoever. Because of the diversity of conditions and hardware under which
;    this source code may be used, no warranty of fitness for a particular purpose
```

```
;    is offered. The user is advised to test the source code thoroughly before
;    relying on it. The user must assume the entire risk of using the source code.
;-
PRO motion_vector

;Read in trackfile that contains points clustered on vehicles
trackfile = 'Feature_Clusters_14April.txt'
trackresult = read_gcp_file_subset1(trackfile)

print, trackresult[0].HorizPosIm2, format = '(e0.0E+0)'
help, trackresult, /structure
help, trackresult[1], /structure


; Compute total number of features that were successfully tracked from frame 1 to frame 2
; (Value = 0).
; Restructure the tracked features into new array

counter1 = 0.0
counter2 = 0.0
counter3 = 0.0
counter4 = 0.0
counter5 = 0.0
maxResultSize = 15000 ; This value needs to be at least as big as number of successfully tracked
features

for i = 0, maxResultSize-1 DO BEGIN
  if trackresult[i].ValueIm1a eq 0.0 then begin
   counter1 = counter1 + 1
  endif
endfor
print, ' counter1 = ', counter1

for i = 0, maxResultSize-1 DO BEGIN
  if trackresult[i].ValueIm2 eq 0.0 then begin
   counter2 = counter2 + 1
  endif
endfor
print, ' counter2 = ', counter2

for i = 0, maxResultSize-1 DO BEGIN
  if trackresult[i].ValueIm3 eq 0.0 then begin
```

```
   counter3 = counter3 + 1
  endif
endfor
print, ' counter3 = ', counter3

;for i = 0, maxResultSize-1 DO BEGIN
;  if trackresult[i].ValueIm4 eq 0.0 then begin
;   counter4 = counter4 + 1
;  endif
;endfor
;print, ' counter4 = ', counter4


STOP

motionDirection1a = fltarr(maxResultSize)
motionMagnitude1a = fltarr(maxResultSize)
motionDirection2 = fltarr(maxResultSize)
motionMagnitude2 = fltarr(maxResultSize)
motionDirection3 = fltarr(maxResultSize)
motionMagnitude3 = fltarr(maxResultSize)
;motionDirection4 = fltarr(maxResultSize)
;motionMagnitude4 = fltarr(maxResultSize)

; Compute motion vector for each tracked feature
for i = 0, maxResultSize-1 DO BEGIN
  IF trackresult[i].ValueIm1a eq 0.0 then begin
  motionDirection1a[i] = 180/!PI*atan((trackresult[i].VertPosIm1a - trackresult[i].VertPosIm1),$
                                (trackresult[i].HorizPosIm1a - trackresult[i].HorizPosIm1))

  motionMagnitude1a[i] = sqrt((trackresult[i].HorizPosIm1a - trackresult[i].HorizPosIm1)^2 + $
                  (trackresult[i].VertPosIm1a - trackresult[i].VertPosIm1)^2 )
  ENDIF
ENDFOR

for i = 0, maxResultSize-1 DO BEGIN
  IF trackresult[i].ValueIm2 eq 0.0 then begin
  motionDirection2[i] = 180/!PI*atan((trackresult[i].VertPosIm2 - trackresult[i].VertPosIm1a),$
                                (trackresult[i].HorizPosIm2 - trackresult[i].HorizPosIm1a))

  motionMagnitude2[i] = sqrt((trackresult[i].HorizPosIm2 - trackresult[i].HorizPosIm1a)^2 + $
                  (trackresult[i].VertPosIm2 - trackresult[i].VertPosIm1a)^2 )
```

```
  ENDIF
ENDFOR

for i = 0, maxResultSize-1 DO BEGIN
  IF trackresult[i].ValueIm3 eq 0.0 then begin
  motionDirection3[i] = 180/!PI*atan((trackresult[i].VertPosIm3 - trackresult[i].VertPosIm2),$
                                (trackresult[i].HorizPosIm3 - trackresult[i].HorizPosIm2))

  motionMagnitude3[i] = sqrt((trackresult[i].HorizPosIm3 - trackresult[i].HorizPosIm2)^2 + $
                  (trackresult[i].VertPosIm3 - trackresult[i].VertPosIm2)^2 )
  ENDIF
ENDFOR

;for i = 0, maxResultSize-1 DO BEGIN
;  IF trackresult[i].ValueIm4 eq 0.0 then begin
;  motionDirection4[i] = 180/!PI*atan((trackresult[i].VertPosIm4 - trackresult[i].VertPosIm3),$
;                                (trackresult[i].HorizPosIm4 - trackresult[i].HorizPosIm3))

;  motionMagnitude4[i] = sqrt((trackresult[i].HorizPosIm4 - trackresult[i].HorizPosIm3)^2 + $
;                  (trackresult[i].VertPosIm4 - trackresult[i].VertPosIm3)^2 )
;  ENDIF
;ENDFOR

print, 'motion Direction1a 0 = ', motionDirection1a[0]
print, 'motionMagnitude1a 0 = ', motionMagnitude1a[0]
print, 'motion Direction2 0 = ', motionDirection2[0]
print, 'motionMagnitude2 0 = ', motionMagnitude2[0]
print, 'motion Direction3 0 = ', motionDirection3[0]
print, 'motionMagnitude3 0 = ', motionMagnitude3[0]
;print, 'motion Direction4 0 = ', motionDirection4[0]
;print, 'motionMagnitude4 0 = ', motionMagnitude4[0]

; Create one big array with each pixel location, value from KLT tracker, and motion vector
; Array is of format [HorizPosIm1, VertPosIm1, ValueIm1, HorizPosIm1a, VertPosIm1a, ValueIM1a,
MotionDirection1a,
; MotionMagnitude1a, HorizPosIm2, VertPosIm2, ValueIm2, MotionDirection2, MotionMagnitude2
; HorizPosIm3, VertPosIm3, ValueIm3, MotionDirection3, MotionMagnitude3
; HorizPosIm4, VertPosIm4, ValueIm4, MotionDirection4, MotionMagnitude4]  ;               ]

TrackedFeatures = fltarr(18, maxResultSize)

for i = 0, maxResultSize-1 DO BEGIN
```

```
  TrackedFeatures[0,i] = trackresult[i].HorizPosIm1
  TrackedFeatures[1,i] = trackresult[i].VertPosIm1
  TrackedFeatures[2,i] = trackresult[i].ValueIm1
  TrackedFeatures[3,i] = trackresult[i].HorizPosIm1a
  TrackedFeatures[4,i] = trackresult[i].VertPosIm1a
  TrackedFeatures[5,i] = trackresult[i].ValueIm1a
  TrackedFeatures[6,i] = motionDirection1a[i]
  TrackedFeatures[7,i] = motionMagnitude1a[i]
  TrackedFeatures[8,i] = trackresult[i].HorizPosIm2
  TrackedFeatures[9,i] = trackresult[i].VertPosIm2
  TrackedFeatures[10,i] = trackresult[i].ValueIm2
  TrackedFeatures[11,i] = motionDirection2[i]
  TrackedFeatures[12,i] = motionMagnitude2[i]
  TrackedFeatures[13,i] = trackresult[i].HorizPosIm3
  TrackedFeatures[14,i] = trackresult[i].VertPosIm3
  TrackedFeatures[15,i] = trackresult[i].ValueIm3
  TrackedFeatures[16,i] = motionDirection3[i]
  TrackedFeatures[17,i] = motionMagnitude3[i]
;  TrackedFeatures[18,i] = trackresult[i].HorizPosIm4
;  TrackedFeatures[19,i] = trackresult[i].VertPosIm4
;  TrackedFeatures[20,i] = trackresult[i].ValueIm4
;  TrackedFeatures[21,i] = motionDirection4[i]
;  TrackedFeatures[22,i] = motionMagnitude4[i]

endfor

help, TrackedFeatures

openw, lun, 'TrackedFeatures_clusters.txt', /get_lun
PRINTF, lun, FORMAT='(18(2x, F12.6) )', TrackedFeatures
free_lun, lun

END


------------------------------------------------------------------------
;+
; :NAME:
;     MOTIONLAYERS
;
; :PURPOSE:
;     This program computes a set of Motion Layers allowing the user to track
;     moving vehicles across 3 image frames. This code works as described in
```

```
;     the paper "An Implementation of Vehicle Tracking Using Motion Layers"
;     written by Rachel Kitzmann, dated 04 May 2013.
;
; :CATEGORY:
;     Image Processing.
;
; :CALLING SEQUENCE:
;     MOTIONLAYERS
;
; :INPUTS:
;     NONE
;
; :KEYWORD PARAMETERS:
;     NONE
;
; :RETURN VALUE:
;     NONE
;
; :SIDE EFFECTS:
;     NONE
;
; :REQUIRES:
;     read_envi_image
;     morph_open
;     cgDisplay
;     cgImage
;     ddread
;     replicate_vector
;     is_equal
;
;     A 960x1204 pixel image warp at time t0 (warped and registered to an image at time t1)
;     A 950x1200 pixel image at time t1
;     A 950x1200 pixel image at time t2
;     A text file containing the KLT tracked points. The file should contain the columns:
;
;     [HorizPosIm1  VertPosIm1  ValueIm1  HorizPosIM1a  VertPosIm1a
;      ValueIm1a    MtnDir1a    MtnMag1a  HorizPOsIm2   VertPOsIm2
;      ValueIm2     MtnDirIm2   MtnMagIm2 HorizPosIm3   VertPOsIm3
;      ValueIm3     MtnDirIm3   MtnMagIm3 HorizPosIm4   VertPOsIm4  ValueIm4  MtnDirIm4  MtnMagIm4]
;
;      where the positions, values, and motion in frame 1a are duplicates of the position,
;      values, and motion in frame 1. Frame 1 and 1a both represent the image at time t0.
```

```
;      Frame 2 represents the image at time t1.
;      Frame 3 represents the image at time t2.
;
;
;
; :MODIFICATION HISTORY:
;     Written by:       Rachel Kitzmann
;     May, 2013        Original code
;
;
; :DISCLAIMER:
;    This source code is provided "as is" and without warranties as to performance
;    or merchantability. The author and/or distributors of this source code may
;    have made statements about this source code. Any such statements do not
;    constitute warranties and shall not be relied on by the user in deciding
;    whether to use this source code.
;
;    This source code is provided without any express or implied warranties
;    whatsoever. Because of the diversity of conditions and hardware under which
;    this source code may be used, no warranty of fitness for a particular purpose
;    is offered. The user is advised to test the source code thoroughly before
;    relying on it. The user must assume the entire risk of using the source code.
;-
;
;
;
;This program computes the motion layers between images.


PRO MotionLayers

;Read in the warped image and the original image frames.
warpname = FILE_SEARCH('warp*.img')
filename = FILE_SEARCH('im*.img')

print, 'warpname = ', warpname
print, 'filename = ', filename[0]

xsize = 1200  ;  4872 is full image size. Subset is 1200
ysize = 950 ;    3248 is full image size. Subset is 950

xWarpSize = 1204   ; 1204 is subset
```

```
yWarpSize = 960    ; 960 is subset

megawarp = bytarr(xWarpSize, yWarpSize)
warpfile = read_envi_image(warpname, HEADER = warpheader)

; Reduce the size of the warped file to 1200x950
warp = bytarr(xsize, ysize)
for i = 0, xsize-1 do begin
for j = 0, ysize-1 do begin
  warp[i,j] = warpfile[i,j]
end
end

; Create one "megafile" that contains all X non-warped images.

num_images = 3

megafile = bytarr(xsize, ysize, num_images)

for i=0, num_images-1 do begin

  file = read_envi_image(filename[i], HEADER = fileheader)

  megafile[*,*,i] = file

end

;Subtract the two frames. Can shift the warp if needed.
frame_difference = bytarr(xsize, ysize)
frame_difference =  DOUBLE(megafile[*,*,0])- SHIFT(DOUBLE(warp), 0, 0) + 128

; pad edges of shifted array with zeroes (if needed) to get rid of junk around edges
;frame_difference[0:1, *, *] = 0
;frame_difference[*, 936:949, *] = 0

WINDOW, /FREE, XSIZE = 1200, YSIZE = 950, $
TITLE = 'frame difference'
tv, bytscl(frame_difference), /order

minval = min(frame_difference, max = maxval)
;print, 'min and max values of array: ', minval, maxval
;help, frame_difference
```

```
;*****make a copy of the frame difference array, just in case **********
temporal_diff =  bytarr(xsize, ysize, num_images)
temporal_diff = frame_difference
;*********************************************************************

;Plot the histogram of the differenced image to determine best thresholding approach
;WINDOW, /FREE
;hist_plot, frame_difference, MIN = 0, MAX = maxval, BINSIZE = 1, xtitle = 'Temporal Differencing
Intensity', ytitle = 'Frequency'


;Reduce noise in the differenced image before thresholding. Apply arithmetic mean filter
;**********this ultimately was not used because it smeared small targets of interest. ********
kernel = replicate(0.111, 3, 3)
smoothed = CONVOL(frame_difference, kernel, /EDGE_TRUNCATE)

;WINDOW, 5, XSIZE = 1200, YSIZE = 950, TITLE = 'noise-reduced image'
;tv, bytscl(smoothed), /order

;frame_difference1 = bytscl(frame_difference)
;WINDOW, /FREE
;hist_plot, frame_difference1, BINSIZE = 1, xtitle = 'DC (scaled)', ytitle = 'Frequency'

;Play with this thresholding to show the importance of establishing a good threshold.
frame_difference[where(frame_difference LT 0, /NULL)] = 0
frame_difference[where(frame_difference gt 255, /NULL)] = 255
frame_difference[where(frame_difference LE 150, /NULL)] = 0
frame_difference[where(frame_difference GT 150, /NULL)] = 255
;frame_difference[where(frame_difference GE 141, /NULL)] = 255

;WINDOW, /FREE
;hist_plot, frame_difference, BINSIZE = 1, xtitle = 'DC (scaled)', ytitle = 'Frequency'


WINDOW, /FREE, XSIZE = 1200, YSIZE = 950, TITLE = 'Thresholded Image - Manual'
tv, frame_difference, /order

thresholdedImage = frame_difference

;perform morphological opening operation on original differenced image
open = morph_open(thresholdedImage, REPLICATE(1, 2, 2))
```

```
WINDOW, /FREE, XSIZE = 1200, YSIZE = 950, TITLE = 'Manual Thresholded Image - opened'
tvscl, open, /order

;Use label_region to find individual regions in the differenced image
; Regioning is based on unfiltered, opened, manual thresholded image.

generalregion_array = intarr(xsize, ysize, num_images)
generalregion_array[*,*,0] = LABEL_REGION(open, /ALL_NEIGHBORS, /ULONG)

region_array = generalregion_array[*,*,0]

; get population of each blob:

  h = HISTOGRAM(region_array, REVERSE_INDICES = r)
 ; help, h
  ;print, 'Number of regions = ', N_ELEMENTS(h)-1
  ;print, 'population of region 0 = ', h[0]

;Rescale original image difference for viewing purposes

rescaled_difference = bytarr(xsize, ysize);, num_images)
rescaled_difference = DOUBLE(megafile[*,*,0]) - SHIFT(DOUBLE(warp), 0, 0) + 128
rescaled_difference[where(temporal_diff LT 0, /NULL)] = 0
rescaled_difference[where(temporal_diff gt 255, /NULL)] = 255
;rescaled_difference[where(temporal_diff LE 128, /NULL)] = 0

;cg routines are taken from Coyote's graphics library, http://www.idlcoyote.com/documents/programs.php
cgDisplay, XSIZE = 1200, YSIZE=950, /FREE, Title = 'Original region array'
cgImage, ROTATE(region_array, 7), CTIndex = 40
cgImage, ROTATE(thresholdedImage, 7), CTIndex = 0, Transparent = 70, Missing_Value = 0

;Remove regions less than 12 pixels and greater than 375 pixels, since these are not vehicles.
;Find the max and min values of region array
min_region = min(region_array)
max_region = max(region_array)
;print, 'min region array = ', min_region
;print, 'max region array = ', max_region
regionLT12 = 0
regionGT375 = 0

FOR i = 1, max_region DO BEGIN
```

```
  temp = WHERE(region_array EQ i, count)
  IF count LT 12 THEN BEGIN
    region_array[WHERE(region_array EQ i)] = 0
    regionLT12 = regionLT12+1
  ENDIF
  IF count GT 375 THEN BEGIN
    region_array[WHERE(region_array EQ i)] = 0
    regionGT375 = regionGT375+1
  ENDIF
END

;print, 'number of regions LT 12 = ', regionLT12
;print, 'number of regions GT375 = ', regionGT375


cgDisplay, XSIZE = 1200, YSIZE=950, /FREE, Title = 'Reduced region array'
cgImage, ROTATE(rescaled_difference, 7 ), CTIndex = 0
cgImage, ROTATE(region_array, 7), CTIndex = 40, Transparent = 50, Missing_Value = 0


file = 'TrackedFeatures_Clusters.txt'

result = ddread(file, /formatted)
; Reformat result into an integer array for locations of pixels

intresult = fix(round(result[0:17,*]))
;help, intresult

; Structure of intresult is:
;[HorizPosIm1  VertPosIm1  ValueIm1  HorizPosIM1a  VertPosIm1a
; ValueIm1a    MtnDir1a    MtnMag1a  HorizPOsIm2   VertPOsIm2
; ValueIm2     MtnDirIm2   MtnMagIm2 HorizPosIm3   VertPOsIm3
; ValueIm3     MtnDirIm3   MtnMagIm3 HorizPosIm4   VertPOsIm4  ValueIm4  MtnDirIm4  MtnMagIm4]

;Compute mean and standard deviation of motion magnitude, and motion direction
;in frame 1 in order to eliminate background from real movers

sz_result = size(result, /DIMENSIONS)

mean_magnitude = mean(result[12,*])
mean_direction = mean(result[11,*])
```

```
print, 'mean mag = ', mean_magnitude
print, 'mean dir = ', mean_direction

stddev_magnitude = stddev(result[12,*])
stddev_direction = stddev(result[11,*])

print, 'stddev_mag = ', stddev_magnitude
print, 'stddev_dir = ', stddev_direction

z_magnitude = fltarr(sz_result[1])
z_direction = fltarr(sz_result[1])

;compute z-score for each pixel in list
for i = 0, sz_result[1] - 1 DO BEGIN
  z_magnitude[i] = (result[12,i] - mean_magnitude)/stddev_magnitude
  z_direction[i] = (result[11,i] - mean_direction)/stddev_direction
END

;print, 'z_magnitude = ', z_magnitude , '     z_direction = ', z_direction

;Compute an overlay of points that were tracked into frame 2
MovingOverlay = bytarr(1200, 950)

;This makes each mover bigger than 1 pixel for ease of display
FOR x = 0, (N_ELEMENTS(intresult)/24)-1 DO BEGIN
   MovingOverlay[intresult[8,x], intresult[9,x]] = 128
    if (intresult[8,x]-1) GE 0 THEN BEGIN
      MovingOverlay[(intresult[8,x])-1 , intresult[9,x]] = 128
    ENDIF
    if intresult[8,x]+1 LE xsize-1 THEN BEGIN
      MovingOverlay[(intresult[8,x])+1 ,intresult[9,x]] = 128
    ENDIF
    if intresult[9,x] - 1 GE 0 THEN BEGIN
      MovingOverlay[intresult[8,x], (intresult[9,x])-1] = 128
    ENDIF
    if intresult[9,x] + 1 LE ysize-1 THEN BEGIN
      MovingOverlay[intresult[8,x], (intresult[9,x])+1] = 128
    ENDIF

END

;Now that we have regions, We want to isolate the vehicles into coarse selection rectangles.
```

```
;Choose squares the size of 22x22 pixels. region_array is set to an integer value representing each
; new region.

trackOverlay2 = bytarr(xsize, ysize)

; Set max length to width vehicle ratio
vrat = 10.0

; MaxResult is maximum number of tracked features read from the file
maxResult = 15000
layerArray = fltarr(xsize, ysize)
rectArray = fltarr(xsize, ysize)
layerArray2 = fltarr(xsize, ysize)
reg = 0
;help, reg
;
;The reason I did this as a list was because at the time I didn't know how to
;add values to the end of an array of unknown size. But I figured out how to
;do that with a list, and then change the list to an array.
Frame2list_i = list([0])
Frame2list_x = list([0])
Frame2list_y = list([0])
Frame2list_value = list([0.0])
Frame2list_direction = list([0.0])
Frame2list_magnitude = list([0.0])

Frame2list_Fr3x = list([0])
Frame2list_Fr3y = list([0])
Frame2list_Fr3value = list([0.0])
Frame2list_Fr3direction = list([0.0])
Frame2list_Fr3magnitude = list([0.0])

Frame2list_Fr4x = list([0])
Frame2list_Fr4y = list([0])
Frame2list_Fr4value = list([0.0])
Frame2list_Fr4direction = list([0.0])
Frame2list_Fr4magnitude = list([0.0])


;print, 'new list = ', Frame2list

FOR i = 1, max_region DO BEGIN
```

```
 ;FOR i = 179, 180 DO BEGIN
  temp = WHERE(region_array EQ i, count)
  IF count NE 0 THEN BEGIN
    pointcount = 0
    ;turns 1-D indices into 2-D indices
    indices = Array_Indices(region_array, temp)
    ;compute covariance and eigenvals of covariance matrix to eliminate regions based on shape
    covar = correlate(indices, /covariance)
    eigenvals = eigenql(covar)
    erat = eigenvals[0]/eigenvals[1]

    ;compute mean x and mean y (center) coordinates of the blob
    meanx = MEAN(indices[0, *])
    meany = MEAN(indices[1, *])

    ;print, 'indices 1 = ', indices
    ;print, 'meanx1 = ', meanx
    ;print, 'meany1 = ', meany

        ;check that edges of rectangle won't run off of the image.
        ;if shape of the blob (determined by eigenvalue ratio) is acceptable, then draw a box around the
blob.
        ;Also, turn value of region_array to zero if ratio check doesn't pass
        ;erat = 1.0 implies square vehicle shape.
        if (meanx-10) GE 0 AND meanx+10 LE xsize AND meany-10 GE 0 AND meany+10 LE ysize THEN BEGIN
            if erat GE 1.1 AND erat LE vrat THEN BEGIN


              ;Believe the following FOR loop runs through the feature list and assigns each KLT point
with a rectangle region.
             FOR j = 0, maxResult-1 DO BEGIN
               IF (meanx-10 LE intresult[8,j] AND intresult[8,j] LE meanx+10) AND (meany-10 LE
intresult[9,j] AND intresult[9,j] LE meany+10) THEN BEGIN

                    IF (abs(z_direction[j]) GT 1.5) THEN BEGIN ;1.975
                    Frame2List_i.Add, [i]
                    Frame2List_x.Add, [intresult[8,j]]
                    Frame2List_y.Add, [intresult[9,j]]
                    Frame2List_value.Add, [result[10,j]]
                    Frame2List_direction.Add, [result[11,j]]
                    Frame2List_magnitude.Add, [result[12,j]]
```

```
Frame2List_Fr3x.Add, [intresult[13,j]]
Frame2List_Fr3y.Add, [intresult[14,j]]
Frame2List_Fr3value.Add, [result[15,j]]
Frame2List_Fr3direction.Add, [result[16,j]]
Frame2List_Fr3magnitude.Add, [result[17,j]]

Frame2List_Fr4x.Add, [intresult[13,j]]
Frame2List_Fr4y.Add, [intresult[14,j]]
Frame2List_Fr4value.Add, [result[15,j]]
Frame2List_Fr4direction.Add, [result[16,j]]
Frame2List_Fr4magnitude.Add, [result[17,j]]

;Add the KLT point to the rectangle overlay for display purposes
layerArray[intresult[8,j], intresult[9,j]] = i
pointcount = pointcount + 1

ENDIF ELSE BEGIN
IF (abs(z_magnitude[j])) GT 1.2 THEN BEGIN ;1.4
Frame2List_i.Add, [i]
Frame2List_x.Add, [intresult[8,j]]
Frame2List_y.Add, [intresult[9,j]]
Frame2List_value.Add, [result[10,j]]
Frame2List_direction.Add, [result[11,j]]
Frame2List_magnitude.Add, [result[12,j]]

Frame2List_Fr3x.Add, [intresult[13,j]]
Frame2List_Fr3y.Add, [intresult[14,j]]
Frame2List_Fr3value.Add, [result[15,j]]
Frame2List_Fr3direction.Add, [result[16,j]]
Frame2List_Fr3magnitude.Add, [result[17,j]]

Frame2List_Fr4x.Add, [intresult[13,j]]
Frame2List_Fr4y.Add, [intresult[14,j]]
Frame2List_Fr4value.Add, [result[15,j]]
Frame2List_Fr4direction.Add, [result[16,j]]
Frame2List_Fr4magnitude.Add, [result[17,j]]

;Add the KLT point to the rectangle overlay for display purposes
layerArray[intresult[8,j], intresult[9,j]] = i
pointcount = pointcount + 1
ENDIF ELSE BEGIN
Frame2List_i.Add, [0]
```

```
                    Frame2List_x.Add, [intresult[8,j]]
                    Frame2List_y.Add, [intresult[9,j]]
                    Frame2List_value.Add, [result[10,j]]
                    Frame2List_direction.Add, [result[11,j]]
                    Frame2List_magnitude.Add, [result[12,j]]

                    Frame2List_Fr3x.Add, [intresult[13,j]]
                    Frame2List_Fr3y.Add, [intresult[14,j]]
                    Frame2List_Fr3value.Add, [result[15,j]]
                    Frame2List_Fr3direction.Add, [result[16,j]]
                    Frame2List_Fr3magnitude.Add, [result[17,j]]

                    Frame2List_Fr4x.Add, [intresult[13,j]]
                    Frame2List_Fr4y.Add, [intresult[14,j]]
                    Frame2List_Fr4value.Add, [result[15,j]]
                    Frame2List_Fr4direction.Add, [result[16,j]]
                    Frame2List_Fr4magnitude.Add, [result[17,j]]
                    ENDELSE
                  ENDELSE

                  ;Frame2list contains all of the KLT points that pass the rectangle check in frame 2,
without any
                  ;zeroed rows.

                  ENDIF
                ENDFOR
                ; Only make a rectangle across regions containing at least one KLT point.
                IF pointcount NE 0 THEN BEGIN
                   rectArray[meanx-10:meanx+10, meany-10] = i
                   rectArray[meanx-10:meanx+10, meany+10] = i
                   rectArray[meanx-10, meany-10:meany+10] = i
                   rectArray[meanx+10, meany-10:meany+10] = i

                ;rectArray is an overlay of the course selection rectangles corresponding to each region
                ;layerArray is an overlay showing the points within course selection rectangles
corresponding to each region
                ; reg is a counter for the number of rectangles created.
                reg = reg + 1
                ENDIF
              ENDIF ELSE BEGIN
                region_array[WHERE(region_array EQ i)] = 0
              ENDELSE
```

```
        ENDIF

    ENDIF

END

;Remove first default row from Frame2List
Frame2list_i.remove, 0
Frame2list_x.remove, 0
Frame2list_y.remove, 0
Frame2list_value.remove, 0
Frame2list_direction.remove, 0
Frame2list_magnitude.remove, 0

Frame2list_Fr3x.remove, 0
Frame2list_Fr3y.remove, 0
Frame2list_Fr3value.remove, 0
Frame2list_Fr3direction.remove, 0
Frame2list_Fr3magnitude.remove, 0

Frame2list_Fr4x.remove, 0
Frame2list_Fr4y.remove, 0
Frame2list_Fr4value.remove, 0
Frame2list_Fr4direction.remove, 0
Frame2list_Fr4magnitude.remove, 0

;print, 'new list = ', Frame2list



min_layer = min(layerArray)
max_layer = max(layerArray)

;print, 'min layer array = ', min_layer
;print, 'max layer array = ', max_layer
;print, 'number of regions = ', reg

;WINDOW, /FREE, XSIZE = 1200, YSIZE = 950, TITLE = 'layerArray'
;tvscl, layerArray, /order
```

```
cgDisplay, XSIZE = 1200, YSIZE=950, /FREE, Title = 'layerArray'
cgImage, ROTATE(layerArray, 7 ), CTIndex = 40
;cgImage, ROTATE(TrackOverlay2, 7), CTIndex = 40, Transparent = 50, Missing_Value = 0
cgImage, ROTATE(rectArray, 7), CTIndex = 40, Transparent = 25, Missing_Value = 0


; Need to compute the KLT point closest to the center of the motion rectangle.
; That is, the KLT point whose distance to meanx and meany is the minimum.
;If there is only one KLT point within the rectangle, this returns the singleton point.

closest_to_ctr_vec = fltarr(2, max_layer+1)
nonzerocount = 1

motiondir = fltarr(max_layer+1)
motionmag = fltarr(max_layer+1)
closestToCtrOverlay = fltarr(xsize,ysize)
closestToCtr = fltarr(xsize,ysize)

FOR i = 1, max_layer DO BEGIN
; FOR i = 179, 180 DO BEGIN
  temp = WHERE(layerArray EQ i, count)
  IF count NE 0 THEN BEGIN

    indices = Array_Indices(layerArray, temp)

    ;print, 'indices = ', indices

    meanx = MEAN(indices[0, *])
    meany = MEAN(indices[1, *])

    ;print, 'meanx = ', meanx
    ;print, 'meany = ', meany

    nreps = N_ELEMENTS(indices)/2

    centerpt = [meanx, meany]
    centervec = replicate_vector(centerpt, nreps)

    dist_to_center = sqrt( (centervec[0,*] - indices[0,*])^2 + (centervec[1,*] - indices[1,*])^2)
    mindist = min(dist_to_center, min_subscript)

    ;print, 'dist_to_center = ', dist_to_center
```

```
    ;print, 'min subscript = ', min_subscript
    ;print, 'mindist = ', mindist

    closest_to_ctr = [indices[0,min_subscript], indices[1, min_subscript]]

    ;print, 'closest_to_ctr = ', closest_to_ctr

    closest_to_ctr_vec[*, i] = closest_to_ctr

    ;Build an overlay where the pixel closest to teh center of the region
    ;has the same value as the region.
    closestToCtr[indices[0,min_subscript], indices[1, min_subscript]] = i
    ;This makes centerpoints larger than 1 pixel in a separate overlay
    closestToCtrOverlay[indices[0,min_subscript]+1, indices[1, min_subscript]] = i
    closestToCtrOverlay[indices[0,min_subscript]-1, indices[1, min_subscript]] = i
    closestToCtrOverlay[indices[0,min_subscript], indices[1, min_subscript]+1] = i
    closestToCtrOverlay[indices[0,min_subscript], indices[1, min_subscript]-1] = i

    nonzerocount = nonzerocount + 1
  ENDIF
ENDFOR

;print, 'closest to center vector = ', closest_to_ctr_vec


; Create an overlay that shows the points closest to the center of the rectangle -------------
cgDisplay, XSIZE = 1200, YSIZE=950, /FREE, Title = 'Coarse motion layers w ctr points'
cgImage, ROTATE(megafile[*,*,0], 7 ), CTIndex = 0
cgImage, ROTATE(layerArray, 7 ), CTIndex = 40, Transparent = 5, Missing_Value = 0
cgImage, ROTATE(rectArray, 7), CTIndex = 40, Transparent = 15, Missing_Value = 0
cgImage, ROTATE(closestToCtrOverlay, 7), CTIndex = 3, Transparent = 25, Missing_Value = 0


; ------------------------------------------------------------------------------------------

;Have a long vector with a lot of zeroed rows that contains the nonzero points that are closest to the
center of each region.
;Want to get rid of zeros and keep only centerpoints of regions. Then, need to find motion of that
center point

arrayNoZeros = Where(closestToCtr NE 0)
;help, arrayNoZeros
```

```
ClosestIndices = Array_Indices(closestToCtr, arrayNoZeros)
;print, 'Closest to Ctr indices = ', ClosestIndices
;help, ClosestIndices


;Now, retrieve the motion history of that KLT point that is nearest the center of the motion layer.
;Current code computes point closest to center in frame 2. Find history of motion from frame 1

;print, 'N_ELEMENTS FRAME2List_i = ', N_ELEMENTS(Frame2List_i)

;Recall, frame2List has the form [i, intresult[8,j], intresult[9,j], result[10,j], result[11,j],
result[12,j]
;make frame2list into an array

Frame2Array_i = Frame2List_i.Toarray(TYPE = 'FLOAT')
Frame2Array_x = Frame2List_x.Toarray(TYPE = 'FLOAT')
Frame2Array_y = Frame2List_y.Toarray(TYPE = 'FLOAT')
Frame2Array_value = Frame2List_value.Toarray(TYPE = 'FLOAT')
Frame2Array_direction = Frame2List_direction.Toarray(TYPE = 'FLOAT')
Frame2Array_magnitude = Frame2List_magnitude.Toarray(TYPE = 'FLOAT')

Frame2Array_Fr3x = Frame2List_Fr3x.Toarray(TYPE = 'FLOAT')
Frame2Array_Fr3y = Frame2List_Fr3y.Toarray(TYPE = 'FLOAT')
Frame2Array_Fr3value = Frame2List_Fr3value.Toarray(TYPE = 'FLOAT')
Frame2Array_Fr3direction = Frame2List_Fr3direction.Toarray(TYPE = 'FLOAT')
Frame2Array_Fr3magnitude = Frame2List_Fr3magnitude.Toarray(TYPE = 'FLOAT')

Frame2Array_Fr4x = Frame2List_Fr4x.Toarray(TYPE = 'FLOAT')
Frame2Array_Fr4y = Frame2List_Fr4y.Toarray(TYPE = 'FLOAT')
Frame2Array_Fr4value = Frame2List_Fr4value.Toarray(TYPE = 'FLOAT')
Frame2Array_Fr4direction = Frame2List_Fr4direction.Toarray(TYPE = 'FLOAT')
Frame2Array_Fr4magnitude = Frame2List_Fr4magnitude.Toarray(TYPE = 'FLOAT')

Frame2Array = fltarr(16, N_ELEMENTS(Frame2Array_i))
Frame2Array = [ TRANSPOSE(Frame2Array_i), TRANSPOSE(Frame2Array_x), TRANSPOSE(Frame2Array_y),$
                TRANSPOSE(Frame2Array_value),TRANSPOSE(Frame2Array_direction),
TRANSPOSE(Frame2Array_magnitude),$
                TRANSPOSE(Frame2Array_Fr3x), TRANSPOSE(Frame2Array_Fr3y),$
                TRANSPOSE(Frame2Array_Fr3value),TRANSPOSE(Frame2Array_Fr3direction),
TRANSPOSE(Frame2Array_Fr3magnitude),$
                TRANSPOSE(Frame2Array_Fr4x), TRANSPOSE(Frame2Array_Fr4y),$
```

```
                TRANSPOSE(Frame2Array_Fr4value),TRANSPOSE(Frame2Array_Fr4direction),
TRANSPOSE(Frame2Array_Fr4magnitude)]


sz = size(ClosestIndices, /DIMENSIONS)
;print, 'sz = ', sz

sz_frame2Array = size(Frame2Array, /DIMENSIONS)


motiondir = fltarr(sz[1])
motionmag = fltarr(sz[1])
regionNumber = fltarr(sz[1])
tempmotiondir = 0.0
tempmotionmag = 0.0


;Find the motion of the point closest to the center of the rectangle:
for m = 0, sz[1]-1 DO BEGIN
  for n = 0, sz_frame2Array[1]-1 DO BEGIN ;Was n = 0, max_result-1

    ; Was IF (intresult[8,n] EQ ClosestIndices[0,m] AND intresult[9,n] eq ClosestIndices[1,m]) THEN
BEGIN
    IF (fix(Frame2Array[1,n]) EQ ClosestIndices[0,m]) AND (fix(Frame2Array[2,n]) eq ClosestIndices[1,m])
THEN BEGIN

      ;motion of point closest to center
      motiondir[m] = [Frame2Array[4,n]]
      motionmag[m] = [Frame2Array[5,n]]

      ;Find region number for the centerpixel of interest
      regionNumber[m] = [Frame2Array[0,n]]

      ; Printed value is the pixel location of each point in region
      ; in frame 2, and the motion direction and magnitude calculated between frame 1 and 2.
      ;print, 'found one: indices! ', Frame2Array[1,n], ClosestIndices[0,m], Frame2Array[2,n],
ClosestIndices[1,m],$
      ;     'FrameRegion = ', Frame2Array[0,n]

    ENDIF
  ENDFOR
ENDFOR
```

```
;print, 'starting motion compare loop'

;This loop compares the motion of all of the KLT points assigned to a region to the
;motion of the point closest to the center of the region. If the motion is equal,
;the point is kept in the region. If the motion is not equal, the point is thrown away.

mag_is_equal = 0
dir_is_equal = 0
CenterPTisNew = 0
Trackedpoint = 0
not_in_layerArray = 0
new_objects_in_layerArray = 0
Trackpoint_is_new = 0
centerpoint_is_new = 0
FinallayerArray2 = fltarr(xsize, ysize)

;print, 'motion direction = ', motiondir[0:10]
;print, 'Frame2Array = ', Frame2Array[*,0:10]

FOR m = 0, sz[1]-1 DO BEGIN
;print, 'm = ', m

;Check to see if Centerpoint is new. If not, then begin
  ;(Really, if the centerpoint is new for frame 2 then I really should have chosen a
  ; point that was closest to teh center with some motion history. Instead,
  ;I threw away the layer if the center point had no motion history.)

CenterPTisNew = is_equal(motionmag[m], 0.000, ACCEPTABLE_DIFFERENCE = 0.001)

  IF CenterPTisNew EQ 0 THEN BEGIN

    FOR n = 0, sz_frame2Array[1]-1 DO BEGIN

      IF Frame2Array[0,n] EQ regionNumber[m] THEN BEGIN

        ;Check to see if the KLT point is new, or if it has been tracked.
        ;If tracked, (is equal check returns value of 1 = TRUE), then begin.
        Trackedpoint = is_equal(Frame2Array[3,n], 0.000, ACCEPTABLE_DIFFERENCE = 0.001)
        IF Trackedpoint EQ 1 THEN BEGIN
           tempmotionmag = Frame2Array[5,n]
```

```
            tempmotiondir = Frame2Array[4,n]

            MS = exp(abs(motionmag[m] - tempmotionmag)+abs(motiondir[m]-tempmotiondir))
           ;print, 'motion simularity = ', MS
             ;IF dir_is_equal EQ 0 THEN BEGIN
             IF MS GT 50 THEN BEGIN
                Frame2Array[0,n] = 0
                not_in_layerArray = not_in_layerArray + 1
                ;print, 'not in layer array'
             ENDIF ELSE BEGIN
               ;print, 'found a match'
               FinallayerArray2[ Frame2Array[1,n], Frame2Array[2,n] ] = Frame2Array[0,n]
             ENDELSE

          ENDIF ELSE BEGIN
           Trackpoint_is_new = Trackpoint_is_new + 1
           ;print, 'Tracked Point is new for frame 2'
          ENDELSE
        ENDIF

      ENDFOR


     ENDIF ELSE BEGIN
       centerpoint_is_new = centerpoint_is_new + 1
       ;print, 'centerpoint is new'
     ENDELSE
  ENDFOR
    print, 'not in layer array = ', not_in_layerArray
    print, 'new centerpoints = ', centerpoint_is_new
    print, 'new tracked point = ', trackpoint_is_new



;cgDisplay, XSIZE = 1200, YSIZE=950, /FREE, Title = 'Initial detected layers, frame 2'
;cgImage, ROTATE(layerArray, 7 ), CTIndex = 40
;cgImage, ROTATE(rectArray, 7), CTIndex = 40, Transparent = 25, Missing_Value = 0
;cgImage, ROTATE(closestToCtrOverlay, 7), CTIndex = 40, Transparent = 25, Missing_Value = 0
;cgImage, ROTATE(rescaled_difference, 7 ), CTIndex = 0, Transparent = 50, Missing_Value = 0
```

```
cgDisplay, XSIZE = 1200, YSIZE=950, /FREE, Title = 'Final motion layers, frame 2, overlaying differenced
image'
cgImage, ROTATE(rescaled_difference, 7 ), CTIndex = 0
cgImage, ROTATE(FinallayerArray2, 7 ), CTIndex = 40, Transparent = 5, Missing_Value = 0
cgImage, ROTATE(rectArray, 7), CTIndex = 40, Transparent = 15, Missing_Value = 0
cgImage, ROTATE(closestToCtrOverlay, 7), CTIndex = 40, Transparent = 25, Missing_Value = 0


cgDisplay, XSIZE = 1200, YSIZE=950, /FREE, Title = 'Final motion layers overlaying original frame 2'
cgImage, ROTATE(megafile[*,*,0], 7 ), CTIndex = 0
cgImage, ROTATE(FinallayerArray2, 7 ), CTIndex = 40, Transparent = 5, Missing_value = 0
cgImage, ROTATE(rectArray, 7), CTIndex = 40, Transparent = 15, Missing_Value = 0
cgImage, ROTATE(closestToCtrOverlay, 7), CTIndex = 40, Transparent = 25, Missing_Value = 0


;******************END FRAME 2 MOTION LAYER CALCULATION ********************************************
;*************************************************************************************************
;*************************************************************************************************

;Next, maintain the motion layer into frame 3.
;This part can be looped X times before running the temporal differncing
;steps again.
;
;Move the rectangles based on centerpoint motion.

max_region = max(rectArray)
;print, 'max region = ', max_region
newrectArray = fltarr(xsize, ysize)

frame3Array = !NULL
Frame3layerArray = fltarr(xsize, ysize)

FOR i = 1, max_region DO BEGIN
; FOR i = 179, 180 DO BEGIN
  temp = WHERE(rectArray EQ i, count)
  IF count NE 0 THEN BEGIN
     pointcount = 0
     FOR m = 0, sz[1]-1 DO BEGIN
       IF regionNumber[m] EQ i THEN BEGIN
           IF motiondir[m] NE 0.0 THEN BEGIN

               indices = Array_Indices(rectArray, temp)
```

```
                meanx = MEAN(indices[0, *])
                meany = MEAN(indices[1, *])

                centerpt = [meanx, meany] ;true center of rectangle

                newx = (motionmag[m] * cos(!pi/180*motiondir[m]))+ meanx
                newy = (sin(!pi/180*motiondir[m]) * motionmag[m])+ meany

               ; print, 'motion mag = ', motionmag[m]
               ; print, 'motion dir = ', motiondir[m]

               ; print, 'newx = ', newx
               ; print, 'newy = ', newy

                newcenterpt = [newx, newy]
                nreps = N_ELEMENTS(indices)/2
                centervec = replicate_vector(newcenterpt, nreps)

                 ;Since we moved the rectangle, recheck all KLT points in frame 3 to see if they
                 ;fall inside the moved course rectangle. Remove any rectangles without any KLT points
                 ;By rechecking all points, we are sure to capture lost track points the KLT algorithm
                 ;replaced with new points, assuming the new points are tracked into frame 3.
                 FOR j = 0, maxResult-1 DO BEGIN

                   IF (newx-10 LE intresult[13,j] AND intresult[13,j] LE newx+10) AND (newy-10 LE
intresult[14,j] AND intresult[14,j] LE newy+10) THEN BEGIN

                   ;Frame3Array contains all of the KLT points that pass the rectangle check in frame 3,
without any
                   ;zeroed rows.
                    Frame3Array = [ [Frame3Array], [i, intresult[13,j], intresult[14,j], result[15,j],
result[16,j], result[17,j]]  ]
                    ;Add the KLT points to an overlay for display purposes
                    Frame3layerArray[intresult[13,j], intresult[14,j]] = i
                    pointcount = pointcount + 1

                     IF pointcount NE 0 THEN BEGIN
                       newrectArray[newx-10:newx+10, newy-10] = i
                       newrectArray[newx-10:newx+10, newy+10] = i
                       newrectArray[newx-10, newy-10:newy+10] = i
                       newrectArray[newx+10, newy-10:newy+10] = i
```

```
              ;newrectArray is an overlay of the moved coarse selection rectangles corresponding to each
region
              ;Frame3layerArray is an overlay showing the points within coarse selection rectangles
              ;corresponding to each region in frame 3
              ; reg is a counter for the number of rectangles created.
                      reg = reg + 1
                    ENDIF
                  ENDIF
                ENDFOR
              ENDIF
          ENDIF
       ENDFOR
    ENDIF
ENDFOR

;Recompute KLT point closest to the center of the rectangle

closestToCtrOverlay = fltarr(xsize,ysize)
closestToCtr = fltarr(xsize,ysize)

FOR i = 1, max_region DO BEGIN
  temp = WHERE(Frame3layerArray EQ i, count)
  IF count NE 0 THEN BEGIN
    indices = Array_Indices(Frame3layerArray, temp)
    ;print, 'indices = ', indices
    meanx = MEAN(indices[0, *])
    meany = MEAN(indices[1, *])
    ;print, 'meanx = ', meanx
    ;print, 'meany = ', meany
    nreps = N_ELEMENTS(indices)/2

    centerpt = [meanx, meany]
    centervec = replicate_vector(centerpt, nreps)

    dist_to_center = sqrt( (centervec[0,*] - indices[0,*])^2 + (centervec[1,*] - indices[1,*])^2)
    mindist = min(dist_to_center, min_subscript)

    ;print, 'dist_to_center = ', dist_to_center
    ;print, 'min subscript = ', min_subscript
    ;print, 'mindist = ', mindist
    closest_to_ctr = [indices[0,min_subscript], indices[1, min_subscript]]
```

```
    ;print, 'closest_to_ctr = ', closest_to_ctr
    closest_to_ctr_vec[*, i] = closest_to_ctr

    ;Build an overlay where the pixel closest to teh center of the region
    ;has the same value as the region.
    closestToCtr[indices[0,min_subscript], indices[1, min_subscript]] = i
    ;This makes centerpoints larger than 1 pixel in a separate overlay
    closestToCtrOverlay[indices[0,min_subscript]+1, indices[1, min_subscript]] = i
    closestToCtrOverlay[indices[0,min_subscript]-1, indices[1, min_subscript]] = i
    closestToCtrOverlay[indices[0,min_subscript], indices[1, min_subscript]+1] = i
    closestToCtrOverlay[indices[0,min_subscript], indices[1, min_subscript]-1] = i

    nonzerocount = nonzerocount + 1
  ENDIF
ENDFOR
;Remove zeroes from the array
arrayNoZeros = Where(closestToCtr NE 0)
ClosestIndices = Array_Indices(closestToCtr, arrayNoZeros)

;Now, retrieve the motion history of that KLT point that is nearest the center of the motion layer.
;Find the motion of the point closest to the center of the rectangle:
sz = size(ClosestIndices, /DIMENSIONS)
;print, 'sz = ', sz

sz_frame3Array = size(Frame3Array, /DIMENSIONS)

motiondir = fltarr(sz[1])
motionmag = fltarr(sz[1])
regionNumber = fltarr(sz[1])
tempmotiondir = 0.0
tempmotionmag = 0.0

for m = 0, sz[1]-1 DO BEGIN
  for n = 0, sz_frame3Array[1]-1 DO BEGIN

    IF (fix(Frame3Array[1,n]) EQ ClosestIndices[0,m]) AND (fix(Frame3Array[2,n]) eq ClosestIndices[1,m])
THEN BEGIN

      ;motion of point closest to center
      motiondir[m] = [Frame3Array[4,n]]
      motionmag[m] = [Frame3Array[5,n]]
```

```
      ;Find region number for the centerpixel of interest
      regionNumber[m] = [Frame3Array[0,n]]

      ; Printed value is the pixel location of each point in region
      ; in frame 3, and the motion direction and magnitude calculated between frame 2 and 3.
      ;print, 'found one: indices! ', Frame3Array[1,n], ClosestIndices[0,m], Frame3Array[2,n],
ClosestIndices[1,m],$
       ;    'FrameRegion = ', Frame3Array[0,n]

    ENDIF
  ENDFOR
ENDFOR

;Compare points in rectangle against motion of centerpoint
FinallayerArray3 = fltarr(xsize, ysize)

FOR m = 0, sz[1]-1 DO BEGIN
CenterPTisNew = is_equal(motionmag[m], 0.000, ACCEPTABLE_DIFFERENCE = 0.001)

  IF CenterPTisNew EQ 0 THEN BEGIN

    FOR n = 0, sz_frame3Array[1]-1 DO BEGIN

      IF Frame3Array[0,n] EQ regionNumber[m] THEN BEGIN

        ;Check to see if the KLT point is new, or if it has been tracked.
        ;If tracked, (is equal check returns value of 1 = TRUE), then begin.
        Trackedpoint = is_equal(Frame3Array[3,n], 0.000, ACCEPTABLE_DIFFERENCE = 0.001)
        IF Trackedpoint NE 0 THEN BEGIN
           tempmotionmag = Frame3Array[5,n]
           tempmotiondir = Frame3Array[4,n]

          ; mag_is_equal = is_equal(motionmag[m], tempmotionmag, ACCEPTABLE_DIFFERENCE = 0.01) ;changed
from .001
          ; dir_is_equal = is_equal(motiondir[m], tempmotiondir, ACCEPTABLE_DIFFERENCE = 0.01) ;changed
from .001

           MS = exp(abs(motionmag[m] - tempmotionmag)+abs(motiondir[m]-tempmotiondir))
            ;print, 'motion simularity = ', MS
            ;IF dir_is_equal EQ 0 THEN BEGIN
            IF MS GT 33 THEN BEGIN
               Frame3Array[0,n] = 0
```

```
               not_in_layerArray = not_in_layerArray + 1
               ;print, 'not in layer array'
             ENDIF ELSE BEGIN
               ;print, 'found a match'
               FinallayerArray3[ Frame3Array[1,n], Frame3Array[2,n] ] = Frame3Array[0,n]
             ENDELSE
          ENDIF ELSE BEGIN
           Trackpoint_is_new = Trackpoint_is_new + 1
           ;print, 'Tracked Point is new for frame 2'
          ENDELSE
        ENDIF

      ENDFOR


     ENDIF ELSE BEGIN
       centerpoint_is_new = centerpoint_is_new + 1
       ;print, 'centerpoint is new'
     ENDELSE
ENDFOR
  print, 'not in layer array = ', not_in_layerArray
  print, 'new centerpoints = ', centerpoint_is_new
  print, 'new tracked point = ', trackpoint_is_new

frame3_image = bytarr(xsize, ysize)
frame3_image = megafile[*,*,1]

cgDisplay, XSIZE = 1200, YSIZE=950, /FREE, Title = 'moved rectangles'
cgImage, ROTATE(frame3_image, 7), CTIndex = 0
;cgImage, ROTATE(layerArray, 7 ), CTIndex = 40
cgImage, ROTATE(rectArray, 7), CTIndex = 40, Transparent = 5, Missing_Value = 0
;cgImage, ROTATE(newrectArray, 7), CTIndex = 40, Transparent = 5, Missing_Value = 0
cgImage, ROTATE(newrectArray, 7), CTIndex = 40, Transparent = 15, Missing_Value = 0

cgDisplay, XSIZE = 1200, YSIZE=950, /FREE, Title = 'Frame 3 Coarse Layers'
cgImage, ROTATE(frame3_image, 7), CTIndex = 0
cgImage, ROTATE(Frame3layerArray, 7 ), CTIndex = 40, Transparent = 5, Missing_value = 0
cgImage, ROTATE(newrectArray, 7), CTIndex = 40,  Transparent = 15, Missing_Value = 0
cgImage, ROTATE(closestToCtrOverlay, 7), CTIndex = 3,  Transparent = 25, Missing_Value = 0


cgDisplay, XSIZE = 1200, YSIZE=950, /FREE, Title = 'Final Frame 3 Layers'
```

```
cgImage, ROTATE(frame3_image, 7), CTIndex = 0
cgImage, ROTATE(FinallayerArray3, 7 ), CTIndex = 40, Transparent = 5, Missing_Value = 0
cgImage, ROTATE(newrectArray, 7), CTIndex = 40,  Transparent = 15, Missing_Value = 0
cgImage, ROTATE(closestToCtrOverlay, 7), CTIndex = 40,  Transparent = 25, Missing_Value = 0


print, 'END OF PROGRAM'
END ; End of program
```