Method For Naturalistic Measurement Of Lane-Keeping Behavior

FINAL REPORT

UMTRI-98-51
Contract No.: DTFH-62-93-00017

Gordon McQuade
Robert D. Ervin
Karl C. Kluge

Prepared by:
The University of Michigan
Transportation Research Institute
2901 Baxter Road, Ann Arbor, MI 48109-2150

October 1998

Technical Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle
Method for Naturalistic Measurement of
Lane-Keeping Behavior

5. Report Date
October 1998

6. Performing Organization Code

7. Author(s)
McQuade, G., Ervin, R., Kluge, K.

8. Performing Organization Report No.
UMTRI-98-51

9. Performing Organization Name and Address
The University of Michigan
Transportation Research Institute
2901 Baxter Road, Ann Arbor, Michigan 48109-2150

10. Work Unit No. (TRAIS)

11. Contract or Grant No.
DTFH-62-93-00017

12. Sponsoring Agency Name and Address
ITS Research Center of Excellence
Federal Highway Administration, USDOT
6300 Georgetown Pike, Office HSR-1D
McClean, VA 22101

13. Type of Report and Period Covered
Final Report

14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

A measurement method was developed and applied for capturing the continuous lateral position of naturally prevailing motor vehicles as they are driven on highways. This lane-keeping measurement technique employs a video-equipped vehicle which is used in simply following other road users, recording episodes of steady lane following. Video records are then processed off-line to quantify the continuous lateral displacement of the centroid of the license plate on the preceding vehicle relative to the lane center line. Data are processed for as long a time period as the preceding vehicle remains within an individual travel lane. Data of this type are thought to have distinct value in supporting the engineering development of various forms of driver-assistance products such as road-departure warning, adaptive cruise control (ACC) lane-change aids, and the like.

Algorithms for processing the video data are discussed as are procedural details by which human observation assists in the data-processing sequence. Example data that were obtained during daylight travel on well-marked roadways are presented and discussed. Although the measurement method has been shown to be effective in collecting the desired form of data, recommendations are made for improvements in both the efficiency and accuracy of the data-processing task.

17. Key Words	18. Distribution Statement
lane-keeping, lateral position, video processing, naturalistic driving behavior, on-highway measurements	Unrestricted

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
None	None	97	

Table of Contents

Naturalistic Measurement of Lane-Keeping Behavior

1.0 Introduction

A project was undertaken to develop and apply an experimental method for obtaining naturalistic data that would quantify the lateral lane-keeping exhibited by passenger vehicles. The project was conducted by the University of Michigan Transportation Research Institute, under sponsorship by Toyota Motor Company and affiliated sponsors of the Intelligent Transportation Systems Research Center of Excellence at the University of Michigan. The project focussed upon a means to collect measurements of the lateral position of vehicles relative to lane-edge locations, using a video-imaging technique based upon an instrumented vehicle with which the experimenter simply follows normal road users for a limited period of driving on freeways.

By way of background, it is apparent that the development of many types of driver-assistance systems, even some that address longitudinal-control functions and oblique or blind-spot forms of crash avoidance warning, would benefit from knowledge of the normal lane-keeping behavior of drivers. The probabilistic distribution of lateral proximity to the lane boundary, and its derivatives, curvature, and bandwidth, are seen as ultimately important to the support of engineering decisions. Further, it may be valuable to determine such characteristics as a function of forward speed, lane curvature and transition, and ambient traffic, at the least. Example applications of such data include:

1) design of road-departure warning algorithms to account for the variations in road-edge approach that arise from the personal driving styles of individuals, so as to tune the system performance trade-offs of misses versus false alarms and to evaluate the need for driver-adaptive designs;

2) design of adaptive cruise control (ACC) systems which must detect and respond by braking to avoid stopped objects in one's path ahead. This difficult requirement must eventually be satisfied with due cognizance to the variation that exists in normal path-following behavior.

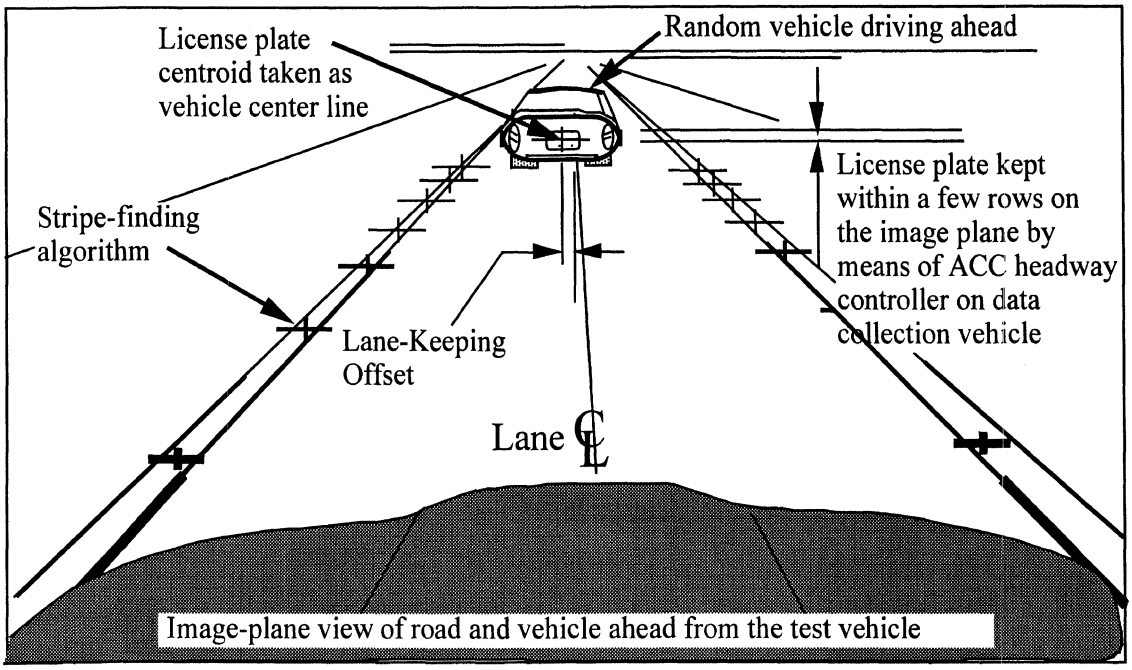
3) design of blind-spot warning devices will benefit from information showing the distributions of vehicle lateral position both during lane-keeping and leading up to lane-change maneuvers.

4) an extension of the technique to measure the lane-change transient, itself, as an aid in the design of many driver-assistance systems that must accomodate lane change transitions in the context of their primary functionalities.

Based upon the data needs implied by these possible applications, this project was undertaken to go as completely as possible "from A to Z" in the creation of a measurement method. The report covers the progress made in this pursuit. Namely, an operative concept was defined, as outlined briefly in Section 2, and it was developed into a working tool and applied in the field, as discussed in Sections 3 and 4, respectively. Section 3 presents the design approach that was taken to determine the locations of both the highway lane and an effective centroid of the preceding vehicle from which a lateral displacement, or offset, variable could be obtained. Computer code underlying the methods of video image analysis are included in Appendices A and B. Section 4 summarizes the field measurement procedure that was implemented to collect and then process data, samples of which are presented and discussed in Section 5. A complete listing of the data channels produced from these measurements is presented in Appendix C and a larger sample of actual lane-keeping results, in the form of histograms, is attached as Appendix D. Section 6 presents conclusions and recommendations drawn from the study.

2.0 Measurement Concept

The basic concept for this project was that the lane-keeping behavior of randomly selected vehicles could be meaningfully sampled over modestly long driving episodes by means of video recording from a following vehicle, as sketched below.



Features of the method concept are cited below, along with parenthetical comments indicating how the actual scope of work differed from the concept that has been envisioned for long-term development and application:

a) the driver of the lead vehicle was assumed to behave naturalistically (although, under sparse traffic conditions, it has become apparent that a lone driver may tend to become uncomfortable when the observer vehicle approaches and then retains a following position at relatively close range—e.g., at a headway time of approximately 1.5 seconds);

b) the data collection vehicle is equipped with a video camera and with additional instruments for measuring speed, yaw rate, and range to the vehicle ahead (measurements from yaw rate and range sensors were not taken during this initial stage of development);

c) the data collection vehicled is further equipped with an ACC system for controlling the range at a regularized distance so as to simplify video processing of the preceding vehicle image (this technique was employed and was found to be a distinct help in securing a more or less constant framing of the image of the vehicle ahead);

d) the image-based and numerical data are processed off-line, beginning with a software package that measures lateral position in the lane by detecting and tracking lane-edge lines (the provisions for image processing of lane-edge lines are discussed in Section 3.5) ;

e) the center line of the preceding vehicle is defined by locating and tracking the imaged centroid of the license plate—the primary feature extracted from the image of the preceding vehicle (using an algorithm that is presented in Section 3.6);

f) the center line of the roadway is defined by the best fit function lying midway between the imaged lane-edge stripes;

g) data would be collected in the field during steady-state following, at the ACC-established headway condition, for as long as a selected target vehicle sustained travel in a single lane (as discussed in Section 4.0).

The remainder of the report documents the data collection and processing methods and also presents and discusses a small sampling of results.

3.0 Elements of the Measurement Method

The method whereby lateral-lane-position data are measured consists of two phases, collection and analysis. In this discussion, the raw data are simply recorded sequences of video images. Each sequence, or clip, of video is one episode of observation of a target vehicle as it proceeds immediately ahead of the observer vehicle. In the presentation which follows, the techniques for collecting and analyzing each video clip are described.

3.1 On-Board Data-Collection System

Headway distance, or range between observer and target vehicle is maintained by the UMTRI-developed ACC. Steering control and all other kinds of required intervention are provided by the human driver of the observer vehicle. Figure 1 shows a schematic representation of the raw data-acquisition system used to acquire data while driving behind a selected target vehicle.

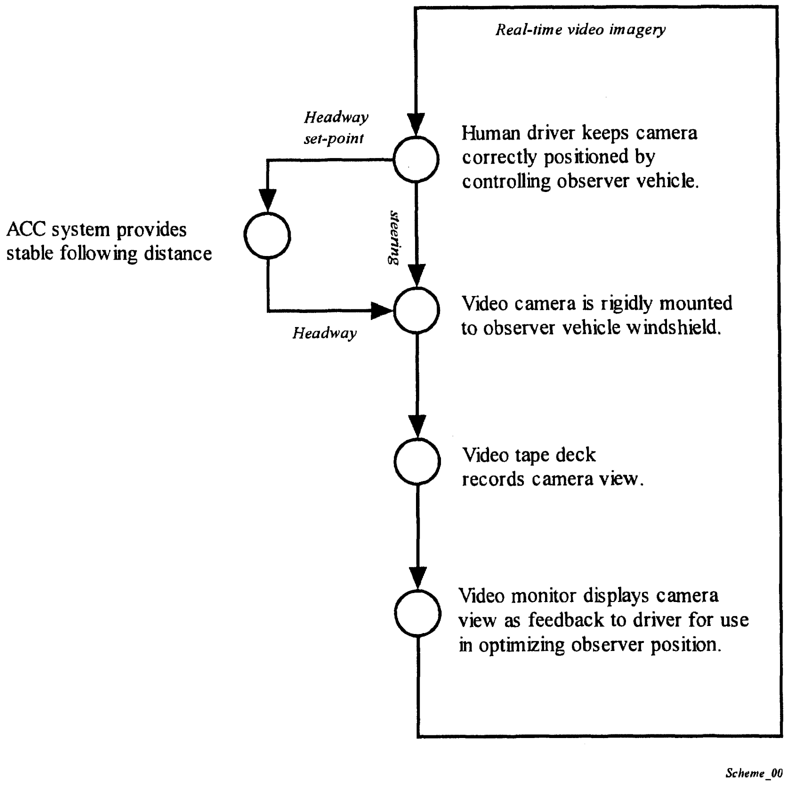


Figure 1. The driving process within which video data are acquired.

3.2 Image File Creation

Once the observation episodes have been taped, the next step is the creation of video-clip computer disk files. These files are then available for processing by a desktop computer-hosted software tool which analyzes the computerized video and extracts the data of interest. An overview of the image file creation process is now presented.

A videocassette on which has been recorded one or more observation episodes is mounted in a computer-controllable tape deck located near the computer. The tape is first manually positioned at the beginning of the clip of interest. Then a software utility is invoked to digitize and store on hard disk the clip of interest. Once the clip of interest resides on disk in digitized form the analysis tool can be used to extract the lateral lane-keeping data.

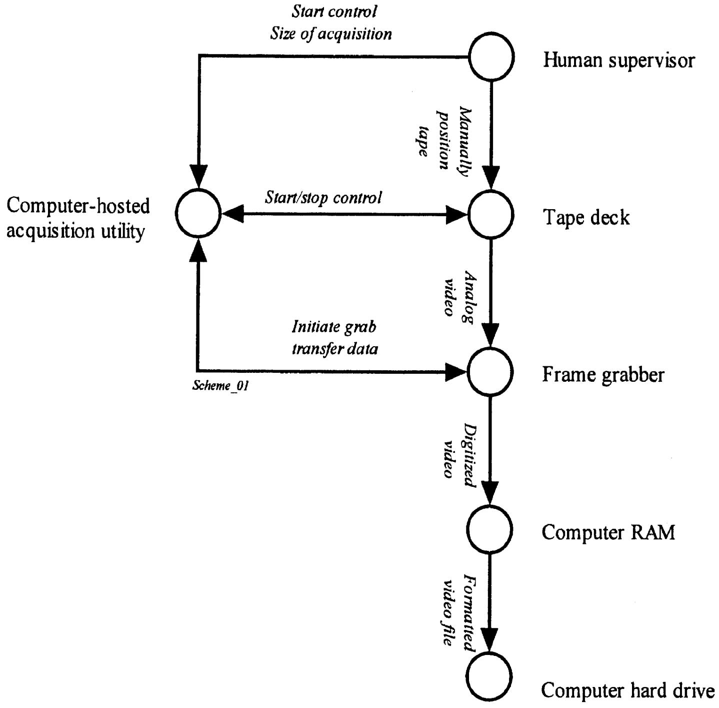


Figure 2. Video clip digitization process.

The digitization process is carried out with the use of both a computer-controllable video tape deck and a low-cost frame-grabber device. The tape deck is located near the desktop computer while the frame grabber is actually hosted by the computer as an internal plug-in. Serial communications are established between the tape deck and the desktop computer via an external cable. The tape is manually positioned to the beginning of the episode of interest and the digitization utility is invoked with an argument specifying the number of video frames to acquire. The video sequence is acquired at the nominal rate of 30 hertz and stored on disk for subsequent computerized analysis. See figure 2 for an illustration of this process.

3.3 Desktop Utility for Manual Interaction with Data Processing

Once a clip has been digitized it is in a format compatible with automatic data processing. The utility which accomplishes this processing is a customized tool designed specifically for this task by UMTRI staff. An overview of the main features of this tool and its operation are the subject of this section.

The task of analyzing a sequence of digitized frames of a given video clip is fundamentally one of automatic image understanding. The desktop machine is capable of understanding a visual scene by computing the contextual relationships existing among some set of objects present within the scene. For our purposes, that set of objects is composed of the right and left lane boundaries, the subject vehicle's license plate, and lastly, the shadow which the subject vehicle casts beneath itself.

Once these objects have been initially located, the tool simply tracks them from frame to frame. This is possible because the narrow bandwidth of vehicle motion in the yaw plane yields relatively smooth motion from frame to frame. Large-amplitude bouncing of the observer vehicle due to bumpy freeway road surfaces does present a problem on rougher roads, but should be solvable in the future given auxiliary information such as accelerometer data. The present implementation kept things very simple: only smooth, relatively straight segments of highway were selected as the venue for initial observations of target vehicles. Video clips obtained under such conditions have been efficiently handled with minimum operator intervention.

The machine requires relatively strong contrast gradients in order to determine the boundary of an object. Given the fluctuations in ambient lighting occurring during any given normal drive down a freeway, sometimes the machine will lose track of an object. A common example is when the subject vehicle passes under a bridge during conditions of very bright general ambient illumination. When this happens at highway speeds, the automatic shutter mechanism of the charge coupled device (CCD) camera does not have sufficient time to adjust to the new, much darker lighting conditions beneath the bridge, and the tracker will not be able to detect the subject vehicle's license plate boundary for several frames. Often this will cause the tracking algorithm to lose the subject vehicle. When such a loss of target occurs, the operator must intervene and reinitialize the tracker.

3.4 Manual Identification of Targeted Zones

The data-analysis system needs a human operator to establish initial conditions for its object recognition and tracking process to start up properly. These initial conditions are the initial search zones within the image frame. The system is configured to find the various objects it needs to recognize within these zones. The license plate recognizer requires an additional initial condition as well: an accurate estimate of the initial width of the license plate. The units of this value are in pixels. Experience with the tool has demonstrated that an operator can get pretty good at guessing closely the initial width. Given a modest degree of inaccuracy in the estimate of initial width, the license-plate-tracking routine will adapt to the correct width as the first several frames are analyzed.

For purposes of estimating headway distance to the target vehicle, the target's cast shadow is also identified as an initial condition. The motivation for such an activity is based upon the observation that the lower edge of a shadow cast on the road surface by a vehicle falls consistently within a few inches either way of the plane containing the rearmost extremity of that vehicle. Hence the target's rear shadow is used to initially locate a scan line within the image frame which is taken to be at the same distance as the rear of the target vehicle. That scan line intersects both the right and left lane boundaries as they present themselves in the image in perspective view. Given knowledge that these lane-boundary markings are nominally at a separation of twelve feet, we now have the means of scaling our image pixels to the real world engineering units of inches, feet, or meters. Once the target vehicle's rear shadow has been initially located, it is not tracked. The reason is that once we have a scan line offset from the lower shadow boundary to the license plate centroid, we will be able to adjust the target's shadow location solely by using the ratio of the current license plate width to the initial license plate width. This situation is represented by the following relation

$$h = \frac{w}{w_i} h_i ,$$

where h is the current separation between license plate centroid and scan line containing rear of subject vehicle and w is current width of the license plate. The same terms subscripted represent their respective initial values.

Lastly, the operator is required to select the best zone in which to locate and subsequently track the near-field lane-boundary markers. More will be said about the notion of a field zone in the next section. Figure 3 illustrates an initial video-clip frame with the regions selected.

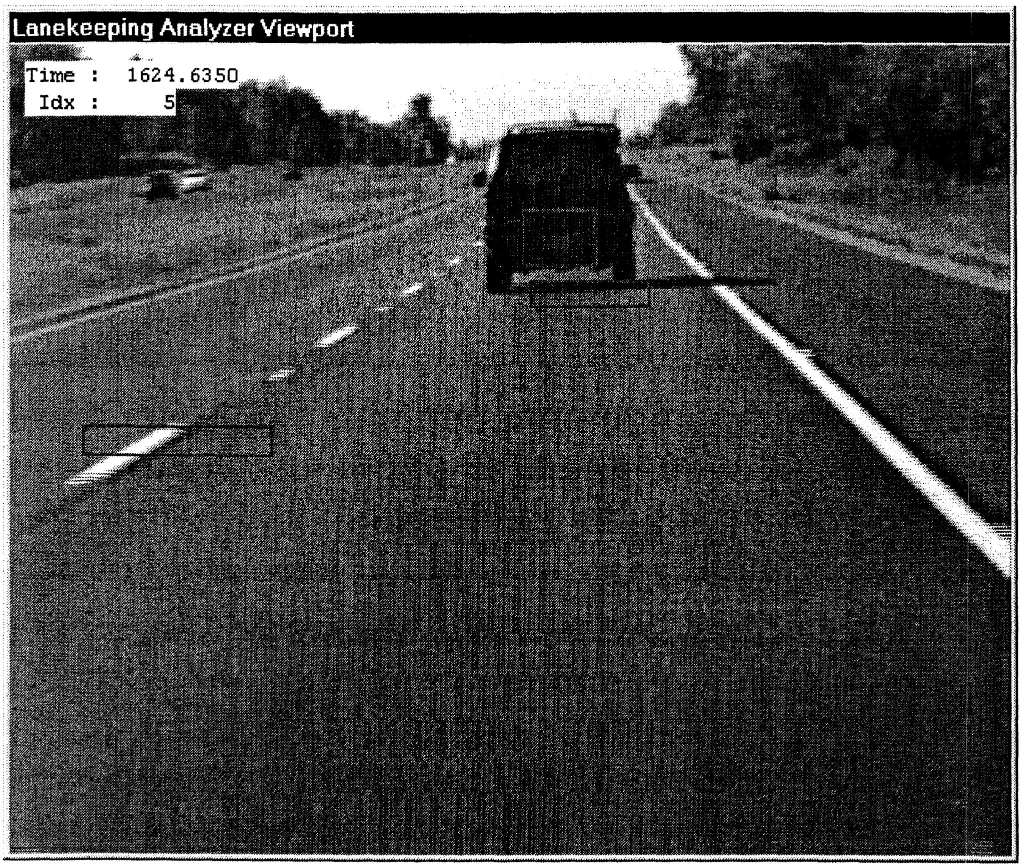


Figure 3. Setting initial conditions for the video clip analyzer: selecting the initial evaluation zones.

3.5 Lane-Finding Algorithm

The performance objective for this algorithm is to detect and track the lane boundaries of a relatively straight, smooth section of the rightmost lane of a limited-access high-speed roadway. These restrictions were considered acceptable for an adequate result within the time frame and funding levels for this project. The software package developed for this purpose of lane tracking does indeed adequately meet the stated performance objectives.

Data from previous studies of roadway boundary markings [1] provided the basis for a reasonable expectation that the luminance, or brightness, of these markings on well maintained highways would generally be statistically significant in their contrast to the average brightness of the entire roadway surface. The success of the design has born this out in fact, though the actual level of significance is quite variable. The approach taken to cope with this somewhat random significance level was to implement an adaptive or soft thresholding method to detect either the presence or lack of a roadway boundary marking. Based on experience with the algorithm, it seems that for our tests roadway markings tend to have a luminance level which averages in the near field about 3.0 standard deviations above the general luminance of the adjacent roadway. Far field thresholds tended to average about half that, around 1.8. See figure 4 for an illustration of this thresholding.

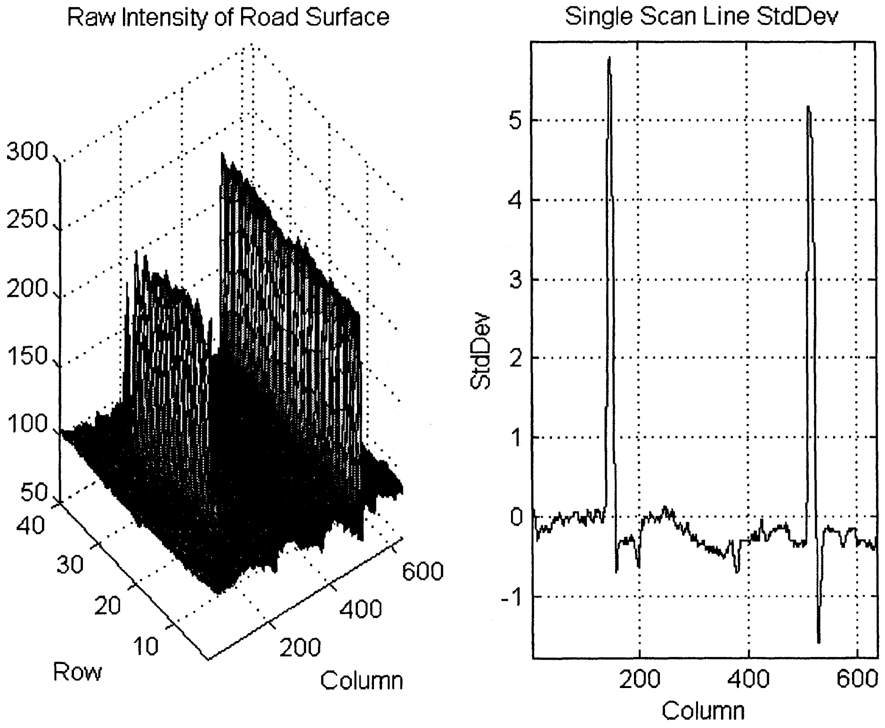


Figure 4. On the left is a surface of the raw intensity of a patch of road, while on the right is the distribution of intensities of a single near-field scan line.

Since a basic requirement is that the roadway segments of interest would be relatively straight, we chose to detect the lane markings at three distinct points, near, mid, and far, relative to the observer. Since the test protocol was confined to operation on relatively straight roadways, boundary detection at these three points then provided for connection into continuous right- and left-line segments approximating the lane boundaries. These points are detected by processing individual sets of scan lines. Each set is chosen from one of the near, mid, or far fields. If we detect a valid threshold hit for enough scan lines in a given field, then it is decided that a valid lane-boundary marking hit for that field has occurred. All the hits for that field are combined by least squares regression, and the offset term becomes the horizontal coordinate of that particular boundary point. The vertical (row) coordinate is set equal to the mean of the vertical coordinates of the various component scan lines. The near-field vertical coordinate is set during initialization, the far-field vertical coordinate is set equal to the vertical coordinate that intersects the image row established by the subject vehicle's rear-shadow boundary, while the mid-field vertical coordinate is set at the row intersections placed midway between the near and far fields. See Appendix A for a complete listing of the working code.

3.6 License-Plate-Tracking Algorithm

Designing a robust license-plate-tracking algorithm required a higher level of sophistication than in the lane tracker just described. An issue which became significant during this design was the unreliability of thresholding methods for this type of tracking task in general. It seems reasonable to suppose that the mapping of contrast gradients within an apertured evaluation region could be thresholded to detect a given object's boundaries. Although seemingly reasonable, in practice such approaches are not adequately robust. The basic difficulty with applying a thresholding method to this problem lies in the fact that there simply are not enough nonedge pixels to give good statistical significance to contrast edge levels. Clearly another approach was called for.

Accordingly, a robust adaptive template-based method of acquiring and subsequently tracking a subject vehicle's license plate was developed. Briefly, the method involves preprocessing the raw image by forming a velocity-transformed image of pixel-to-pixel luminosity changes. This secondary image will have the highest hills at locations which correspond precisely with the most strongly contrasted edges of objects in the raw image. An ideal license plate template is then moved around on the secondary-image climbing hills. When the template is overlayed on the highest set of hills which match its outline, highest correlation possible is achieved between the template and the original image of the object represented by the idealized template, in this case a license plate. With this method no thresholding is necessary.

Since the aspect ratio of a license plate is fixed at 2:1, twice as wide as it is high, an initial width condition is combined with the apriori knowledge of the plate's aspect ratio to form the template. This template is exhaustively scanned throughout the apertured evaluation region and the location of the highest correlation value is remembered. The template's width is shrunk by one pixel, and the region is rescanned. The highest correlation value for this pass is remembered as well. The template's width is now grown by one pixel and the process is repeated a third and final time. Whichever correlation value is the highest determines the new size of the template, thereby adapting to inevitable fluctuations in headway distance, and hence, to apparent target license plate size which will occur. As long as there are sufficiently high hills in the transformed image (edges of sufficient definition in the raw image), this method works very well. If the road gets too rough, the target plate will bounce out of the aperture zone and the tracker will lose its target. In this event, the operator is required to reinitialize the tracker. See figure 5 for an illustration of the template matching process. See Appendix B for a complete listing of the working code.

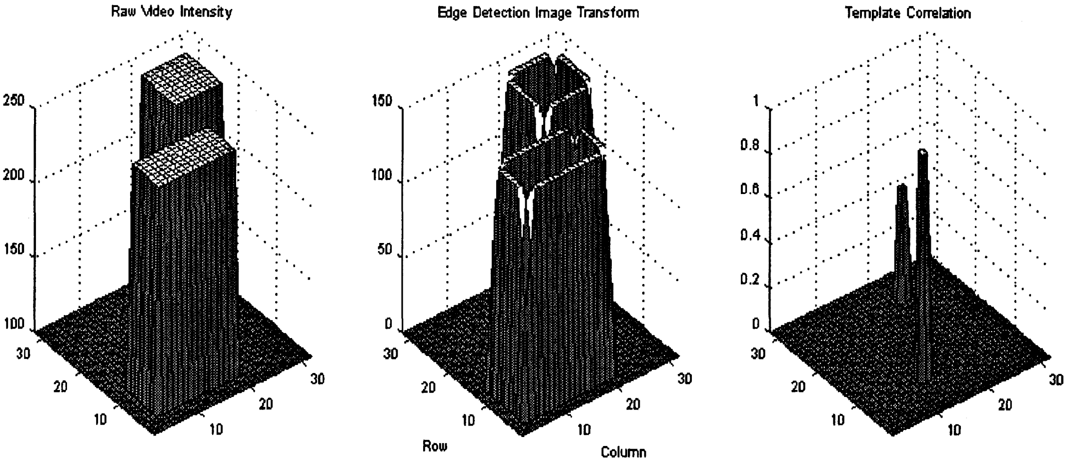


Figure 5. The process begins on the left with a surface showing image intensity. Two objects are present. One is a license plate while the other is something *not quite* a license plate. Edges are detected and correlation found between a license plate template and both edge-transformed objects. The nearer object is then identified as a license plate.

3.7 Logging the Analysis Data

The fundamental measurement of interest is the subject vehicle's lateral displacement from the local-lane center line, as a function of time. We track the subject vehicle's license plate and thereby derive our estimate of the centroid of the target's license plate. This license plate centroid becomes our surrogate for the centroid of the entire target vehicle. The center line of the subject vehicle's roadway lane is derived directly as the midpoint between the right and left far field lane boundary marker horizontal coordinates.

These two basic measurements provide us with our fundamental datum for evaluating the lateral displacement variable, y, simply as an offset value, measuring negative to the left, positive to the right. Timestamp values for each video frame are captured as well.

There is a third fundamental data channel, the invalid flag. When true, its associated data table record is considered invalid. This flag is under the direct control of the human operator of the analysis utility. If the license plate tracker ever drifts off target, the operator can flag all records associated with the low contrast conditions (normally these are cast shadow transients) as invalid. Once the transient has dispersed, the operator can reinitialize the tracker and proceed with valid analysis logging.

There are also some auxiliary data channels recorded. The most notable of these is our estimate of subject-vehicle headway. The other channels are internal software state information.

These data channels are organized into records, one for each video frame analyzed. These records are then organized into tables, indexed by their timestamp field in ascending order. The resultant data tables are organized into a data source structure within the Microsoft Access database environment. This database environment is fully ODBC compliant (Open Database Connectivity standard) and should be accessible by most current data-processing applications. See Appendix C for a complete listing of the data channels.

4.0 Field Measurement Procedure

Finding a straight, smooth, local, and adequately uninterrupted section of limited-access highway proved unexpectedly challenging. Ultimately a few suitable venues were located and a reasonable volume of subject observations was recorded on videotape.

A typical recording session could only begin after the following conditions were all present simultaneously. First, it was necessary to be located on a relatively straight piece of roadway. Second, a suitable target vehicle was required to present itself. Researchers had absolutely no control over this factor, although certain hunting and lurking techniques on their part seemed to optimize the frequency of acquiring a suitable target vehicle. Visually, the ideal target vehicle would be a red colored passenger car carrying a white license plate. This evidently is related to the operation of our CCD camera. Next in preference would be any dark-colored car with a light-colored license plate. Although any of these ideal vehicles would be rendered unsuitable if their rear body geometry caused strong horizontal lines of glare. This type of glare pattern was commonly observed to be thrown by rear bumper contours and also curved rear trunk contours. When these glare patterns were present they became a problem for the license-plate-tracker by jamming the template correlation process. The horizontal glare signal is so bright that when it correlates with a single long edge of the license plate template a higher numerical correlation value results than for the template's correlation with the actual license plate edge image. This was observed to occur rather frequently, in particular with late model cars of one of the American automakers. This shortcoming is likely fixable by specifically identifying and preventing these jamming events.

Once a suitable target had been located on a suitable section of roadway, then the relative level of success of that observation would depend on the length of time the subject would remain located in the right lane at relatively steady speed in front of our observing vehicle. The duration of such a typical observation in this study turned out to be certainly less than a minute. The possibility that the subject was aware of being observed must be admitted. It seemed that an inordinate number of subjects would take an exit without signaling or would jump out in the passing lane after a very short observation period in the right lane. The observation camera being mounted on the inside center windshield is probably visible to some subjects via their rearview mirror.

Because of the very unpredictable nature of the duration of an observational episode, the camera observations were taped continuously once the observer vehicle was cruising for targets at the venue. The tapes were then returned to the office for processing.

5.0 Example Results

The observations presented in this section are intended to be representative of the entire data set at large. The six ensembles were selected on the basis of content illustrating significant features of both the method and the data itself. Figure 1 is comprised of six histograms. The format of each of these histograms is the same. Magnitude of lateral displacement from lane center (in feet) is shown on the horizontal axis while normalized frequency is on the vertical axis. Hence, these histograms estimate the probability density function of a particular observational ensemble. This format has become a standard way of studying the data collected during this experiment.

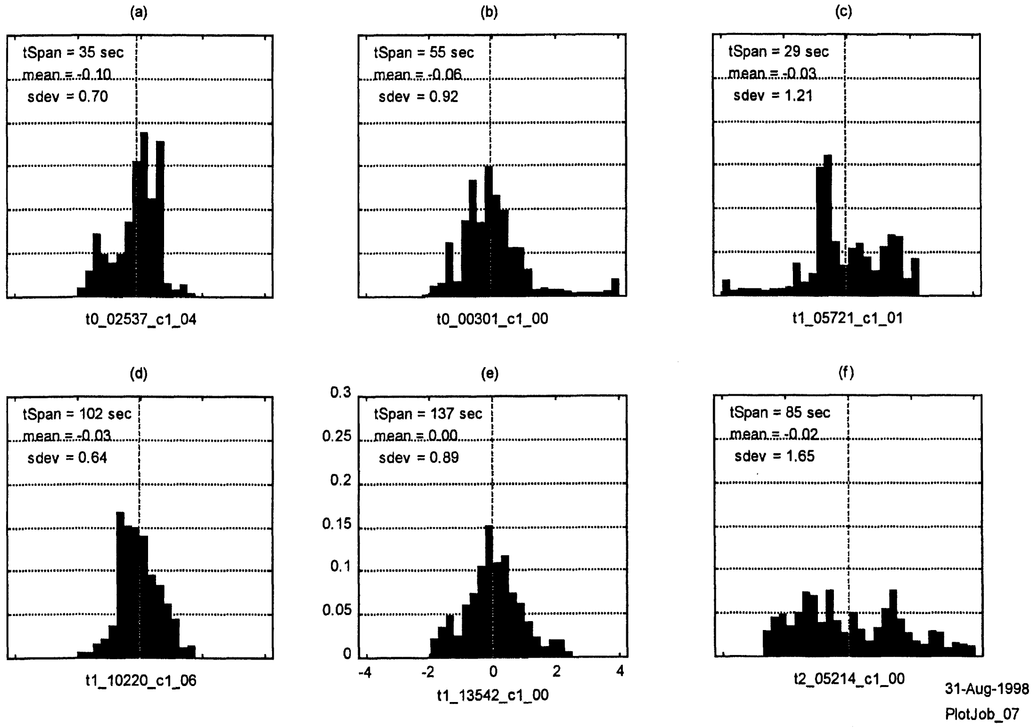


Figure 1. A representative selection of observational ensembles.

Fundamentally, all the distributions are more or less gaussian in nature although skewing is observed especially in data taken over rather brief time intervals. For example, ensemble (a) shows one lobe of response that is roughly centered about –1.75 ft, while another is centered roughly about 0. Ensemble (a) is typical of an observation during which the subject spent the majority of the time roughly centered in the lane, then moved to a position well left of lane center while approaching a slower moving vehicle from the rear in preparation for passing. The period of data gathering in this case would have ended at the time passing occurred.

Histograms (b) and (c), while arguably gaussian, exhibit skewed tails to the right and left respectively. This tailing feature of the data is directly attributable to process noise introduced by insufficiencies of the lane-boundary-tracking algorithm. The algorithm had some measurable trouble locking on to the right-lane boundary in (b) while it had similar difficulty locking on to the left-lane boundary in (c). The effects of this type of process noise were observed to be caused predominantly by the algorithm's relative weakness in locking on to the dashed-lane boundary, the left-lane boundary for observations in this study. Thus the left-handed tail shown in (c) is observed in the overall data set somewhat more often than is a right-handed tail.

Histograms (d) and (e) illustrate good quality ensembles, uncontaminated by any observable process noise. Histogram (e) clearly is composed of three components, with the preponderance of the data centered in the lane. Histogram (f) is included as an example of an atypical observation. This driver had no clear preference for any lateral lane position. Some process-induced tailing to the right is evident as well. Please see appendix D for a complete compilation of histogrammed lane keeping ensembles.

In terms of the quantitative content of the data in Appendix D, it is noted that the average duration of a measurement episode was 64.5 seconds and that the longest and shortest episodes captured in this data set were 168 seconds and 7.6 seconds, respectively. While many episodes that were shorter than 7.6 seconds simply were discarded, episodes lasting longer than 168 seconds occurred rather infrequently. The average of the mean values of lateral displacement over the 45 data samples was -0.012 feet, indicating that an offset toward the left from the lane center was typically observed.

Noting the generally gaussian nature of the lane-keeping data and an average value of the standard deviations at 0.774 feet this very limited sample of vehicles would portray a driving pattern that places the vehicle centroid within +/- 1.55 feet of the lane center, 95% of the time (i.e., at 2-sigma). Of the 19 vehicles yielding episode lengths of a minute or longer, the average mean value was 0.0324 feet, and 95% of the driving remained with +/- 0.0649 feet of the lane center.

Moreover, this very small initial batch of data has provided example profiles that help in planning refinements in the measurement method and for anticipating forms of analysis that may help in explaining driving styles and the lane-keeping performance of individuals.

6.0 Conclusions and Recommendations

This work has developed and demonstrated an initial version of a system for measuring naturalistic lane-keeping motions in a highway environment. Having taken the original concept to the point of full-scale application, a number of observations are made, some of which support recommendations for extensions of the work.

1) Good quality measurements of the lateral displacement variable can be made within a tolerance of approximately +/- 3 inches using mid-grade, off-the-shelf, video equipment.

2) Although the measurement requirement for daytime illumination and well-marked roadways is an obvious constraint of the present work, the authors believe that the vast need for naturalistic data of this kind can be largely addressed within these conditions. (That is, it is believed that normal steering behavior can be broadly and meaningfullly sampled, even if measurements are restricted to daylight hours and well-marked roads.)

3) Improvements are needed in the processing software for tracking both the lane markings and the license plate target on the subject vehicle. The objective of such improvements is primarily to increase the productivity of the method, whereby the frequency of manual intervention on the process is minimized. It is also noted that both the lane-tracking and license-tracking algorithm can occasionally lose track in a surreptitious manner, tending to corrupt the final data in ways not readily identified by the human processing monitor. In either case, the need is for improvements in robustness—the image-processing demand that invariably arises when video is taken in complex natural environments.

4) Initial measurement activity was confined to a rather sparsely trafficked freeway and to a protocol in which sustained motion of the preceding vehicle was captured only while the travel remained confined to a single highway lane. Having observed that few drivers remain within an individual lane for a very long period of time, subsequent advancements in the measurement protocol should include the procedure of changing lanes so as to observe the transitional lane-keeping activity that is exhibited just-before and just-after making a lane change. Even if we put off, for now, the need to characterize the gross lateral transient appearing during the lane-change movement, itself, the lane-keeping manifestations on either end of this transition are thought to be important in the design of driver-assistance systems.

5) Although it was straightforward to compile collected data into histograms, much additional development is needed to derive additional meaning and insight from lane-keeping data. While it is fair to observe that such further analysis is properly beyond the scope of lane-keeping measurement, per se, it is typical of such endeavors that efforts to penetrate naturalistic data often reveal additional requirements for the measurement process. Accordingly, having brought forward a rudimentary system for measurement, it seems appropriate that complementary efforts proceed for addressing both the measurement and analysis of this segment of the driving task.

References

1. Intelligent Vehicles '95 Symposium, Proceedings, New York, IEEE, 1995, p. 488-493

Appendix A

```
/*
 * file : gmLaneDetector_1.h
 * date : 01/28/97
 *
 *
 * Analyse video stream for lane boundaries
 *
 */

#if !defined(GMLANEDETECTOR_H__INCLUDED_)
#define GMLANEDETECTOR_H__INCLUDED_

/**********************
 *
 **********************/

class gmVideoStream ;

/**********************
 *
 **********************/

//#define gm12MM_LENS

/**********************
 *
 **********************/

#define gmLD_SYSTEM_STATES               7

#define gmLD_SYSTEM_HISTORYDEPTH         10

#define gmLD_HITSETSIZE_DEFAULT          5

#define gmLD_HITSETSIZE                  gmLD_HITSETSIZE_DEFAULT

/**********************
 *
 **********************/

#ifdef  gm12MM_LENS
  #define gmBASEVIEWOFFSET               00
#else
  #define gmBASEVIEWOFFSET               85
#endif

#define gmFARFIELD_MIDLINE_DEFAULT       285 - gmBASEVIEWOFFSET
#define gmMIDFIELD_MIDLINE_DEFAULT       350 - gmBASEVIEWOFFSET
#define gmNEARFIELD_MIDLINE_DEFAULT      415 - gmBASEVIEWOFFSET

/**********************
 *
 **********************/

#define gmMIDFRAME_COL_DEFAULT           320

#define gmFIELDBOUNDARY_LEFT_DEFAULT     0
#define gmFIELDBOUNDARY_RIGHT_DEFAULT    639
#define gmFIELDHEIGHT_DEFAULT            15
#define gmFARFIELD_BASELINE_DEFAULT      ( gmFARFIELD_MIDLINE_DEFAULT  - gmFIELDHEIGHT_DEFAULT / 2 )
#define gmMIDFIELD_BASELINE_DEFAULT      ( gmMIDFIELD_MIDLINE_DEFAULT  - gmFIELDHEIGHT_DEFAULT / 2 )
#define gmNEARFIELD_BASELINE_DEFAULT     ( gmNEARFIELD_MIDLINE_DEFAULT - gmFIELDHEIGHT_DEFAULT / 2 )

/**********************
 *
 **********************/

#define THRESHOLD_STEPFACTOR_DEFAULT      (double)0.250
#define MAXALLOWABLE_DETECTION_THRESHOLD  (double)10.000
#define MINALLOWABLE_DETECTION_THRESHOLD  (double)(1.000+THRESHOLD_STEPFACTOR_DEFAULT)
#define DETECTION_THRESHOLD_DEFAULT       (double)5.000

#define gmMINIMUM_TARGETSIZE_DEFAULT     3   // minimum width for a statistical target detection to be s
ignificant
```

```
#define gmMINIMUM_BOUNDARYWIDTH_DEFAULT   3   // minimum width for a statistical target detection to be s
ignificant
#define gmMAXIMUM_BOUNDARYWIDTH_DEFAULT   20  // maximum width for a statistical target detection to be s
ignificant

#define gmMINIMUM_FIELDSETHITS_DEFAULT    3
#define gmMINIMUM_FIELDSETHITS            gmMINIMUM_FIELDSETHITS_DEFAULT

#define gmDISPLAYRESULT_ENB               TRUE
#define gmDISPLAYRESULT_DISB              FALSE

#define gmTHRESHOLD_ISSOFT                TRUE
#define gmTHRESHOLD_ISHARD                FALSE

#define gmTARGET_NOTFOUND                 0
#define gmTARGETS_MULTIPLEFOUND           -1

//#define gmNEARFILED_FILTERFREQ_DEFAULT    (double)3.500
#define gmNEARFILED_FILTERFREQ_DEFAULT    (double)2.000
//#define gmMIDFILED_FILTERFREQ_DEFAULT     (double)4.000
#define gmMIDFILED_FILTERFREQ_DEFAULT     (double)2.200
//#define gmFARFILED_FILTERFREQ_DEFAULT     (double)5.500
#define gmFARFILED_FILTERFREQ_DEFAULT     (double)3.500

#define gmINITIAL_FARFIELD_LEFTLIMIT      20
#define gmINITIAL_FARFIELD_RIGHTLIMIT     gmFIELDBOUNDARY_RIGHT_DEFAULT - 20

#define gmVARIANCE_SMOOTHINGFREQ_INITIAL  (double)1e6
#define gmVARIANCE_SMOOTHINGFREQ_ONLINE   (double)2.000

///#define gmVALIDSLOPE_THRESHOLD_DEFAULT    (double)0.250
#define gmVALIDSLOPE_THRESHOLD_DEFAULT    (double)0.500
#define gmMAXVARIANCE_THRESHOLD_DEFAULT   (double)1.000

///#define gmLANELOC_TIMEOUT_DEFAULT         (double)0.100
#define gmLANELOC_TIMEOUT_DEFAULT         (double)0.200

/**********************
 *
 **********************/

typedef enum taggmFIELDTYPE
{
  gmFT_NULL      = 0x0000 ,
  gmFT_NEARLEFT  = 0x1000 ,
  gmFT_NEARRIGHT = 0x1005 ,
  gmFT_MIDLEFT   = 0x1010 ,
  gmFT_MIDRIGHT  = 0x1015 ,
  gmFT_FARLEFT   = 0x1020 ,
  gmFT_FARRIGHT  = 0x1025
} gmFIELDTYPE ;

/**********************
 *
 **********************/

typedef struct taggmLANEBOUNDARY
{
  gmBUFLOC   Left   ;   // gmBUFLOC <==> ( row,col )
  gmBUFLOC   Right  ;
  double     LaneLocTimeOutVal  ;
  double     MostRecentLeftDetectionTime ;
  double     MostRecentRightDetectionTime ;
} gmLANEBOUNDARY    ;

/**********************
 *
 **********************/

typedef struct gmtagSLOPEINFO
{
  BOOL   Valid ;
  double Value ;
} gmSLOPEINFO ;
```

```
/**********************
 *
 **********************/

typedef struct taggmLANEINFO
{
  gmSLOPEINFO     LeftNearSlope  ;
  gmSLOPEINFO     LeftFarSlope  ;
  gmSLOPEINFO     RightNearSlope ;
  gmSLOPEINFO     RightFarSlope ;
  gmLANEBOUNDARY  Near ;
  gmLANEBOUNDARY  Mid  ;
  gmLANEBOUNDARY  Far  ;
  gmLANEBOUNDARY  Shadow ;
} gmLANEINFO ;

/**********************
 *
 **********************/

typedef struct gmTHRESHINFO
{
  BOOL   Modifiable ;
  double MinimumAllowable ;
  double MaximumAllowable ;
  double StepFactor ;
  double Level ;

} gmTHRESHINFO ;

/**********************
 *
 **********************/

typedef struct taggmFIELD
{
  gmFIELDTYPE  Tag         ;
  unsigned     BaseLine    ;
  unsigned     Height      ;
  RECT         Region      ;
  gmTHRESHINFO Threshold   ;
  unsigned     TotalScans  ;
  unsigned     Detections  ;
  double       HitRatio    ;
  double       LowPassFc   ;
} gmVIDEOFIELD ;

/**********************
 *
 **********************/

typedef struct taggmVIDEOFIELDINFO
{
  gmVIDEOFIELD NearLeft ;
  gmVIDEOFIELD NearRight ;
  gmVIDEOFIELD MidLeft ;
  gmVIDEOFIELD MidRight ;
  gmVIDEOFIELD FarLeft ;
  gmVIDEOFIELD FarRight ;
  gmVIDEOFIELD ShadowLeft ;
  gmVIDEOFIELD ShadowRight ;

} gmVIDEOFIELDINFO ;

/**********************
 *
 **********************/

typedef struct taggmSYSTEM_STATE
{
  double   NearFieldThreshold ;
  double   MidFieldThreshold ;
  double   FarFieldThreshold ;
```

```
  double   ShadowFieldThreshold ;

  double   LeftNearSlope ;
  double   LeftFarSlope ;
  double   RightNearSlope ;
  double   RightFarSlope ;

} gmSYSTEM_STATE ;

/*********************
 *
 *********************/

typedef struct taggmSTATEINFO
{

  unsigned           HistoryDepth ;
  double             VarianceSmoothingFreq ;
  gmSYSTEM_STATE     RollAvg ;
  gmSYSTEM_STATE     DerivativeHistoryVariance ;
  gmSYSTEM_STATE     SmoothedDerivativeHistoryVariance ;
  gmSYSTEM_STATE * * History ;
  gmSYSTEM_STATE * * DerivativeHistory ;

} gmSTATEINFO ;

/*********************
 *
 *********************/

typedef struct gmtagTIMEINFO
{

  BOOL   Initialized ;
  double InitialFrameStamp ;
  double CurrentFrameStamp ;
  double ThisDelta ;
  double Elapsed ;

} gmTIMEINFO ;

/*********************
 *
 *********************/

typedef struct taggmSYSINFO
{
  BOOL              LaneLockedOn ;
  BOOL              LaneAnchored ;
  BOOL              NearLaneAcquired ;
  BOOL              MidLaneAcquired ;
  BOOL              FarLaneAcquired ;
  BOOL              FarSlopesEnabled ;

  gmTIMEINFO        Time ;

  gmLANEINFO        PreviousLane ;
  gmLANEINFO        CurrentLane ;

  gmVIDEOFIELDINFO  Field ;
  gmSTATEINFO       State ;

} gmSYSINFO ;

/*********************
 *
 *********************/

typedef struct taggmSTATISTICS
{
  double Sigma    ;
  double Rms      ;
  double Mean     ;
  double Variance ;
```

```
  double StdDev   ;

} gmSTATISTICS ;


/**********************
 *
 **********************/

class gmBuffer ;
class gmViewPort ;
class gmKlugeDetector ;
class gmDataLogger ;

/**********************
 *
 **********************/

class gmLaneDetector
{

  typedef struct taggmHITINFO
  {
    unsigned   Center ;
    unsigned   RowNo  ;
  } gmHITINFO ;


  typedef struct taggmHITSET
  {
    unsigned    Size     ;
    gmHITINFO * Element  ;
  } gmHITSET ;


private :

  BOOL               TestEnable ;

  BOOL               DumpEnable ;

  unsigned           LaneLockCount ;

  gmSYSINFO        * SystemInfo ;

  gmVideoStream    * VideoStream ;

  gmViewPort       * ViewPort ;

  gmKlugeDetector  * LicencePlateDetector ;

  gmBuffer         * VideoFrameBuffer ;
  BYTE             * VideoFrameImageBaseAdx ;

  gmBuffer         * RowImageBuf ;
  gmBuffer         * RowDistributionBuf ;
  BYTE             * RowHitBuf ;

  unsigned           FrameWidth ;
  unsigned           FrameHeight ;
  DWORD              FrameArea ;
  unsigned           FrameVertCenter  ;
  unsigned           FrameHorizCenter ;

  unsigned           FrameNo ;
  unsigned           FramesProcessed ;

  unsigned           HitSetCapacity ;
  gmHITSET           HitSet ;

  unsigned           MinimumDetectionWidth ;
  unsigned           MaximumDetectionWidth ;

  unsigned           LeftPointShots  ;
```

```
  unsigned           RightPointShots ;

  unsigned           FarLaneBias ;
  unsigned           FarLaneOffset ;
  unsigned           FarLaneCenter ;
  unsigned           FarLaneWidth ;

  void               CreateDataLoggingTable   ( ) ;
  void               ClearRowHitBuf           ( unsigned LeftSearchBound , unsigned RightSearchBound ) ;
  void               FillValRowHitBuf         ( BYTE Val , unsigned LeftSearchBound , unsigned RightSearc
hBound ) ;
  void               InitSystemInfo           ( unsigned Depth ) ;
  double             SlopeCalc                ( gmBUFLOC * p0 , gmBUFLOC * p1 ) ;
  unsigned           PointShoot               ( gmBUFLOC * Ref , double Slope , unsigned Row ) ;
  double             RollingAverage           ( double * Data , unsigned n ) ;
  double             LowPassIIR               ( double x , double yPrev , double CutOffFreqHz ) ;
  void               StatCalc                 ( gmSTATISTICS * Stats , double * Data , unsigned n ) ;
  void               StateDerivativeCalc      ( gmSYSTEM_STATE * Derivative , gmSYSTEM_STATE * Newer , gm
SYSTEM_STATE * Older ) ;
  void               UpdateTimeInfo           () ;
  void               UpdateStateInfo          () ;
  void               UpdateSystemInfo         () ;
  void               UpdateLaneTrackingStatus () ;
  void               AdaptParameters          () ;

public :

  gmLaneDetector     ( gmVideoStream * VideoStream , gmKlugeDetector * PlateDetector = NULL , BOOL TestEn
b = FALSE ) ;
  ~gmLaneDetector    ( ) ;

  void               SmoothNewLaneLocation    ( double CutOffFreqHz ) ;
  gmBUFLOC           RegressHitSet            ( gmHITSET * HitSet ) ;
  gmBUFLOC           RegressHitSet            ( ) ;

  void               DisplayResult            ( ) ;

  unsigned           ProcessRow               ( gmVIDEOFIELD * Field , unsigned   Offset ) ;

  unsigned           ScanField                ( gmVIDEOFIELD * Field ) ;

  void               ProcessShadowField       ( );
  void               ProcessFarField          ( );
  void               ProcessMidField          ( );
  void               ProcessNearField         ( );
  void               ProcessFields            ( );

  gmLANEINFO       * ProcessFrame             ( BOOL DisplayResult , BOOL DebugEnb = gmDEBUG_DISB ) ;
  void               TestProcess              ( ) ;

  unsigned           GetFarLaneBias()         { return FarLaneBias   ; } ;
  unsigned           GetFarLaneCenter()       { return FarLaneCenter ; } ;
  unsigned           GetFarLaneOffset()       { return FarLaneOffset ; } ;
  unsigned           GetFarLaneWidth()        { return FarLaneWidth  ; } ;

  unsigned           GetLeftNearRow()         { return SystemInfo->CurrentLane.Near.Left.Row  ; } ;
  unsigned           GetLeftNearCol()         { return SystemInfo->CurrentLane.Near.Left.Col  ; } ;
  unsigned           GetRightNearRow()        { return SystemInfo->CurrentLane.Near.Right.Row ; } ;
  unsigned           GetRightNearCol()        { return SystemInfo->CurrentLane.Near.Right.Col ; } ;

  unsigned           GetLeftMidRow()          { return SystemInfo->CurrentLane.Mid.Left.Row   ; } ;
  unsigned           GetLeftMidCol()          { return SystemInfo->CurrentLane.Mid.Left.Col   ; } ;
  unsigned           GetRightMidRow()         { return SystemInfo->CurrentLane.Mid.Right.Row  ; } ;
  unsigned           GetRightMidCol()         { return SystemInfo->CurrentLane.Mid.Right.Col  ; } ;

  unsigned           GetLeftFarRow()          { return SystemInfo->CurrentLane.Far.Left.Row   ; } ;
  unsigned           GetLeftFarCol()          { return SystemInfo->CurrentLane.Far.Left.Col   ; } ;
  unsigned           GetRightFarRow()         { return SystemInfo->CurrentLane.Far.Right.Row  ; } ;
  unsigned           GetRightFarCol()         { return SystemInfo->CurrentLane.Far.Right.Col  ; } ;

} ;

/**********************
 *
```

```
 **********************/

#endif
```

```
/*
 * file : gmLaneDetector_1.cpp
 * date : 01/28/97
 *
 *
 * Analyse video frame for lane boundaries
 *
 */

/**********************
 *
 **********************/

//#define gmTRACE_ENB

/**********************
 *
 **********************/

#include "MilCons.h"

/**********************
 *
 **********************/

static gmTIMEINFO         NullTimeInfo        = {FALSE,0,0,0,0};

static gmSYSTEM_STATE     NullSystemState     = {0,0,0,0,0,0,0};

static gmLANEBOUNDARY     NullLaneBoundary    = {{0,0},{0,0},0,0,0};

static gmTHRESHINFO       NullThreshInfo      = {0,0,0,0} ;

static gmVIDEOFIELD       NullField           = {gmFT_NULL,0,0,{0,0,0,0},{0,0,0,0,0},0,0} ;

static gmVIDEOFIELD       InitialNearLeftField  = {gmFT_NEARLEFT ,gmNEARFIELD_BASELINE_DEFAULT,gmFIELDH
EIGHT_DEFAULT,{gmFIELDBOUNDARY_LEFT_DEFAULT,gmNEARFIELD_BASELINE_DEFAULT,gmMIDFRAME_COL_DEFAULT       ,gm
NEARFIELD_BASELINE_DEFAULT+gmFIELDHEIGHT_DEFAULT},{FALSE,MINALLOWABLE_DETECTION_THRESHOLD,MAXALLOWABLE_DE
TECTION_THRESHOLD,THRESHOLD_STEPFACTOR_DEFAULT,DETECTION_THRESHOLD_DEFAULT},0,0,0,gmNEARFILED_FILTERFREQ_
DEFAULT} ;
static gmVIDEOFIELD       InitialNearRightField = {gmFT_NEARRIGHT,gmNEARFIELD_BASELINE_DEFAULT,gmFIELDH
EIGHT_DEFAULT,{gmMIDFRAME_COL_DEFAULT       ,gmNEARFIELD_BASELINE_DEFAULT,gmFIELDBOUNDARY_RIGHT_DEFAULT,gm
NEARFIELD_BASELINE_DEFAULT+gmFIELDHEIGHT_DEFAULT},{TRUE ,MINALLOWABLE_DETECTION_THRESHOLD,MAXALLOWABLE_DE
TECTION_THRESHOLD,THRESHOLD_STEPFACTOR_DEFAULT,DETECTION_THRESHOLD_DEFAULT},0,0,0,gmNEARFILED_FILTERFREQ_
DEFAULT} ;
static gmVIDEOFIELD       InitialMidLeftField   = {gmFT_MIDLEFT  ,gmMIDFIELD_BASELINE_DEFAULT ,gmFIELDH
EIGHT_DEFAULT,{gmFIELDBOUNDARY_LEFT_DEFAULT,gmMIDFIELD_BASELINE_DEFAULT ,gmMIDFRAME_COL_DEFAULT       ,gm
MIDFIELD_BASELINE_DEFAULT +gmFIELDHEIGHT_DEFAULT},{FALSE,MINALLOWABLE_DETECTION_THRESHOLD,MAXALLOWABLE_DE
TECTION_THRESHOLD,THRESHOLD_STEPFACTOR_DEFAULT,DETECTION_THRESHOLD_DEFAULT},0,0,0,gmMIDFILED_FILTERFREQ_D
EFAULT } ;
static gmVIDEOFIELD       InitialMidRightField  = {gmFT_MIDRIGHT ,gmMIDFIELD_BASELINE_DEFAULT ,gmFIELDH
EIGHT_DEFAULT,{gmMIDFRAME_COL_DEFAULT       ,gmMIDFIELD_BASELINE_DEFAULT ,gmFIELDBOUNDARY_RIGHT_DEFAULT,gm
MIDFIELD_BASELINE_DEFAULT +gmFIELDHEIGHT_DEFAULT},{TRUE ,MINALLOWABLE_DETECTION_THRESHOLD,MAXALLOWABLE_DE
TECTION_THRESHOLD,THRESHOLD_STEPFACTOR_DEFAULT,DETECTION_THRESHOLD_DEFAULT},0,0,0,gmMIDFILED_FILTERFREQ_D
EFAULT } ;
static gmVIDEOFIELD       InitialFarLeftField   = {gmFT_FARLEFT  ,gmFARFIELD_BASELINE_DEFAULT ,gmFIELDH
EIGHT_DEFAULT,{gmFIELDBOUNDARY_LEFT_DEFAULT,gmFARFIELD_BASELINE_DEFAULT ,gmMIDFRAME_COL_DEFAULT       ,gm
FARFIELD_BASELINE_DEFAULT +gmFIELDHEIGHT_DEFAULT},{FALSE,MINALLOWABLE_DETECTION_THRESHOLD,MAXALLOWABLE_DE
TECTION_THRESHOLD,THRESHOLD_STEPFACTOR_DEFAULT,DETECTION_THRESHOLD_DEFAULT},0,0,0,gmFARFILED_FILTERFREQ_D
EFAULT } ;
static gmVIDEOFIELD       InitialFarRightField  = {gmFT_FARRIGHT ,gmFARFIELD_BASELINE_DEFAULT ,gmFIELDH
EIGHT_DEFAULT,{gmMIDFRAME_COL_DEFAULT       ,gmFARFIELD_BASELINE_DEFAULT ,gmFIELDBOUNDARY_RIGHT_DEFAULT,gm
FARFIELD_BASELINE_DEFAULT +gmFIELDHEIGHT_DEFAULT},{FALSE,MINALLOWABLE_DETECTION_THRESHOLD,MAXALLOWABLE_DE
TECTION_THRESHOLD,THRESHOLD_STEPFACTOR_DEFAULT,DETECTION_THRESHOLD_DEFAULT},0,0,0,gmFARFILED_FILTERFREQ_D
EFAULT } ;

static gmVIDEOFIELDINFO   NullFieldInfo       =  {
                                                  {gmFT_NULL,0,0,{0,0,0,0},{0,0,0,0,0},0,0,0,0} ,
                                                  {gmFT_NULL,0,0,{0,0,0,0},{0,0,0,0,0},0,0,0,0} ,
                                                  {gmFT_NULL,0,0,{0,0,0,0},{0,0,0,0,0},0,0,0,0} ,
                                                  {gmFT_NULL,0,0,{0,0,0,0},{0,0,0,0,0},0,0,0,0} ,
                                                  {gmFT_NULL,0,0,{0,0,0,0},{0,0,0,0,0},0,0,0,0} ,
                                                  {gmFT_NULL,0,0,{0,0,0,0},{0,0,0,0,0},0,0,0,0}
                                                 };
```

```
static gmVIDEOFIELDINFO     InitialFieldInfo       =  {
                                                         {gmFT_NEARLEFT ,gmNEARFIELD_BASELINE_DEFAULT,gmFIEL
DHEIGHT_DEFAULT,{gmFIELDBOUNDARY_LEFT_DEFAULT,gmNEARFIELD_BASELINE_DEFAULT,gmMIDFRAME_COL_DEFAULT        ,
gmNEARFIELD_BASELINE_DEFAULT+gmFIELDHEIGHT_DEFAULT},{FALSE,MINALLOWABLE_DETECTION_THRESHOLD,MAXALLOWABLE_
DETECTION_THRESHOLD,THRESHOLD_STEPFACTOR_DEFAULT,DETECTION_THRESHOLD_DEFAULT},0,0,0,gmNEARFILED_FILTERFRE
Q_DEFAULT} ,
                                                         {gmFT_NEARRIGHT,gmNEARFIELD_BASELINE_DEFAULT,gmFIEL
DHEIGHT_DEFAULT,{gmMIDFRAME_COL_DEFAULT      ,gmNEARFIELD_BASELINE_DEFAULT,gmFIELDBOUNDARY_RIGHT_DEFAULT,
gmNEARFIELD_BASELINE_DEFAULT+gmFIELDHEIGHT_DEFAULT},{TRUE ,MINALLOWABLE_DETECTION_THRESHOLD,MAXALLOWABLE_
DETECTION_THRESHOLD,THRESHOLD_STEPFACTOR_DEFAULT,DETECTION_THRESHOLD_DEFAULT},0,0,0,gmNEARFILED_FILTERFRE
Q_DEFAULT} ,
                                                         {gmFT_MIDLEFT  ,gmMIDFIELD_BASELINE_DEFAULT ,gmFIEL
DHEIGHT_DEFAULT,{gmFIELDBOUNDARY_LEFT_DEFAULT,gmMIDFIELD_BASELINE_DEFAULT ,gmMIDFRAME_COL_DEFAULT        ,
gmMIDFIELD_BASELINE_DEFAULT +gmFIELDHEIGHT_DEFAULT},{FALSE,MINALLOWABLE_DETECTION_THRESHOLD,MAXALLOWABLE_
DETECTION_THRESHOLD,THRESHOLD_STEPFACTOR_DEFAULT,DETECTION_THRESHOLD_DEFAULT},0,0,0,gmMIDFILED_FILTERFREQ
_DEFAULT } ,
                                                         {gmFT_MIDRIGHT ,gmMIDFIELD_BASELINE_DEFAULT ,gmFIEL
DHEIGHT_DEFAULT,{gmMIDFRAME_COL_DEFAULT      ,gmMIDFIELD_BASELINE_DEFAULT ,gmFIELDBOUNDARY_RIGHT_DEFAULT,
gmMIDFIELD_BASELINE_DEFAULT +gmFIELDHEIGHT_DEFAULT},{TRUE ,MINALLOWABLE_DETECTION_THRESHOLD,MAXALLOWABLE_
DETECTION_THRESHOLD,THRESHOLD_STEPFACTOR_DEFAULT,DETECTION_THRESHOLD_DEFAULT},0,0,0,gmMIDFILED_FILTERFREQ
_DEFAULT } ,
                                                         {gmFT_FARLEFT  ,gmFARFIELD_BASELINE_DEFAULT ,gmFIEL
DHEIGHT_DEFAULT,{gmFIELDBOUNDARY_LEFT_DEFAULT,gmFARFIELD_BASELINE_DEFAULT ,gmMIDFRAME_COL_DEFAULT        ,
gmFARFIELD_BASELINE_DEFAULT +gmFIELDHEIGHT_DEFAULT},{FALSE,MINALLOWABLE_DETECTION_THRESHOLD,MAXALLOWABLE_
DETECTION_THRESHOLD,THRESHOLD_STEPFACTOR_DEFAULT,DETECTION_THRESHOLD_DEFAULT},0,0,0,gmFARFILED_FILTERFREQ
_DEFAULT } ,
                                                         {gmFT_FARRIGHT ,gmFARFIELD_BASELINE_DEFAULT ,gmFIEL
DHEIGHT_DEFAULT,{gmMIDFRAME_COL_DEFAULT      ,gmFARFIELD_BASELINE_DEFAULT ,gmFIELDBOUNDARY_RIGHT_DEFAULT,
gmFARFIELD_BASELINE_DEFAULT +gmFIELDHEIGHT_DEFAULT},{FALSE,MINALLOWABLE_DETECTION_THRESHOLD,MAXALLOWABLE_
DETECTION_THRESHOLD,THRESHOLD_STEPFACTOR_DEFAULT,DETECTION_THRESHOLD_DEFAULT},0,0,0,gmFARFILED_FILTERFREQ
_DEFAULT }
                                                        };

static gmLANEINFO           NullLaneInfo           =  {{0,0} ,
                                                         {0,0} ,
                                                         {0,0} ,
                                                         {0,0} ,
                                                         {{0,0},{0,0},0,0,0} ,
                                                         {{0,0},{0,0},0,0,0} ,
                                                         {{0,0},{0,0},0,0,0}
                                                        };

static gmLANEINFO           InitialLaneInfo        =  {{0,0} ,
                                                         {0,0} ,
                                                         {0,0} ,
                                                         {0,0} ,
                                                         {{320,320},{320,320},gmLANELOC_TIMEOUT_DEFAULT,0,0}
 ,
                                                         {{320,320},{320,320},gmLANELOC_TIMEOUT_DEFAULT,0,0}
 ,
                                                         {{320,320},{320,320},gmLANELOC_TIMEOUT_DEFAULT,0,0}
                                                        };

/***************************************************************************

                    Following are this class's Private Methods

 ***************************************************************************/

/*********************
 *
 *********************/

void gmLaneDetector::ClearRowHitBuf ( unsigned LeftBound , unsigned RightBound )
{
  gmTRACE_MAC ( "gmLaneDetector::ClearRowHitBuf() : Entering\n" ) ;
  unsigned n = RightBound - LeftBound + 1 ;
  memset( ( void * ) ( RowHitBuf + LeftBound ) , 0x00 , n ) ;
  gmTRACE_MAC ( "gmLaneDetector::ClearRowHitBuf() : Entering\n" ) ;
}

/*********************
 *
```

```
 **********************/

void gmLaneDetector::FillValRowHitBuf ( BYTE Val , unsigned LeftSearchBound , unsigned RightSearchBound )
{
  gmTRACE_MAC ( "gmLaneDetector::FillValRowHitBuf() : Entering\n" ) ;
  unsigned n = RightSearchBound - LeftSearchBound + 1 ;
  memset( ( void * ) ( RowHitBuf + LeftSearchBound ) , Val , n ) ;
  gmTRACE_MAC ( "gmLaneDetector::FillValRowHitBuf() : Entering\n" ) ;
}

/**********************
 *
 **********************/

void gmLaneDetector::InitSystemInfo ( unsigned Depth )
{
  gmTRACE_MAC ( "gmLaneDetector::InitSystemInfo : Entering\n" ) ;

  unsigned i ;


  SystemInfo = new gmSYSINFO ;

  SystemInfo->LaneLockedOn      = FALSE ;
  SystemInfo->LaneAnchored      = FALSE ;
  SystemInfo->NearLaneAcquired  = FALSE ;
  SystemInfo->MidLaneAcquired   = FALSE ;
  SystemInfo->FarLaneAcquired   = FALSE ;
  SystemInfo->FarSlopesEnabled  = FALSE ;

  SystemInfo->Time              = NullTimeInfo ;

  SystemInfo->PreviousLane      = InitialLaneInfo ;
  SystemInfo->CurrentLane       = InitialLaneInfo ;

  SystemInfo->Field             = InitialFieldInfo ;


  SystemInfo->State.History           = new gmSYSTEM_STATE * [ Depth ] ;
  SystemInfo->State.DerivativeHistory = new gmSYSTEM_STATE * [ Depth ] ;


  for ( i = 0 ; i < Depth ; i ++ )
  {

    SystemInfo->State.History[i]              = new gmSYSTEM_STATE ;
    SystemInfo->State.DerivativeHistory[i]    = new gmSYSTEM_STATE ;

    * SystemInfo->State.History[i]            = NullSystemState ;
    * SystemInfo->State.DerivativeHistory[i]  = NullSystemState ;

  }

  SystemInfo->State.HistoryDepth          = Depth ;
  SystemInfo->State.VarianceSmoothingFreq = gmVARIANCE_SMOOTHINGFREQ_INITIAL ;


  SystemInfo->State.RollAvg                            = NullSystemState ;
  SystemInfo->State.DerivativeHistoryVariance          = NullSystemState ;
  SystemInfo->State.SmoothedDerivativeHistoryVariance = NullSystemState ;


  gmTRACE_MAC ( "gmLaneDetector::InitSystemInfo : Exiting\n" ) ;
}

/**********************
 *
 **********************/

double gmLaneDetector::SlopeCalc ( gmBUFLOC * p0 , gmBUFLOC * p1 )
{
  gmTRACE_MAC ( "gmLaneDetector::CalcBoundrySlope : Entering\n" ) ;

  double  Rise , Run , Slope ;
```

```
  Rise   = ( double ) p1->Row - p0->Row ;
  Run    = ( double ) p1->Col - p0->Col ;

  if ( fabs ( Run ) < 1e-10 )
    Slope = 1e10 ;
  else
    Slope = Rise / Run ;

  return Slope ;

  gmTRACE_MAC ( "gmLaneDetector::CalcBoundrySlope : Exiting\n" ) ;
}

/*********************
 *
 *********************/

unsigned gmLaneDetector::PointShoot ( gmBUFLOC * Ref , double Slope , unsigned Row )
{
  gmTRACE_MAC ( "gmLaneDetector::PointShoot : Entering\n" ) ;


  int Col = ( int ) ( ( ( ( double ) Row - ( double ) Ref->Row ) / Slope ) + Ref->Col ) ;

  if ( Col < 0 )
    Col = 0 ;

  return ( unsigned ) Col ;

  gmTRACE_MAC ( "gmLaneDetector::PointShoot : Exiting\n" ) ;
}

/*********************
 *
 *********************/

double gmLaneDetector::RollingAverage ( double * Data , unsigned n )
{
  gmTRACE_MAC ( "gmLaneDetector::RollingAverage() : Entering\n" ) ;

  double Sigma = 0 ;
  unsigned i ;

  if ( ! n )
    return 0 ;

  for ( i = 0 ; i < n ; i ++ )
    Sigma += Data[i] ;

  gmTRACE_MAC ( "gmLaneDetector::RollingAverage() : Exiting\n" ) ;

  return Sigma / (( double ) n ) ;
}

/*********************
 *
 *********************/

double gmLaneDetector::LowPassIIR ( double x , double yPrev , double CutOffFreqHz )
{
  gmTRACE_MAC ( "gmLaneDetector:: : Entering\n" ) ;

  double T     = 6.283185 * CutOffFreqHz * SystemInfo->Time.ThisDelta ;
  double Alpha = T/(1.00+T) ;


  double y = Alpha * x + ( 1.00 - Alpha ) * yPrev ;


  gmTRACE_MAC ( "gmLaneDetector:: : Exiting\n" ) ;

  return y ;
```

```
}

/**********************
 *
 **********************/

void gmLaneDetector::StatCalc ( gmSTATISTICS * Stats , double * Data , unsigned n )
{
  gmTRACE_MAC ( "gmLaneDetector::StatCalc() : Entering\n" ) ;


  double y0 , y1 ;
  unsigned i ;

  for ( i = 0 , y0=y1=0 ; i < n ; i ++ )
  {
    y0 += Data[i] ;
    y1 += Data[i] * Data[i] ;
  }


  Stats->Sigma    = y0 ;
  Stats->Rms      = sqrt ( y1 ) ;
  Stats->Mean     = y0 / n ;
  Stats->Variance = ( n * y1 - y0 * y0 ) / ( n * ( n - 1 ) ) ;
  Stats->StdDev   = sqrt ( Stats->Variance ) ;


  gmTRACE_MAC ( "gmLaneDetector::StatCalc() : Exiting\n" ) ;
}

/**********************
 *
 **********************/

void gmLaneDetector::StateDerivativeCalc ( gmSYSTEM_STATE * Derivative , gmSYSTEM_STATE * Newer , gmSYSTE
M_STATE * Older )
{
  gmTRACE_MAC ( "gmLaneDetector::StateDerivativeCalc() : Entering\n" ) ;

  double ThisDelta = SystemInfo->Time.ThisDelta ;

  Derivative->NearFieldThreshold = ( Newer->NearFieldThreshold - Older->NearFieldThreshold ) / ThisDelta
;
  Derivative->MidFieldThreshold  = ( Newer->MidFieldThreshold  - Older->MidFieldThreshold  ) / ThisDelta
;
  Derivative->FarFieldThreshold  = ( Newer->FarFieldThreshold  - Older->FarFieldThreshold  ) / ThisDelta
;
  Derivative->LeftNearSlope      = ( Newer->LeftNearSlope      - Older->LeftNearSlope      ) / ThisDelta
;
  Derivative->LeftFarSlope       = ( Newer->LeftFarSlope       - Older->LeftFarSlope       ) / ThisDelta
;
  Derivative->RightNearSlope     = ( Newer->RightNearSlope     - Older->RightNearSlope     ) / ThisDelta
;
  Derivative->RightFarSlope      = ( Newer->RightFarSlope      - Older->RightFarSlope      ) / ThisDelta
;

  gmTRACE_MAC ( "gmLaneDetector::StateDerivativeCalc() : Exiting\n" ) ;
}

/**********************
 *
 **********************/

void gmLaneDetector::UpdateTimeInfo ( )
{
  gmTRACE_MAC ( "gmLaneDetector::UpdateTimeInfo() : Entering\n" ) ;


  gmTIMEINFO  * p             = & SystemInfo->Time ;
  double        ThisTimeStamp = VideoFrameBuffer->GetTimeStamp() ;


  if ( ! p->Initialized )
```

```
  {
    p->InitialFrameStamp = ThisTimeStamp ;
    p->CurrentFrameStamp = ThisTimeStamp ;
    p->Initialized       = TRUE ;
  }

  p->ThisDelta         = ThisTimeStamp - p->CurrentFrameStamp ;
  p->CurrentFrameStamp = ThisTimeStamp ;

  if ( p->ThisDelta < 0.00 )
  {
    printf ( "% 5d : Negative Time Step : %12.7f\n" , FrameNo , p->ThisDelta ) ;
    p->ThisDelta *= -1.00 ;
  }

  p->Elapsed += p->ThisDelta ;


  gmTRACE_MAC ( "gmLaneDetector::UpdateTimeInfo() : Exiting\n" ) ;
}

/**********************
 *
 **********************/

void gmLaneDetector::UpdateStateInfo ( )
{
  gmTRACE_MAC ( "gmLaneDetector::UpdateStateInfo() : Entering\n" ) ;

  static gmSTATISTICS  ScratchStats ;
  static double ScratchArea [gmLD_SYSTEM_STATES][gmLD_SYSTEM_HISTORYDEPTH] ;


  SystemInfo->CurrentLane.LeftNearSlope.Value  = SlopeCalc ( & SystemInfo->CurrentLane.Mid.Left  , & Syst
emInfo->CurrentLane.Near.Left  ) ;
  SystemInfo->CurrentLane.RightNearSlope.Value = SlopeCalc ( & SystemInfo->CurrentLane.Mid.Right , & Syst
emInfo->CurrentLane.Near.Right ) ;

  if ( FramesProcessed > ( 2 * gmLD_SYSTEM_HISTORYDEPTH ) )
  {
    SystemInfo->FarSlopesEnabled = TRUE ;
    SystemInfo->CurrentLane.LeftFarSlope.Value   = SlopeCalc ( & SystemInfo->CurrentLane.Far.Left  , & Sy
stemInfo->CurrentLane.Mid.Left   ) ;
    SystemInfo->CurrentLane.RightFarSlope.Value  = SlopeCalc ( & SystemInfo->CurrentLane.Far.Right , & Sy
stemInfo->CurrentLane.Mid.Right  ) ;
  }


  gmSYSTEM_STATE * * p  = SystemInfo->State.History , * q ;
  gmSYSTEM_STATE * * p1 = SystemInfo->State.DerivativeHistory , * q1 ;
  unsigned i ;


  //
  // here we push the history stacks
  // down one position .
  //

  q  = p  [ gmLD_SYSTEM_HISTORYDEPTH - 1 ] ;
  q1 = p1 [ gmLD_SYSTEM_HISTORYDEPTH - 1 ] ;

  for ( i = gmLD_SYSTEM_HISTORYDEPTH - 1 ; i ; i -- )
  {
    p[i]  = p[i-1] ;
    p1[i] = p1[i-1] ;
  }

  p[0]  = q ;
  p1[0] = q1 ;


  //
  // here we push the current state vector
  // into the top location of the history
```

```
  // stack .
  //

  p[0]->NearFieldThreshold = SystemInfo->Field.NearRight.Threshold.Level  ;
  p[0]->MidFieldThreshold  = SystemInfo->Field.MidRight.Threshold.Level   ;
  p[0]->FarFieldThreshold  = SystemInfo->Field.FarRight.Threshold.Level   ;
  p[0]->LeftNearSlope      = SystemInfo->CurrentLane.LeftNearSlope.Value  ;
  p[0]->LeftFarSlope       = SystemInfo->CurrentLane.LeftFarSlope.Value   ;
  p[0]->RightNearSlope     = SystemInfo->CurrentLane.RightNearSlope.Value ;
  p[0]->RightFarSlope      = SystemInfo->CurrentLane.RightFarSlope.Value  ;


  //
  // now , if we've got a time delta ,
  // ( sometimes we won't , like on the
  // first frame , and probably other times
  // too , if Murphy still lives in this
  // universe ... ) we calculate dx/dt's .
  //

  if ( SystemInfo->Time.ThisDelta )
  {
    StateDerivativeCalc ( p1[0] , p[0] , p[1] ) ;
    SystemInfo->PreviousLane = SystemInfo->CurrentLane ;
  }
  else
    * p1[0] = * p1[1] ;


  //
  // now , if we don't have a full queue
  // of history data , then , don't waste
  // any more time here .
  //

  if ( FramesProcessed < gmLD_SYSTEM_HISTORYDEPTH )
    return ;

  //
  // once we've got complete trajectory information
  // let's summarize it in our statistical state
  // vectors .
  //
  // here we load up the scratch area with the
  // state history information
  //

  for ( i = 0 ; i < gmLD_SYSTEM_HISTORYDEPTH ; i ++ )
  {
    ScratchArea[0][i] = p[i]->NearFieldThreshold ;
    ScratchArea[1][i] = p[i]->MidFieldThreshold ;
    ScratchArea[2][i] = p[i]->FarFieldThreshold ;
    ScratchArea[3][i] = p[i]->LeftNearSlope ;
    ScratchArea[4][i] = p[i]->LeftFarSlope ;
    ScratchArea[5][i] = p[i]->RightNearSlope ;
    ScratchArea[6][i] = p[i]->RightFarSlope ;
  }

  //
  // 1st-order statistical state history summary
  //

  SystemInfo->State.RollAvg.NearFieldThreshold = RollingAverage ( ScratchArea[0] , gmLD_SYSTEM_HISTORYDEP
TH ) ;
  SystemInfo->State.RollAvg.MidFieldThreshold  = RollingAverage ( ScratchArea[1] , gmLD_SYSTEM_HISTORYDEP
TH )  ;
  SystemInfo->State.RollAvg.FarFieldThreshold  = RollingAverage ( ScratchArea[2] , gmLD_SYSTEM_HISTORYDEP
TH )  ;
  SystemInfo->State.RollAvg.LeftNearSlope      = RollingAverage ( ScratchArea[3] , gmLD_SYSTEM_HISTORYDEP
TH )  ;
  SystemInfo->State.RollAvg.LeftFarSlope       = RollingAverage ( ScratchArea[4] , gmLD_SYSTEM_HISTORYDEP
TH )  ;
  SystemInfo->State.RollAvg.RightNearSlope     = RollingAverage ( ScratchArea[5] , gmLD_SYSTEM_HISTORYDEP
TH )  ;
```

```
  SystemInfo->State.RollAvg.RightFarSlope       = RollingAverage ( ScratchArea[6] , gmLD_SYSTEM_HISTORYDEP
TH )  ;

  //
  // now load up the scratch area with
  // the derivative history information
  //

  for ( i = 0 ; i < gmLD_SYSTEM_HISTORYDEPTH ; i ++ )
  {
    ScratchArea[0][i] = p1[i]->NearFieldThreshold ;
    ScratchArea[1][i] = p1[i]->MidFieldThreshold ;
    ScratchArea[2][i] = p1[i]->FarFieldThreshold ;
    ScratchArea[3][i] = p1[i]->LeftNearSlope ;
    ScratchArea[4][i] = p1[i]->LeftFarSlope ;
    ScratchArea[5][i] = p1[i]->RightNearSlope ;
    ScratchArea[6][i] = p1[i]->RightFarSlope ;
  }

  //
  // 2dn-order statistical state derivative summary
  //


  StatCalc ( & ScratchStats , ScratchArea[0] , gmLD_SYSTEM_HISTORYDEPTH ) ;
  SystemInfo->State.DerivativeHistoryVariance.NearFieldThreshold = ScratchStats.Variance ;

  StatCalc ( & ScratchStats , ScratchArea[1] , gmLD_SYSTEM_HISTORYDEPTH ) ;
  SystemInfo->State.DerivativeHistoryVariance.MidFieldThreshold  = ScratchStats.Variance ;

  StatCalc ( & ScratchStats , ScratchArea[2] , gmLD_SYSTEM_HISTORYDEPTH ) ;
  SystemInfo->State.DerivativeHistoryVariance.FarFieldThreshold  = ScratchStats.Variance ;

  StatCalc ( & ScratchStats , ScratchArea[3] , gmLD_SYSTEM_HISTORYDEPTH ) ;
  SystemInfo->State.DerivativeHistoryVariance.LeftNearSlope      = ScratchStats.Variance ;

  StatCalc ( & ScratchStats , ScratchArea[4] , gmLD_SYSTEM_HISTORYDEPTH ) ;
  SystemInfo->State.DerivativeHistoryVariance.LeftFarSlope       = ScratchStats.Variance ;

  StatCalc ( & ScratchStats , ScratchArea[5] , gmLD_SYSTEM_HISTORYDEPTH ) ;
  SystemInfo->State.DerivativeHistoryVariance.RightNearSlope     = ScratchStats.Variance ;

  StatCalc ( & ScratchStats , ScratchArea[6] , gmLD_SYSTEM_HISTORYDEPTH ) ;
  SystemInfo->State.DerivativeHistoryVariance.RightFarSlope      = ScratchStats.Variance ;

  //
  // now we produce the current smoothed value
  //

  gmSYSTEM_STATE * r = & SystemInfo->State.DerivativeHistoryVariance ;
  gmSYSTEM_STATE * s = & SystemInfo->State.SmoothedDerivativeHistoryVariance ;
  double  SmoothingFc = SystemInfo->State.VarianceSmoothingFreq ;

  s->NearFieldThreshold = LowPassIIR ( r->NearFieldThreshold , s->NearFieldThreshold , SmoothingFc ) ;
  s->MidFieldThreshold  = LowPassIIR ( r->MidFieldThreshold  , s->MidFieldThreshold  , SmoothingFc ) ;
  s->FarFieldThreshold  = LowPassIIR ( r->FarFieldThreshold  , s->FarFieldThreshold  , SmoothingFc ) ;
  s->LeftNearSlope      = LowPassIIR ( r->LeftNearSlope      , s->LeftNearSlope      , SmoothingFc ) ;
  s->LeftFarSlope       = LowPassIIR ( r->LeftFarSlope       , s->LeftFarSlope       , SmoothingFc ) ;
  s->RightNearSlope     = LowPassIIR ( r->RightNearSlope     , s->RightNearSlope     , SmoothingFc ) ;
  s->RightFarSlope      = LowPassIIR ( r->RightFarSlope      , s->RightFarSlope      , SmoothingFc ) ;


  s->NearFieldThreshold = gmMaxClip ( s->NearFieldThreshold , gmMAXVARIANCE_THRESHOLD_DEFAULT ) ;
  s->MidFieldThreshold  = gmMaxClip ( s->MidFieldThreshold  , gmMAXVARIANCE_THRESHOLD_DEFAULT ) ;
  s->FarFieldThreshold  = gmMaxClip ( s->FarFieldThreshold  , gmMAXVARIANCE_THRESHOLD_DEFAULT ) ;
  s->LeftNearSlope      = gmMaxClip ( s->LeftNearSlope      , gmMAXVARIANCE_THRESHOLD_DEFAULT ) ;
  s->LeftFarSlope       = gmMaxClip ( s->LeftFarSlope       , gmMAXVARIANCE_THRESHOLD_DEFAULT ) ;
  s->RightNearSlope     = gmMaxClip ( s->RightNearSlope     , gmMAXVARIANCE_THRESHOLD_DEFAULT ) ;
  s->RightFarSlope      = gmMaxClip ( s->RightFarSlope      , gmMAXVARIANCE_THRESHOLD_DEFAULT ) ;

  gmTRACE_MAC ( "gmLaneDetector::UpdateStateInfo() : Exiting\n" ) ;

}
```

```
/**********************
 *
 **********************/

void gmLaneDetector::UpdateLaneTrackingStatus ( )
{
  gmTRACE_MAC ( "gmLaneDetector::UpdateLaneTrackingStatus() : Entering\n" ) ;



  if ( ! ( FramesProcessed > gmLD_SYSTEM_HISTORYDEPTH ) )
    return ;


  SystemInfo->CurrentLane.RightNearSlope.Valid = FALSE ;
  SystemInfo->CurrentLane.LeftNearSlope.Valid  = FALSE ;
  SystemInfo->CurrentLane.RightFarSlope.Valid  = FALSE ;
  SystemInfo->CurrentLane.LeftFarSlope.Valid   = FALSE ;


  SystemInfo->LaneLockedOn      = FALSE ;
  SystemInfo->NearLaneAcquired = FALSE ;
  SystemInfo->MidLaneAcquired  = FALSE ;
  SystemInfo->FarLaneAcquired  = FALSE ;


  if ( SystemInfo->State.SmoothedDerivativeHistoryVariance.RightNearSlope < gmVALIDSLOPE_THRESHOLD_DEFAUL
T )
    SystemInfo->CurrentLane.RightNearSlope.Valid = TRUE ;

  if ( SystemInfo->State.SmoothedDerivativeHistoryVariance.LeftNearSlope < gmVALIDSLOPE_THRESHOLD_DEFAULT
 )
    SystemInfo->CurrentLane.LeftNearSlope.Valid = TRUE ;


  if ( SystemInfo->CurrentLane.RightNearSlope.Valid  &&  SystemInfo->CurrentLane.LeftNearSlope.Valid  )
    SystemInfo->NearLaneAcquired = TRUE ;

  if ( SystemInfo->CurrentLane.RightNearSlope.Valid  &&  SystemInfo->CurrentLane.LeftNearSlope.Valid  )
    SystemInfo->MidLaneAcquired = TRUE ;



  if ( ! SystemInfo->FarSlopesEnabled )
    return ;



  if ( SystemInfo->State.SmoothedDerivativeHistoryVariance.RightFarSlope < gmVALIDSLOPE_THRESHOLD_DEFAULT
 )
    SystemInfo->CurrentLane.RightFarSlope.Valid = TRUE ;

  if ( SystemInfo->State.SmoothedDerivativeHistoryVariance.LeftFarSlope < gmVALIDSLOPE_THRESHOLD_DEFAULT
)
    SystemInfo->CurrentLane.LeftFarSlope.Valid = TRUE ;


  if ( SystemInfo->CurrentLane.RightFarSlope.Valid  &&  SystemInfo->CurrentLane.LeftFarSlope.Valid  )
    SystemInfo->FarLaneAcquired = TRUE ;


  if ( SystemInfo->NearLaneAcquired  &&  SystemInfo->FarLaneAcquired )
    SystemInfo->LaneLockedOn = TRUE ;



  gmTRACE_MAC ( "gmLaneDetector::UpdateLaneTrackingStatus() : Exiting\n" ) ;

}

/*********************
 *
 *********************/
```

```
void gmLaneDetector::UpdateSystemInfo ( )
{
  gmTRACE_MAC ( "gmLaneDetector::UpdateSystemInfo() : Entering\n" ) ;

  UpdateStateInfo ( ) ;

  UpdateLaneTrackingStatus( ) ;

  gmTRACE_MAC ( "gmLaneDetector::UpdateSystemInfo() : Exiting\n" ) ;
}

/*********************
 *
 *********************/

void gmLaneDetector::AdaptParameters ( )
{
  gmTRACE_MAC ( "gmLaneDetector::AdaptParameters() : Entering\n" ) ;

  if ( FramesProcessed > 3 * gmLD_SYSTEM_HISTORYDEPTH )
    SystemInfo->State.VarianceSmoothingFreq = gmVARIANCE_SMOOTHINGFREQ_ONLINE ;

  gmTRACE_MAC ( "gmLaneDetector::AdaptParameters() : Exiting\n" ) ;
}

/*********************
 *
 *********************

void gmLaneDetector:: ( )
{
  gmTRACE_MAC ( "gmLaneDetector:: : Entering\n" ) ;

  gmTRACE_MAC ( "gmLaneDetector:: : Exiting\n" ) ;
}

/*********************
 *
 *********************/

/**************************************************************************

                    Following are this class's Public Methods

 **************************************************************************/

/*********************
 *
 *********************/

gmLaneDetector::gmLaneDetector  ( gmVideoStream * VideoStream , gmKlugeDetector * PlateDetector , BOOL Te
stEnable )
{
  gmTRACE_MAC ( "gmLaneDetector::gmLaneDetector() : Entering\n" ) ;

  DumpEnable = FALSE ;


  this->TestEnable = TestEnable ;

  this->VideoStream = VideoStream ;

  ViewPort = VideoStream->GetViewPort() ;

  LicencePlateDetector = PlateDetector ;


  VideoFrameBuffer          = VideoStream->GetFrameImageBuf() ;
  FrameWidth                = VideoFrameBuffer->GetCols() ;
  FrameHeight               = VideoFrameBuffer->GetRows() ;
  FrameArea                 = FrameWidth * FrameHeight ;
  FrameVertCenter           = FrameWidth >> 1 ;
  FrameHorizCenter          = FrameHeight >> 1 ;
```

```
  VideoFrameImageBaseAdx     = ( BYTE * ) VideoFrameBuffer->GetImageBaseAdx() ;

  RowImageBuf                = new gmBuffer ( 1 , FrameWidth , gmDT_BYTE ) ;
  RowDistributionBuf         = new gmBuffer ( 1 , FrameWidth , gmDISTRIBUTION_DATATYPE_DEFAULT ) ;
  RowHitBuf                  = new BYTE [ FrameWidth ] ;

  HitSetCapacity             = gmLD_HITSETSIZE ;
  HitSet.Element             = new gmHITINFO [ gmLD_HITSETSIZE ] ;

  FramesProcessed            = 0 ;

  LaneLockCount              = 0 ;

  LeftPointShots             = 0 ;
  RightPointShots            = 0 ;

  FarLaneBias                = 0 ;
  FarLaneOffset              = 0 ;
  FarLaneCenter              = 0 ;
  FarLaneWidth               = 0 ;

  //
  // here we initialize the
  // near field baseline value
  //

  printf ( "   ** Select left-side near Evaluation Region :" ) ;

  while ( ! ViewPort->SelectionReady ( ) )
    Sleep ( 50 ) ;

  RECT InitialNearLeftEvaluationRegion = ViewPort->GetSelectedRegion ( ) ;

  ViewPort->ResetSelection() ;

  printf ( "\n" ) ;

  //
  // 07/14/98
  // we've got to make sure our licence plate
  // detector is not NULL . later we can add some
  // more logic for the NULL case . but for now
  // the problem is a fatal error .
  //                  -gm
  //

  if ( ! LicencePlateDetector )
    gmERROR_MAC ( "  gmLaneDetector::gmLaneDetector() : LicencePlateDetector is NULL : Cannot proceed " )

  //
  // here we now adapt our working video fields
  // to the currently selected regions .
  //

  InitialNearLeftField.BaseLine      = InitialNearRightField.BaseLine   = InitialNearLeftEvaluationRegion
.top ;

  InitialNearLeftField.Region.top    = InitialNearRightField.Region.top = InitialNearLeftEvaluationRegion
.top ;
  InitialNearLeftField.Region.bottom = InitialNearRightField.Region.bottom = InitialNearLeftEvaluationReg
ion.top + gmFIELDHEIGHT_DEFAULT ;


  InitialFarLeftField.BaseLine       = InitialFarRightField.BaseLine    = ( unsigned ) LicencePlateDetecto
r->GetCurrentRearShadowRow ( ) - 2 ;
  InitialFarLeftField.Region.top     = InitialFarRightField.Region.top = InitialFarRightField.BaseLine ;

  InitialFarLeftField.Region.bottom  = InitialFarLeftField.Region.top  + InitialFarLeftField.Height ;
  InitialFarRightField.Region.bottom = InitialFarRightField.Region.top + InitialFarRightField.Height ;


  InitialMidLeftField.BaseLine       = InitialMidRightField.BaseLine    = ( InitialFarLeftField.BaseLine +
 InitialNearLeftField.BaseLine ) >> 1 ;
  InitialMidLeftField.Region.top     = InitialMidRightField.Region.top = InitialMidLeftField.BaseLine ;
```

```
  InitialMidLeftField.Region.bottom  = InitialMidLeftField.Region.top  + InitialMidLeftField.Height ;
  InitialMidRightField.Region.bottom = InitialMidRightField.Region.top + InitialMidRightField.Height ;



  InitialFieldInfo.NearLeft  = InitialNearLeftField ;
  InitialFieldInfo.NearRight = InitialNearRightField ;

  InitialFieldInfo.MidLeft   = InitialMidLeftField ;
  InitialFieldInfo.MidRight  = InitialMidRightField ;

  InitialFieldInfo.FarLeft   = InitialFarLeftField ;
  InitialFieldInfo.FarRight  = InitialFarRightField ;



  InitSystemInfo ( gmLD_SYSTEM_HISTORYDEPTH ) ;


  gmTRACE_MAC ( "gmLaneDetector::gmLaneDetector() : Entering\n" ) ;

}

/*********************
 *
 *********************/

gmLaneDetector::~gmLaneDetector  ( )
{
  gmTRACE_MAC ( "gmLaneDetector::~gmLaneDetector() : Entering\n" ) ;

  delete   RowImageBuf ;
  delete   RowDistributionBuf ;
  delete []RowHitBuf ;

  delete [](HitSet.Element) ;

  unsigned i ;
  for ( i = 0 ; i < gmLD_SYSTEM_HISTORYDEPTH ; i ++ )
    delete SystemInfo->State.DerivativeHistory[i] ;

  delete [](SystemInfo->State.DerivativeHistory) ;
  delete    SystemInfo ;


  gmTRACE_MAC ( "gmLaneDetector::~gmLaneDetector() : Exiting\n" ) ;
}

/*********************
 *
 *********************/

void gmLaneDetector::SmoothNewLaneLocation ( double CutOffFreqHz )
{
  gmTRACE_MAC ( "gmLaneDetector::SmoothNewLaneLocation : Entering\n" ) ;


  if ( ! FramesProcessed )
    return ;


  gmLANEINFO        * p = & SystemInfo->CurrentLane  ;
  gmLANEINFO        * q = & SystemInfo->PreviousLane ;
  gmVIDEOFIELDINFO  * r = & SystemInfo->Field  ;


  p->Near.Left.Row  = ( unsigned ) LowPassIIR ( ( double ) p->Near.Left.Row  , ( double ) q->Near.Left.Ro
w  , r->NearLeft.LowPassFc  ) ;
  p->Near.Left.Col  = ( unsigned ) LowPassIIR ( ( double ) p->Near.Left.Col  , ( double ) q->Near.Left.Co
l  , r->NearLeft.LowPassFc  ) ;
  p->Mid.Left.Row   = ( unsigned ) LowPassIIR ( ( double ) p->Mid.Left.Row   , ( double ) q->Mid.Left.Row
   , r->MidLeft.LowPassFc   ) ;
```

```
  p->Mid.Left.Col    = ( unsigned ) LowPassIIR ( ( double ) p->Mid.Left.Col    , ( double ) q->Mid.Left.Col
   , r->MidLeft.LowPassFc   ) ;
  p->Far.Left.Row    = ( unsigned ) LowPassIIR ( ( double ) p->Far.Left.Row    , ( double ) q->Far.Left.Row
   , r->FarLeft.LowPassFc   ) ;
  p->Far.Left.Col    = ( unsigned ) LowPassIIR ( ( double ) p->Far.Left.Col    , ( double ) q->Far.Left.Col
   , r->FarLeft.LowPassFc   ) ;

  p->Near.Right.Row = ( unsigned ) LowPassIIR ( ( double ) p->Near.Right.Row , ( double ) q->Near.Right.R
ow , r->NearRight.LowPassFc ) ;
  p->Near.Right.Col = ( unsigned ) LowPassIIR ( ( double ) p->Near.Right.Col , ( double ) q->Near.Right.C
ol , r->NearRight.LowPassFc ) ;
  p->Mid.Right.Row  = ( unsigned ) LowPassIIR ( ( double ) p->Mid.Right.Row  , ( double ) q->Mid.Right.Ro
w  , r->MidRight.LowPassFc  ) ;
  p->Mid.Right.Col  = ( unsigned ) LowPassIIR ( ( double ) p->Mid.Right.Col  , ( double ) q->Mid.Right.Co
l  , r->MidRight.LowPassFc  ) ;
  p->Far.Right.Row  = ( unsigned ) LowPassIIR ( ( double ) p->Far.Right.Row  , ( double ) q->Far.Right.Ro
w  , r->FarRight.LowPassFc  ) ;
  p->Far.Right.Col  = ( unsigned ) LowPassIIR ( ( double ) p->Far.Right.Col  , ( double ) q->Far.Right.Co
l  , r->FarRight.LowPassFc  ) ;


  if ( ! SystemInfo->FarSlopesEnabled )
    return ;


//  if ( ! FarLaneBias )
//    FarLaneBias = p->Far.Right.Col ;


  FarLaneOffset = p->Far.Right.Col - FarLaneBias ;
  FarLaneCenter = ( p->Far.Right.Col + p->Far.Left.Col ) >> 1 ;
  FarLaneWidth  = p->Far.Right.Col - p->Far.Left.Col + 1 ;


  gmTRACE_MAC ( "gmLaneDetector::SmoothNewLaneLocation : Exiting\n" ) ;
}

/**********************
 *
 **********************/

gmBUFLOC gmLaneDetector::RegressHitSet ( gmHITSET * HitSet )
{
  gmTRACE_MAC ( "gmLaneDetector::RegressHitSet() : Entering\n" ) ;

  gmHITINFO * p = HitSet->Element ;
  gmBUFLOC    LocBestFit ;
  double w , x , y , z ;
  double   m , b , xMean , Denom ;

  unsigned i ;

  w = x = y = z = 0 ;


  for ( i = 0 ; i < HitSet->Size ; i ++ )
  {
    w += ( double ) p[i].Center  * ( double ) p[i].RowNo ;
    x += ( double ) p[i].Center  ;
    y += ( double ) p[i].RowNo       ;
    z += ( double ) p[i].Center  * ( double ) p[i].Center ;
  }


  if ( FrameNo == 98 )
    i = i ;


  xMean  = x / HitSet->Size ;

  Denom = HitSet->Size * z - x * x  ;

  LocBestFit.Col  = ( unsigned ) xMean  ;
```

```
  if ( Denom )
  {
    m =  (  HitSet->Size * w - x * y ) / Denom ;
    b = ( y - m * x ) / HitSet->Size   ;

    LocBestFit.Row  = ( unsigned ) ( m * xMean  + b ) ;
  }

  else
    LocBestFit.Row  = p->RowNo ;



  return LocBestFit ;


  gmTRACE_MAC ( "gmLaneDetector::RegressHitSet() : Exiting\n" ) ;
}

/*********************
 *
 *********************/

gmBUFLOC gmLaneDetector::RegressHitSet ( )
{
  gmTRACE_MAC ( "gmLaneDetector::RegressHitSet : Entering\n" ) ;

  return RegressHitSet ( & HitSet ) ;

  gmTRACE_MAC ( "gmLaneDetector::RegressHitSet : Exiting\n" ) ;
}

/*********************
 *
 *********************/

void gmLaneDetector::DisplayResult ( )
{
  gmTRACE_MAC ( "gmLaneDetector::DisplayResult : Entering\n" ) ;


  ViewPort->PaintRectangle ( SystemInfo->Field.NearLeft.Region  ) ;
  ViewPort->PaintRectangle ( SystemInfo->Field.NearRight.Region ) ;
  ViewPort->PaintRectangle ( SystemInfo->Field.MidLeft.Region   ) ;
  ViewPort->PaintRectangle ( SystemInfo->Field.MidRight.Region  ) ;
  ViewPort->PaintRectangle ( SystemInfo->Field.FarLeft.Region   ) ;
  ViewPort->PaintRectangle ( SystemInfo->Field.FarRight.Region  ) ;


  ViewPort->PaintLine ( SystemInfo->CurrentLane.Near.Left , SystemInfo->CurrentLane.Mid.Left );
  ViewPort->PaintLine ( SystemInfo->CurrentLane.Mid.Left  , SystemInfo->CurrentLane.Far.Left );

  ViewPort->PaintLine ( SystemInfo->CurrentLane.Near.Right , SystemInfo->CurrentLane.Mid.Right );
  ViewPort->PaintLine ( SystemInfo->CurrentLane.Mid.Right  , SystemInfo->CurrentLane.Far.Right );


  ViewPort->PaintCrossHairs ( SystemInfo->CurrentLane.Near.Left ) ;
  ViewPort->PaintCrossHairs ( SystemInfo->CurrentLane.Mid.Left  ) ;
  ViewPort->PaintCrossHairs ( SystemInfo->CurrentLane.Far.Left  ) ;

  ViewPort->PaintCrossHairs ( SystemInfo->CurrentLane.Near.Right ) ;
  ViewPort->PaintCrossHairs ( SystemInfo->CurrentLane.Mid.Right  ) ;
  ViewPort->PaintCrossHairs ( SystemInfo->CurrentLane.Far.Right  ) ;


  if ( SystemInfo->LaneLockedOn )
  {
    ViewPort->SetLaneLockedOn() ;

    if ( ! FarLaneBias )
      if ( LaneLockCount )
        FarLaneBias = SystemInfo->CurrentLane.Far.Right.Col ;
```

```
    LaneLockCount ++ ;
  }

  else
    ViewPort->ResetLaneLockedOn() ;


  ViewPort->Update ( ) ;


  gmTRACE_MAC ( "gmLaneDetector::DisplayResult : Exiting\n" ) ;
}

/**********************
 *
 **********************/

unsigned gmLaneDetector::ProcessRow ( gmVIDEOFIELD * Field , unsigned Offset )

{

  gmTRACE_MAC ( "gmLaneDetector::ProcessRow : Entering\n" ) ;

  unsigned i , j , k ;
  unsigned Retries , TargetSizeIncrement = 1 , MaxRetries = 5 ;
  unsigned Hits , Detections , DetectionCenter ;


  if ( Field->Threshold.Modifiable )
    MaxRetries = 5 ;
  else
    MaxRetries = 0 ;


  double   Threshold        = Field->Threshold.Level ;
  unsigned LeftSearchBound  = Field->Region.left  ;
  unsigned RightSearchBound = Field->Region.right ;
  unsigned RowNo            = Field->Region.top + Offset ;


  if ( RowImageBuf->GetIdx() != RowNo )
  {
    * RowImageBuf = VideoFrameBuffer->GetRowPartial ( RowNo , LeftSearchBound , RightSearchBound ) ;
    RowImageBuf->CalcStats ( ) ;
    * RowDistributionBuf = RowImageBuf->Distribute ( ) ;


    if ( DumpEnable )
    {
      printf ( " ** FrameNo : %d\n" , FrameNo ) ;
      printf ( "      RowNo : %d\n" , RowNo    ) ;
      printf ( "  LeftBound : %d\n" , LeftSearchBound  ) ;
      printf ( " RightBound : %d\n" , RightSearchBound ) ;
      printf ( " *************************************\n" ) ;
      DumpByteBuf   ( ( BYTE * )  RowImageBuf->GetImageBaseAdx() , RowImageBuf->GetElements() - 1 ) ;
      DumpFloatBuf  ( ( float * ) RowDistributionBuf->GetImageBaseAdx() , RowDistributionBuf->GetElements(
) - 1 ) ;
      exit ( 1 ) ;
    }

  }


  const float * RowDistributionImage = ( const float * ) RowDistributionBuf->GetImageBaseAdx ( ) ;
  unsigned n = RowImageBuf->GetElements() ;

  Detections = 0 ;
  Retries    = 0 ;


  while ( ! ( Retries > MaxRetries ) )
  {
    ClearRowHitBuf ( 0 , n - 1 ) ;
```

```
DetectionCenter = 0 ;

Hits = 0 ;

for ( i = 0 ; i <= n ; i ++ )
{
  if ( RowDistributionImage[i] >= Threshold )
  {
    RowHitBuf[i] = 0xff ;
    Hits ++ ;
  }

}


if ( Hits >= MinimumDetectionWidth )
{

  for ( i = 0 ; i <= n ; i ++ )
  {

    if ( RowHitBuf[i] )
    {

      for ( j = i ; j <= n ; j ++ )
        if ( ! RowHitBuf[j] )
          break ;

      k = j - i + 1 ;

      if ( k >= MinimumDetectionWidth )
        if ( k <= MaximumDetectionWidth )
        {
          Detections ++ ;

          DetectionCenter = LeftSearchBound + i + k / 2 ;

          if ( Detections > 1 )
            break ;

        }


      i = j + 1 ;

    }

  }

}


//
// if we've found multiples , raise threshold
//
if ( Detections > 1 )
  Threshold *= ( 1.00 + Field->Threshold.StepFactor ) ;

//
// if we've found only one , then we're done
//
else if ( Detections == 1 )
  break ;

//
// if we've found none , then lower threshold
//
else if ( Threshold > Field->Threshold.MinimumAllowable )
  Threshold *= ( 1.00 - Field->Threshold.StepFactor ) ;

//
// if SoftThreshold is already at minimum
// allowable , then there's nothing detectible
// in this data : go home .
```

```
    //
    else
    {
      Retries = MaxRetries + 1 ;
      break ;
    }

    //
    // make sure we count this try
    //

    Retries ++ ;

  }

  //
  // if we've arrived here with
  // less than MaxRetries , we've
  // detected a target which meets
  // our entry criteria .
  //

  if ( ! ( Retries > MaxRetries ) )
  {
    if ( Field->Threshold.Modifiable  )
      Field->Threshold.Level = Threshold ;
  }

  else
    DetectionCenter = gmTARGET_NOTFOUND ;


  gmTRACE_MAC ( "gmLaneDetector::ProcessRow : Exiting\n" ) ;


  return DetectionCenter ;

}

/*********************
 *
 *********************/

unsigned gmLaneDetector::ScanField ( gmVIDEOFIELD * Field )
{

  gmTRACE_MAC ( "gmLaneDetector::ScanField : Entering\n" ) ;

  gmHITSET * p = & HitSet ;

  unsigned i , DetectionCenter ;


  if ( FrameNo == 19 )
    i = i ;


  for ( i = p->Size = 0 ; ( ( i < Field->Height )  &&  ( p->Size < HitSetCapacity ) ) ; i ++ )
  {

    DetectionCenter = ProcessRow ( Field , i ) ;

    if ( DetectionCenter )
    {
      p->Element[p->Size].Center = DetectionCenter ;
      p->Element[p->Size].RowNo  = Field->Region.top + i  ;
      p->Size ++ ;
    }


  }


  gmTRACE_MAC ( "gmLaneDetector::ScanField : Exiting\n" ) ;
```

```
  return HitSet.Size ;
}

/**********************
 *
 **********************/

void gmLaneDetector::ProcessFarField ( )
{
  gmTRACE_MAC ( "gmLaneDetector::ProcessFarField : Entering\n" ) ;

  int i ;

  unsigned LeftLimitFarField  = gmINITIAL_FARFIELD_LEFTLIMIT  ;
  unsigned RightLimitFarField = gmINITIAL_FARFIELD_RIGHTLIMIT ;


  MinimumDetectionWidth = 3 ;
  MaximumDetectionWidth = 10 ;


  if ( FramesProcessed == 100 )
    i = i ;


  if ( SystemInfo->MidLaneAcquired )
  {
    gmVIDEOFIELD * FarLeftField   = & SystemInfo->Field.FarLeft  ;
    gmVIDEOFIELD * FarRightField  = & SystemInfo->Field.FarRight ;
    gmVIDEOFIELD * MidLeftField   = & SystemInfo->Field.MidLeft ;
    gmVIDEOFIELD * MidRightField  = & SystemInfo->Field.MidRight ;
    gmVIDEOFIELD * NearLeftField  = & SystemInfo->Field.NearLeft ;
    gmVIDEOFIELD * NearRightField = & SystemInfo->Field.NearRight ;

    FarLeftField->Threshold.Level = FarRightField->Threshold.Level  ;
    FarLeftField->Threshold.Modifiable  = FALSE ;
    FarRightField->Threshold.Modifiable = TRUE  ;

    // gmFIELDHEIGHT_DEFAULT
    //
    // here is where we adapt our mid and far field
    // parameters to the current location of the
    // target vehicle's rear shadow .
    //

    FarLeftField->BaseLine       = FarRightField->BaseLine   = ( unsigned ) LicencePlateDetector->GetCurr
entRearShadowRow ( ) - 2 ;
    FarLeftField->Region.top     = FarRightField->Region.top = FarRightField->BaseLine ;

    FarRightField->Region.bottom = FarRightField->Region.top + FarRightField->Height ;
    FarLeftField->Region.bottom  = FarLeftField->Region.top  + FarLeftField->Height ;

    MidLeftField->BaseLine       = MidRightField->BaseLine   = ( NearLeftField->BaseLine + FarLeftField->
BaseLine ) >> 1 ;
    MidLeftField->Region.top     = MidRightField->Region.top = MidRightField->BaseLine ;

    MidRightField->Region.bottom = MidRightField->Region.top + MidRightField->Height ;
    MidLeftField->Region.bottom  = MidLeftField->Region.top  + MidLeftField->Height ;


    double TimeSinceMostRecentLeftDetection  = SystemInfo->Time.Elapsed - SystemInfo->CurrentLane.Far.Mos
tRecentLeftDetectionTime  ;
    double TimeSinceMostRecentRightDetection = SystemInfo->Time.Elapsed - SystemInfo->CurrentLane.Far.Mos
tRecentRightDetectionTime ;
    double LaneLocTimeOutVal = SystemInfo->CurrentLane.Far.LaneLocTimeOutVal ;

    unsigned FarLeftColEstimate , FarRightColEstimate ;


    if ( ( TimeSinceMostRecentRightDetection < LaneLocTimeOutVal )  &&  SystemInfo->CurrentLane.RightFarS
lope.Valid )
    {
      FarRightField->Region.left  = SystemInfo->CurrentLane.Far.Right.Col - 20 ;
```

```
      FarRightField->Region.right = SystemInfo->CurrentLane.Far.Right.Col + 20 ;
    }

    else if ( ( SystemInfo->CurrentLane.RightNearSlope.Valid )  &&  ( TimeSinceMostRecentRightDetection <
 3 * LaneLocTimeOutVal ) )
    {
      RightPointShots ++ ;
      FarRightColEstimate = PointShoot ( & SystemInfo->CurrentLane.Mid.Right , SystemInfo->State.RollAvg.
RightNearSlope , FarRightField->BaseLine ) ;
      FarRightField->Region.left  = FarRightColEstimate - 20 ;
      FarRightField->Region.right = FarRightColEstimate + 20 ;

      if ( ( unsigned ) FarRightField->Region.right > SystemInfo->CurrentLane.Mid.Right.Col )
      {
        FarRightField->Region.right = SystemInfo->CurrentLane.Mid.Right.Col ;
        FarRightField->Region.left = FarRightField->Region.right - 40 ;
      }

    }

    else
    {
      * FarRightField = InitialFarRightField;
///      FarRightField->Region.right = SystemInfo->Field.MidRight.Region.right ;
      FarRightField->Region.right = SystemInfo->CurrentLane.Mid.Right.Col ;
      FarRightField->Threshold.Modifiable = TRUE  ;
    }


    if ( ( TimeSinceMostRecentLeftDetection < 5 * LaneLocTimeOutVal )  &&  SystemInfo->CurrentLane.LeftFa
rSlope.Valid  )
    {
      FarLeftField->Region.left  = SystemInfo->CurrentLane.Far.Left.Col - 20 ;
      FarLeftField->Region.right = SystemInfo->CurrentLane.Far.Left.Col + 20 ;
    }

    else if ( ( SystemInfo->CurrentLane.LeftNearSlope.Valid )  &&  ( TimeSinceMostRecentLeftDetection < 5
 * LaneLocTimeOutVal ) )
    {
      LeftPointShots ++ ;
      FarLeftColEstimate = PointShoot ( & SystemInfo->CurrentLane.Mid.Left , SystemInfo->State.RollAvg.Le
ftNearSlope , FarLeftField->BaseLine ) ;
      FarLeftField->Region.left  = FarLeftColEstimate - 20 ;
      FarLeftField->Region.right = FarLeftColEstimate + 20 ;

      if ( ( unsigned ) FarLeftField->Region.left < SystemInfo->CurrentLane.Mid.Left.Col )
      {
        FarLeftField->Region.left = SystemInfo->CurrentLane.Mid.Left.Col ;
        FarLeftField->Region.right = FarLeftField->Region.left + 40 ;
      }

      else if ( FarLeftField->Region.left >= FarRightField->Region.right )
      {
        FarLeftField->Region.left = SystemInfo->CurrentLane.Mid.Left.Col ;
        FarLeftField->Region.right = FarLeftField->Region.left + 40 ;
      }

    }

    else
    {
      * FarLeftField = InitialFarLeftField;
///      FarLeftField->Region.left = SystemInfo->Field.MidLeft.Region.left ;
      FarLeftField->Region.left = SystemInfo->CurrentLane.Mid.Left.Col ;
      FarLeftField->Threshold.Modifiable = TRUE  ;
    }



    if ( ScanField ( FarRightField ) )
    {
      SystemInfo->CurrentLane.Far.Right = RegressHitSet ( ) ;
      SystemInfo->CurrentLane.Far.MostRecentRightDetectionTime = SystemInfo->Time.Elapsed ;
```

```
      if ( ScanField ( FarLeftField ) )
      {
        SystemInfo->CurrentLane.Far.Left = RegressHitSet ( ) ;
        SystemInfo->CurrentLane.Far.MostRecentLeftDetectionTime = SystemInfo->Time.Elapsed ;
      }

    }

  }


  gmTRACE_MAC ( "gmLaneDetector::ProcessFarField : Exiting\n" ) ;

}

/*********************
 *
 *********************/

void gmLaneDetector::ProcessMidField ( )
{
  gmTRACE_MAC ( "gmLaneDetector::ProcessMidField : Entering\n" ) ;


  MinimumDetectionWidth = gmMINIMUM_BOUNDARYWIDTH_DEFAULT ;
  MaximumDetectionWidth = gmMAXIMUM_BOUNDARYWIDTH_DEFAULT ;

  const unsigned FieldWidth = 120 ;
  unsigned i = 0 , HalfFieldWidth = FieldWidth >> 1 ;

  if ( FrameNo == 19 )
    i = i ;

  if ( SystemInfo->LaneAnchored )
  {

    gmVIDEOFIELD * MidLeftField  = & SystemInfo->Field.MidLeft  ;
    gmVIDEOFIELD * MidRightField = & SystemInfo->Field.MidRight ;

    SystemInfo->Field.MidLeft.Threshold.Level = SystemInfo->Field.MidRight.Threshold.Level  ;
    MidLeftField->Threshold.Modifiable  = FALSE ;
    MidRightField->Threshold.Modifiable = TRUE  ;

    double TimeSinceMostRecentLeftDetection  = SystemInfo->Time.Elapsed - SystemInfo->CurrentLane.Mid.Mos
tRecentLeftDetectionTime  ;
    double TimeSinceMostRecentRightDetection = SystemInfo->Time.Elapsed - SystemInfo->CurrentLane.Mid.Mos
tRecentRightDetectionTime ;
    double LaneLocTimeOutVal = SystemInfo->CurrentLane.Mid.LaneLocTimeOutVal ;


    if ( TimeSinceMostRecentRightDetection < LaneLocTimeOutVal )
    {
      if ( SystemInfo->CurrentLane.Mid.Right.Col > ( 639 - HalfFieldWidth ) )
        MidRightField->Region.right = 639 ;
      else
        MidRightField->Region.right = SystemInfo->CurrentLane.Near.Right.Col ;

      MidRightField->Region.left = MidRightField->Region.right - FieldWidth ;

    }

    else
    {
      SystemInfo->Field.MidRight = InitialMidRightField;
      SystemInfo->MidLaneAcquired = FALSE ;
    }


    if ( TimeSinceMostRecentLeftDetection < 2.5 * LaneLocTimeOutVal )
    {

      if ( SystemInfo->CurrentLane.Mid.Left.Col <= HalfFieldWidth )
        MidLeftField->Region.left = 0 ;
      else
```

```
        MidLeftField->Region.left = SystemInfo->CurrentLane.Near.Left.Col ;

      MidLeftField->Region.right  = MidLeftField->Region.left  + FieldWidth ;

    }

    else
    {
      SystemInfo->Field.MidLeft  = InitialMidLeftField;
      MidLeftField->Threshold.Modifiable  = TRUE ;
    }


    if ( ScanField ( & SystemInfo->Field.MidRight ) )
    {
      SystemInfo->CurrentLane.Mid.Right = RegressHitSet ( ) ;

      SystemInfo->CurrentLane.Mid.MostRecentRightDetectionTime = SystemInfo->Time.Elapsed ;


      if ( ScanField ( & SystemInfo->Field.MidLeft ) )
      {
        SystemInfo->CurrentLane.Mid.Left = RegressHitSet ( ) ;

        SystemInfo->CurrentLane.Mid.MostRecentLeftDetectionTime = SystemInfo->Time.Elapsed ;

      }

    }

  }

  gmTRACE_MAC ( "gmLaneDetector::ProcessMidField : Exiting\n" ) ;

}

/**********************
 *
 **********************/

void gmLaneDetector::ProcessNearField ( )
{
  gmTRACE_MAC ( "gmLaneDetector::ProcessNearField : Entering\n" ) ;


  MinimumDetectionWidth = gmMINIMUM_BOUNDARYWIDTH_DEFAULT ;
  MaximumDetectionWidth = gmMAXIMUM_BOUNDARYWIDTH_DEFAULT ;


  const unsigned FieldWidth = 120 ;
  unsigned i = 0 , HalfFieldWidth = FieldWidth >> 1 ;

  if ( FramesProcessed == 104 )
    i = i ;

  gmVIDEOFIELD * NearLeftField  = & SystemInfo->Field.NearLeft  ;
  gmVIDEOFIELD * NearRightField = & SystemInfo->Field.NearRight ;

  SystemInfo->Field.NearLeft.Threshold.Level = SystemInfo->Field.NearRight.Threshold.Level  ;
  NearLeftField->Threshold.Modifiable  = FALSE ;
  NearRightField->Threshold.Modifiable = TRUE  ;

  double TimeSinceMostRecentLeftDetection  = SystemInfo->Time.Elapsed - SystemInfo->CurrentLane.Near.Most
RecentLeftDetectionTime  ;
  double TimeSinceMostRecentRightDetection = SystemInfo->Time.Elapsed - SystemInfo->CurrentLane.Near.Most
RecentRightDetectionTime ;
  double LaneLocTimeOutVal = SystemInfo->CurrentLane.Near.LaneLocTimeOutVal ;


  if ( FramesProcessed > 3 * gmLD_SYSTEM_HISTORYDEPTH )
  {
```

```
    if ( TimeSinceMostRecentRightDetection < LaneLocTimeOutVal )
    {
      if ( SystemInfo->CurrentLane.Near.Right.Col > ( 639 - HalfFieldWidth ) )
        NearRightField->Region.right = 639 ;
      else
        NearRightField->Region.right = SystemInfo->CurrentLane.Near.Right.Col + HalfFieldWidth ;

      NearRightField->Region.left = NearRightField->Region.right - FieldWidth ;

    }

    else
    {
      SystemInfo->Field.NearRight = InitialNearRightField;
    }

    if ( TimeSinceMostRecentLeftDetection < 2.5 * LaneLocTimeOutVal )
    {
      if ( SystemInfo->CurrentLane.Near.Left.Col <= HalfFieldWidth )
        NearLeftField->Region.left = 0 ;
      else
        NearLeftField->Region.left = SystemInfo->CurrentLane.Near.Left.Col  - HalfFieldWidth ;

      NearLeftField->Region.right  = NearLeftField->Region.left  + FieldWidth ;

    }

    else
    {
      SystemInfo->Field.NearLeft = InitialNearLeftField;
      NearLeftField->Threshold.Modifiable  = TRUE ;
    }



  }



  if ( ScanField ( & SystemInfo->Field.NearRight ) )
  {

    SystemInfo->LaneAnchored = TRUE ;
    SystemInfo->CurrentLane.Near.Right = RegressHitSet ( ) ;
    SystemInfo->CurrentLane.Near.MostRecentRightDetectionTime = SystemInfo->Time.Elapsed ;

    if ( ScanField ( & SystemInfo->Field.NearLeft ) )
    {
      SystemInfo->CurrentLane.Near.Left = RegressHitSet ( ) ;
      SystemInfo->CurrentLane.Near.MostRecentLeftDetectionTime = SystemInfo->Time.Elapsed ;
    }

  }

  else
  {
    SystemInfo->LaneAnchored    = FALSE ;
    SystemInfo->Field.NearLeft  = InitialNearLeftField;
    SystemInfo->Field.NearRight = InitialNearRightField;
  }

  gmTRACE_MAC ( "gmLaneDetector::ProcessNearField : Exiting\n" ) ;

}

/**********************
 *
 **********************/

void gmLaneDetector::ProcessFields ( )
{
  gmTRACE_MAC ( "gmLaneDetector::ProcessFields() : Entering\n" ) ;
```

```
  ProcessNearField ( ) ;
  ProcessMidField  ( ) ;
  ProcessFarField  ( ) ;

  gmTRACE_MAC ( "gmLaneDetector::ProcessFields() : Exiting\n" ) ;

}

/**********************
 *
 **********************/

gmLANEINFO * gmLaneDetector::ProcessFrame ( BOOL DisplayResultEnb , BOOL DebugEnb )
{
  gmTRACE_MAC ( "gmLaneDetector::LaneDetect : Entering\n" ) ;


  if ( TestEnable  )
    gmERROR_MAC ( "  gmLaneDetector::ProcessFrame() : Cannot Process normal frame when Test Enabled" )


  FrameNo       = VideoFrameBuffer->GetIdx() ;
  UpdateTimeInfo ( ) ;


  if ( DebugEnb )
    printf ( " ( %8.6f , %8.6f , %8.6f )\n" , SystemInfo->Field.NearRight.Threshold.Level , SystemInfo->F
ield.MidRight.Threshold.Level , SystemInfo->Field.FarRight.Threshold.Level ) ;


  ProcessFields   ( ) ;
  SmoothNewLaneLocation ( 1.00 ) ;

  UpdateSystemInfo ( ) ;

  AdaptParameters ( ) ;


  if ( DisplayResultEnb )
    DisplayResult() ;

  FramesProcessed ++ ;


  gmTRACE_MAC ( "gmLaneDetector::LaneDetect : Exiting\n" ) ;

  return & SystemInfo->CurrentLane ;

}

/**********************
 *
 **********************/

void gmLaneDetector::TestProcess ( )
{
  gmTRACE_MAC ( "gmLaneDetector::TestProcess : Entering\n" ) ;

  if ( ! TestEnable  )
    gmERROR_MAC ( "  gmLaneDetector::ProcessFrame() : Cannot execute Test Process when test not enabled"
)


  FrameNo       = VideoFrameBuffer->GetIdx() ;
  UpdateTimeInfo ( ) ;
  FramesProcessed ++ ;


  gmTRACE_MAC ( "gmLaneDetector::TestProcess : Exiting\n" ) ;
}

/**********************
 *
 **********************
```

```
void gmLaneDetector:: ( )
{
  gmTRACE_MAC ( "gmLaneDetector:: : Entering\n" ) ;

  gmTRACE_MAC ( "gmLaneDetector:: : Exiting\n" ) ;
}

/**********************
 *
 **********************/
```

Appendix B

```
/*
 * file : gmKlugeDetector.h
 * date : 05/19/97
 *
 */

#ifndef GMKLUGEDETECTOR_INCLUDED
#define GMKLUGEDETECTOR_INCLUDED

/*********************
 *
 *********************/

class  gmBuffer          ;
class  gmVideoStream     ;
class  gmViewPort        ;
class  gmDataLogger      ;

/*******************
 *
 ******************/

/* Defined symbolic constants */

#define IM_UNSIGNED 0
#define IM_SIGNED 1
#define IM_FLOAT 2

#define IM_CHAR 0
#define IM_INT 1
/* IM_FLOAT defined as above */
#define IM_LONG 3
#define IM_SHORT 4

#define IM_COLOR "r,g,b"

#define IM_READ 0
#define IM_MODIFY 1

#define gmRAD2DEG  57.29577951308232

/*********************
 *
 *********************/

class gmKlugeDetector
{

private :

  char            * VideoStreamSourceFileName ;

  BOOL              DataLoggingTestEnable ;

  BOOL              FirstPass , ReInit ;

  char            * ExecutionTraceDataDumpPath ;

  RECT              InitialSelectedRegion ;

  RECT              PrevLpRegion ;
  RECT              ThisLpRegion ;

  gmViewPort      * ViewPort ;
  gmBuffer        * ImageBuf ;

  BOOL              LockedOn   ;

  gmDataLogger    * DataLogger ;
  gmTABLEINFO     * DataLogTableInfo ;

  gmBuffer        * VideoFrameBuf ;
  kkIMAGE         * kkRawImage ;
```

```
  RECT                 PlateEvaluationRegion ;
  RECT                 PlateBoundary ;
  gmBUFLOC             PlateCentroid ;

  RECT                 RearShadowEvaluationRegion ;
  RECT                 RearShadowEdge ;

  int                  PlateInitialWidth ;
  int                  RearShadowInitialOffset ;

  int                  PrevPlateULrow , PrevPlateULcol ;
  int                  OldPlateULrow , OldPlateULcol ;
  int                  DeltaPlateULrow , DeltaPlateULcol ;
  int                  NewPlateULrow , NewPlateULcol ;

  int                  CurrentRearShadowRow , PrevRearShadowRow ;

  int                  PlateEvaluationBoundaryMargin ;
  int                  RearShadowEvaluationBoundaryMargin ;

  int                  CurrentPlateWidth ;
  int                  CurrentPlateHeight ;
  int                  CurrentPlateULrow ;
  int                  CurrentPlateULcol ;
  int                  PrevPlateWidth ;


  kkIMAGE          * * PrewRes ;

  int                  FrameIdx ;


  void CreateDataLoggingTable ( );
  double Square ( double x );
  double Spike ( double Scale , double x );
  void DumpKlugeImage2M ( char * FileName , kkIMAGE * Image , int ImageHeight , int ImageWidth );
  void SetSubImage ( kkIMAGE * Image , RECT & SubImageRegion );
  int CloseKKimage ( kkIMAGE * img );
  kkIMAGE * OpenKKimage ( int pixtype , int pixbits , int SubImageHeight , int SubImageWidth );
  kkIMAGE * * Prewitt ( kkIMAGE * RawImage , int RawImageWidth , int RawImageHeight );
  double FindPlate ( int PlateWidth , int * BestULrow , int * BestULcol );
  int    FindRearShadow ( );
  void UpdatePlateParameters ( ) ;

public :

  gmKlugeDetector    ( gmVideoStream * VideoStream ) ;

  ~gmKlugeDetector   ( ) ;

  void ReInitialize ( ) ;

  int AdaptiveFindPlate ( BOOL DebugEnb = FALSE ) ;

  int AdaptiveFindRearShadow ( BOOL DebugEnb = FALSE ) ;

  void ProcessFrame (  int Idx , BOOL DebugEnb = FALSE ) ;


  int      GetCurrentRearShadowRow() { return CurrentRearShadowRow ; } ;
  gmBUFLOC GetPlateCentroid()        { return PlateCentroid ; } ;
  int      GetPlateWidth()           { return CurrentPlateWidth ; } ;

} ;


/*********************
 *
 *********************/

#endif
```

```
/*
 * file : gmKlugeDetector.cpp
 * date : 05/19/97
 *
 *                             -gm
 */

/**********************
 *
 **********************/

#include "MilCons.h"

/**********************
 *
 **********************/

static char LoggerDataFieldsGLB[] = "FrameNo,FrameTimeStamp,ElapsedTime,"\
                                    "CentroidRow,CentroidCol,CentroidVariance" ;

static char LoggerDataTypesGLB[]  = "integer,double,double,"\
                                    "integer,integer,double" ;

/**********************
 *
 **********************/

static char LoggerTestDataFieldsGLB[] = "FrameNo,FrameTimeStamp,ElapsedTime" ;
static char LoggerTestDataTypesGLB [] = "integer,double,double" ;

/**********************
 *
 **********************/

void gmKlugeDetector::CreateDataLoggingTable ( )
{
  gmTRACE_MAC ( "gmKlugeDetector::CreateDataLoggingTable : Entering\n" ) ;


  char * BaseDataLoggingTableName ;


  unsigned n = strlen ( VideoStreamSourceFileName ) ;

  if ( DataLoggingTestEnable )
  {
    BaseDataLoggingTableName = new char [ strlen ( "lpd_" ) + n + strlen ( "_Test" ) + 1 ] ;
    strcpy ( BaseDataLoggingTableName , "lpd_" ) ;
    strcat ( BaseDataLoggingTableName , VideoStreamSourceFileName ) ;
    strcat ( BaseDataLoggingTableName , "_Test" ) ;
    DataLogTableInfo = DataLogger->CreateTable ( BaseDataLoggingTableName , LoggerTestDataFieldsGLB , Log
gerTestDataTypesGLB ) ;
  }

  else
  {
    BaseDataLoggingTableName = new char [ strlen ( "lpd_" ) + n + 1 ] ;
    strcpy ( BaseDataLoggingTableName , "lpd_" ) ;
    strcat ( BaseDataLoggingTableName , VideoStreamSourceFileName ) ;
    DataLogTableInfo = DataLogger->CreateTable ( BaseDataLoggingTableName , LoggerDataFieldsGLB , LoggerD
ataTypesGLB ) ;
  }


  gmTRACE_MAC ( "gmKlugeDetector::CreateDataLoggingTable : Exiting\n" ) ;

}

/*******************
 *
 ******************/

double gmKlugeDetector::Square ( double x )
{
```

```
  gmTRACE_MAC ( "gmKlugeDetector::Square() : Entering\n" ) ;
  double y = x * x ;
  gmTRACE_MAC ( "gmKlugeDetector::Square() : Exiting\n" ) ;
  return y ;
}

/*******************
 *
 ******************/

double gmKlugeDetector::Spike ( double Scale , double x )
{
  gmTRACE_MAC ( "gmKlugeDetector::Spike() : Entering\n" ) ;
  double y = 1.0 / ( 1.0 + ( Scale * x * x ) ) ;
  gmTRACE_MAC ( "gmKlugeDetector::Spike() : Exiting\n" ) ;
  return y ;
}

/*******************
 *
 ******************/

void gmKlugeDetector::DumpKlugeImage2M ( char * FileName , kkIMAGE * Image , int ImageHeight , int ImageW
idth )
{

  gmTRACE_MAC ( "gmKlugeDetector::DumpKlugeImage2M() : Entering\n" ) ;

  FILE * p ;
  float * q ;
  int i , j , k ;
  char FileSpec [256] ;


  if ( Image->pixtype != IM_FLOAT )
  {
    printf("<GordoDumpKlugeImageASCII>: unsupported kkIMAGE pixel type\n");
    exit(0);
  }


  sprintf ( FileSpec , "%s.m" , FileName ) ;


  p = fopen ( FileSpec , "w" ) ;
  q = ( float * ) Image->img ;
  k = 0 ;


  fprintf ( p , "%s=[ ...\n" , FileName ) ;


  for ( i = 0 ; i < ImageHeight ; i ++ )
  {

    for ( j = 0 ; j < ImageWidth ; j ++ )
      fprintf ( p , "%12.6f " , q[k++] ) ;

    fprintf ( p , "\n" ) ;

  }

  fseek ( p , - 1 , SEEK_CUR ) ;

  fprintf ( p , "];\n" ) ;

  fclose ( p ) ;

  gmTRACE_MAC ( "gmKlugeDetector::DumpKlugeImage2M() : Exiting\n" ) ;
}

/*******************
 *
 ******************/
```

```
void gmKlugeDetector::SetSubImage ( kkIMAGE * Image , RECT & SubImageRegion )
{
  gmTRACE_MAC ( "gmKlugeDetector::SetSubImage() : Entering\n" ) ;
  Image->bounds.rs = SubImageRegion.top ;
  Image->bounds.cs = SubImageRegion.left ;
  Image->bounds.re = SubImageRegion.bottom ;
  Image->bounds.ce = SubImageRegion.right ;
  gmTRACE_MAC ( "gmKlugeDetector::SetSubImage() : Exiting\n" ) ;
}

/*******************
 *
 ******************/

int gmKlugeDetector::CloseKKimage ( kkIMAGE * img )
{
  gmTRACE_MAC ( "gmKlugeDetector::CloseKKimage() : Entering\n" ) ;

  INT Status = 0 ;

  if ( ! img )
  {
    Status = 1 ;
    goto EXIT ;
  }

  delete []img->img ;
  delete img ;


EXIT :

  gmTRACE_MAC ( "gmKlugeDetector::CloseKKimage() : Exiting\n" ) ;

  return Status ;

}

/*******************
 *
 ******************/

kkIMAGE * gmKlugeDetector::OpenKKimage ( int pixtype , int pixbits , int SubImageHeight , int SubImageWid
th )
{

  gmTRACE_MAC ( "gmKlugeDetector::OpenKKimage() : Entering\n" ) ;

  kkIMAGE * p = new ( kkIMAGE ) ;


  if ( ! p )
  {
    printf("<OpenKKimage>: failure allocating new kkIMAGE\n");
    exit(0);
  }


  p->bounds.rs = 0 ;
  p->bounds.cs = 0 ;
  p->bounds.re = SubImageHeight - 1 ;
  p->bounds.ce = SubImageWidth  - 1 ;


  p->pixtype = pixtype;


  if (pixtype == IM_FLOAT)
    pixbits = 8 * sizeof(float);
  else if (pixtype == IM_SIGNED)
    pixbits = 8 * sizeof(int);
  else if (pixtype == IM_UNSIGNED)
     pixbits = 8;
```

```
  else
  {
    printf("<OpenKKimage>: unsupported kkIMAGE pixel type\n");
    exit(0);
  }


  p->pixbits = pixbits;
  p->pixtype = pixtype;
  p->img = new unsigned char [ SubImageHeight * SubImageWidth * ( pixbits / 8 ) ] ;


  if ( ! p->img )
  {
    printf("<OpenKKimage>: failure allocating new kkIMAGE pixel buffer\n");
    exit(0);
  }


  gmTRACE_MAC ( "gmKlugeDetector::OpenKKimage() : Exiting\n" ) ;

  return ( p ) ;

}

/*******************
 *
 *******************/

//
//  05/14/98
//  Gordo's version of Kluge's Routine to
//  perform Prewitt image gradient computation
//

kkIMAGE * * gmKlugeDetector::Prewitt ( kkIMAGE * RawImage , int RawImageWidth , int RawImageHeight )
{

  gmTRACE_MAC ( "gmKlugeDetector::Prewitt() : Entering\n" ) ;

  /* static */ kkIMAGE   * * p ;
  int        i , j , k ;
  int        SubImageHeight , SubImageWidth , SubImageVolume , SubImageOffset ;
  unsigned char    * PrevRow , * ThisRow , * NextRow ;
  float    * mbuf , * dbuf , * s0 , * s90 ;
  double     dx , dy ;

  //
  // what gets passed in here is a raw image structure
  // which contains the full extent of the raw image .
  // the kkSUBIMAGE structure defines the boundary of
  // the region of interest within the overall image .
  //
  // hence , all local transform image allocations only
  // need to contain the volume of this sub-image .
  //

  SubImageHeight = RawImage->bounds.re - RawImage->bounds.rs + 1 ;
  SubImageWidth  = RawImage->bounds.ce - RawImage->bounds.cs + 1 ;
  SubImageVolume = SubImageHeight * SubImageWidth ;

  SubImageOffset = RawImage->bounds.rs * RawImageWidth + RawImage->bounds.cs ;

  //
  // we begin by allocate kkIMAGE structures for
  // holding the magnitude and direction
  // transforms of the raw image region bounded
  // by the kkSUBIMAGE parameters.
  //

  p    = new kkIMAGE * [4] ;
  p[0] = OpenKKimage ( IM_FLOAT , 32 , SubImageHeight , SubImageWidth ) ;  // Prewitt magnitude
image
  p[1] = OpenKKimage ( IM_FLOAT , 32 , SubImageHeight , SubImageWidth ) ;  // Prewitt gradient direction
```

```
image
  p[2] = OpenKKimage ( IM_FLOAT , 32 , SubImageHeight , SubImageWidth ) ;  // Prewitt 0-degree spiked
image
  p[3] = OpenKKimage ( IM_FLOAT , 32 , SubImageHeight , SubImageWidth ) ;  // Prewitt 90-degree spiked
image

  //
  // here we get pointers to the
  // base adx of the image buffer
  // for each of the transform images
  //

  mbuf = ( float * ) p[0]->img ;
  dbuf = ( float * ) p[1]->img ;
  s0   = ( float * ) p[2]->img ;
  s90  = ( float * ) p[3]->img ;

  //
  // Fill top transform rows with zeros
  //

  for ( k = 0 ; k < SubImageWidth ; k ++ )
  {
    mbuf[k] = ( float ) 0.0 ;
    dbuf[k] = ( float ) 0.0 ;
    s0  [k] = ( float ) 0.0 ;
    s90 [k] = ( float ) 0.0 ;
  }

  //
  // Fill bottom transform rows with zeros
  //

  for ( k = SubImageVolume - SubImageWidth ; k < SubImageVolume ; k ++ )
  {
    mbuf[k] = ( float ) 0.0 ;
    dbuf[k] = ( float ) 0.0 ;
    s0  [k] = ( float ) 0.0 ;
    s90 [k] = ( float ) 0.0 ;
  }

  //
  // here we perform the Prewitt
  // transform on the raw sub-image
  // producing both a Prewitt gradient magnitude
  // transform image and an image of the
  // Prewitt gradient direction .
  //

  ThisRow = ( unsigned char * ) ( & RawImage->img [ SubImageOffset ] ) ;
  NextRow = ThisRow + RawImageWidth ;
  k = SubImageWidth ;

  for ( i = 1 ; i < SubImageHeight - 1 ; i ++ )
  {

    //
    // we begin by pointing to the raw image
    // row set currently being processed
    //

    PrevRow  = ThisRow ;
    ThisRow  = NextRow ;
    NextRow += RawImageWidth ;

    //
    // set first element of current
    // transform row to zero ,
    // bump transform idx
    //

    mbuf[k] = 0.00 ;
    dbuf[k] = 0.00 ;
    s0  [k] = 0.00 ;
```

```
    s90 [k] = 0.00 ;

    k ++ ;

    //
    // Now transform the current raw sub-image row
    // into magnitude and direction images
    // by scanning from left to right
    //

    for ( j = 1 ; j < SubImageWidth - 1 ; j ++ , k ++ )
    {


      dx = ( double ) ( - ( ( int ) PrevRow[j-1] + ( int ) ThisRow[j-1] + ( int ) NextRow[j-1] )
                        + ( ( int ) PrevRow[j+1] + ( int ) ThisRow[j+1] + ( int ) NextRow[j+1] ) ) ;


      dy = ( double ) (    ( ( int ) PrevRow[j-1] + ( int ) PrevRow  [j] + ( int ) PrevRow[j+1] )
                        - ( ( int ) NextRow[j-1] + ( int ) NextRow  [j] + ( int ) NextRow[j+1] ) ) ;


      mbuf[k] = ( float ) ( sqrt ( ( dx * dx ) + ( dy * dy ) ) ) ;


      if ( mbuf[k] < 0.00001 )
        dbuf[k] = 0.0 ;

            //
      // Normally the following line would be
      // else * dbuf = atan2((double) dy , (double) dx) ;
      // but for the license plate tracking application the following
      // normalization of the angle was needed.
      //

      else
        dbuf[k] = ( float ) ( gmRAD2DEG * acos ( fabs ( cos ( atan2 ( dy , dx ) ) ) ) ) ;

      //
      // now develop both orthogonal
      // 'Spike Enhanced' gradient images
      //

      s0[k]  = ( float ) Spike ( 0.001 , ( double ) dbuf[k] ) ;
      s90[k] = ( float ) Spike ( 0.001 , ( double ) ( 90.00 - dbuf[k] ) ) ;

    }

    //
    // set last element of current
    // transform row to zero ,
    // bump transform idx
    //

    mbuf[k] = 0.00 ;
    dbuf[k] = 0.00 ;
    s0  [k] = 0.00 ;
    s90 [k] = 0.00 ;

    k ++ ;

  }

  gmTRACE_MAC ( "gmKlugeDetector::Prewitt() : Exiting\n" ) ;

  return ( p ) ;

}

/********************
 *
 *******************/
```

```
double gmKlugeDetector::FindPlate ( int PlateWidth , int * BestPlateULrow , int * BestPlateULcol )
{

  gmTRACE_MAC ( "gmKlugeDetector::FindPlate() : Entering\n" ) ;

  int       PrewittWidth , PrewittHeight , PrewittVolume , TemplateOffset ;
  int       i , j , r , c , BestRow , BestCol , PlateHeight ;
  double    BestVal , CorVal ;
  float   * mbuf , * dbuf , * s0buf , * s90buf ;

  //
  // the following order in PrewRes
  // is required :
  //
  //    PrewRes[0] : Prewitt magnitude
  //    PrewRes[1] : Prewitt gradient direction
  //    PrewRes[2] : Prewitt  0-degree spiked gradient direction
  //    PrewRes[3] : Prewitt 90-degree spiked gradient direction
  //

  PrewittWidth  = PrewRes[0]->bounds.ce - PrewRes[0]->bounds.cs + 1 ;
  PrewittHeight = PrewRes[0]->bounds.re - PrewRes[0]->bounds.rs + 1 ;
  PrewittVolume = PrewittWidth * PrewittHeight ;

  /*
   * here we move a licence plate template
   * around on the transform image looking
   * for a best fit .
   *
   * the current template dimensions are
   * what we believe the current licence plate
   * dimensions to be
   *
   * we don't want to scan either the first
   * or last row of the the transform images
   * since they are all zero . this is also
   * true for the first and and last position
   * of each transform image row .
   */


  BestVal = - 1 ;
  PlateHeight = ( PlateWidth >> 1 ) + ( PlateWidth && 0x0001 ) ;


  for ( i = 1 ; i < PrewittHeight - PlateHeight - 1 ; i++ )
  {
    for ( j = 1 ; j < PrewittWidth - PlateWidth - 1 ; j ++ )
    {
      for ( r = 0 , CorVal = 0.00 ; r < PlateHeight ; r ++ )
      {

        TemplateOffset = ( ( i + r ) * PrewittWidth + j ) * sizeof ( float ) ;

        mbuf   = ( float * ) ( & PrewRes[0]->img[TemplateOffset] ) ;
        dbuf   = ( float * ) ( & PrewRes[1]->img[TemplateOffset] ) ;
        s0buf  = ( float * ) ( & PrewRes[2]->img[TemplateOffset] ) ;
        s90buf = ( float * ) ( & PrewRes[3]->img[TemplateOffset] ) ;

        /*
         * If processing rows other than the first and last row of the
         * plate, look at the left and right edge columns only
         */

        if ( ( r != 0 )  &&  ( r != ( PlateHeight - 1 ) ) )
        {
          CorVal += ( double ) ( mbuf[0] * s0buf[0] ) ;
          CorVal += ( double ) ( mbuf [PlateWidth - 1] * s0buf[PlateWidth - 1] ) ;
        }

        /*
         * Otherwise, if looking at the top or bottom row,
         * look at the entire width of the plate
         */
```

```
        else
           for ( c = 0 ; c < PlateWidth ; c ++ )
              CorVal += ( double ) ( mbuf[c] * s90buf[c] ) ;

      }

      /*
       * If a previous location was passed in, weight the result so that
       * points near the previous location are prefered. Have a bigger
       * bias against motion up/down in the image than against motion
       * left/right in the image.
       */

      if ( PrevPlateULrow > 0 )
        CorVal *= (1.000 / (1.000 + 0.001 * Square ( ( double ) ( j - PrevPlateULcol ) ) + 0.003 * Square
 ( ( double ) ( i - PrevPlateULrow ) ) ) ) ;

      /*
       * if we've got a higher correlation
       * than our previous best fit value
       * then save the upper left pixel coord
       * of the current template location .
       */

      if (CorVal > BestVal)
      {
        BestVal = CorVal ;
        BestRow = i ;
        BestCol = j ;
      }

    }

  }


  * BestPlateULrow = BestRow ;
  * BestPlateULcol = BestCol ;


  gmTRACE_MAC ( "gmKlugeDetector::FindPlate() : Exiting\n" ) ;

  return ( BestVal ) ;

}

/**********************
 *
 **********************/

int gmKlugeDetector::FindRearShadow ( )
{

  gmTRACE_MAC ( "gmKlugeDetector::FindRearShadow() : Entering\n" ) ;

  int      i , j , PrewittWidth , PrewittHeight , PrewittVolume , TemplateOffset ;
  int    * BestRowList , BestRowListSize , BestRowListIdx , BestRowListSum , BestRow ;
  double   HighestMagVal ;
  float  * mbuf ;

  //
  // the following order in PrewRes
  // is required :
  //
  //    PrewRes[0] : Prewitt magnitude
  //    PrewRes[1] : Prewitt gradient direction
  //    PrewRes[2] : Prewitt  0-degree spiked gradient direction
  //    PrewRes[3] : Prewitt 90-degree spiked gradient direction
  //

  PrewittWidth  = PrewRes[0]->bounds.ce - PrewRes[0]->bounds.cs + 1 ;
  PrewittHeight = PrewRes[0]->bounds.re - PrewRes[0]->bounds.rs + 1 ;
  PrewittVolume = PrewittWidth * PrewittHeight ;
```

```
/*
 * here we allocate the array which will hold
 * the row of maximum prewitt magnitude for each
 * column evaluated .
 */

BestRowListSize = PrewittWidth >> 1 ;
BestRowList = new int [ BestRowListSize ] ;
BestRowListIdx = 0 ;

/*
 * the way this recognizer works is different from
 * the template locator which finds the best location
 * for the current licence plate .
 *
 * the way we find the current best location for the
 * rear vehicle shadow is conceptually a little simpler .
 *
 * we just scan from top to bottom across every other
 * column in the evaluation region looking for the
 * maximum raw prewitt intensity . based on our assumption
 * that the evaluation region has been appropriately
 * initialized by the operator ( sic ... ) , we will always
 * find the correct single maximum which will be the
 * vehicle's lower rear shadow edge .
 *
 * all the row values are averaged , and the result is
 * returned as the best fit for the current shadow location .
 *
 * once this best value is returned , we don't explicity
 * find the rear shadow for subsequent frames . we'll just
 * get a relative offset to the licence plate centroid and
 * track the rear vehicle shadow relative to that , since
 * we are tracking the licence plate anyway .
 *
 */

for ( j = 1 ; j < PrewittWidth - 1 ; j += 2 )
{

  HighestMagVal = 0.00 ;

  for ( i = 1 ; i < PrewittHeight - 1 ; i ++ )
  {

    TemplateOffset = ( ( i * PrewittWidth ) + j ) * sizeof ( float ) ;

    mbuf   = ( float * ) ( & PrewRes[0]->img[TemplateOffset] ) ;

    if ( * mbuf > HighestMagVal )
    {
      BestRowList[BestRowListIdx] = i ;
      HighestMagVal = * mbuf ;
    }

  }


  BestRowListIdx ++ ;

}

/*
 * now , the Nest fit row of the
 * vehicle's rear shadow loer edge
 * will be the mean of the all
 * the rows which made it into our
 * list of best rows
 */

BestRowListSum = 0 ;

for ( i = 0 ; i < BestRowListIdx ; i ++ )
```

```
  {
    BestRowListSum += BestRowList[i] ;
  }


  BestRow = BestRowListSum / BestRowListSize ;

  /*
   * we're done
   */

  delete []BestRowList ;

  gmTRACE_MAC ( "gmKlugeDetector::FindRearShadow() : Exiting\n" ) ;

  return ( BestRow ) ;

}

/**********************
 *
 **********************/

void gmKlugeDetector::UpdatePlateParameters ( )
{

  gmTRACE_MAC ( "gmKlugeDetector::UpdatePlateParameters() : Entering\n" ) ;

  int MaxAllowableVertMotion = 2 ;
  int MaxAllowableHorzMotion = 2 ;

  CurrentPlateHeight = ( CurrentPlateWidth >> 1 ) + ( CurrentPlateWidth && 0x0001 ) ;

  //
  // if first time thru this code , then
  // we need to initialize the the previous
  // licence plate upper left corner location .
  // we do so by simply setting it equal to
  // the current location .
  //
  // CurrentPlateULrow and CurrentPlateULcol are relative
  // to the UL corner of the EvaluationRegion .
  //

  if ( FirstPass )
    PlateInitialWidth = CurrentPlateWidth ;

  if ( FirstPass  ||  ReInit )
  {
    OldPlateULrow = PlateEvaluationRegion.top  + CurrentPlateULrow ;
    OldPlateULcol = PlateEvaluationRegion.left + CurrentPlateULcol ;
    ReInit = FALSE ;
  }

  NewPlateULrow = PlateEvaluationRegion.top  + CurrentPlateULrow ;
  NewPlateULcol = PlateEvaluationRegion.left + CurrentPlateULcol ;

  //
  // we will allow the UL coordinate to change
  // no more than one row and/or one column per
  // frame .
  //

  DeltaPlateULrow = NewPlateULrow - OldPlateULrow ;
  DeltaPlateULcol = NewPlateULcol - OldPlateULcol ;


  if ( abs ( DeltaPlateULrow ) > MaxAllowableVertMotion )

    if ( DeltaPlateULrow < 0 )
      NewPlateULrow = OldPlateULrow - MaxAllowableVertMotion ;
    else
      NewPlateULrow = OldPlateULrow + MaxAllowableVertMotion ;
```

```
  else
    NewPlateULrow = OldPlateULrow + DeltaPlateULrow ;


  if ( abs ( DeltaPlateULcol ) > MaxAllowableHorzMotion )

    if ( DeltaPlateULcol < 0 )
      NewPlateULcol = OldPlateULcol - MaxAllowableHorzMotion ;
    else
      NewPlateULcol = OldPlateULcol + MaxAllowableHorzMotion ;

  else
    NewPlateULcol = OldPlateULcol + DeltaPlateULcol ;


  PlateBoundary.top  = NewPlateULrow;
  PlateBoundary.left = NewPlateULcol ;

  PlateBoundary.bottom = PlateBoundary.top  + CurrentPlateHeight ;
  PlateBoundary.right  = PlateBoundary.left + CurrentPlateWidth ;

  ViewPort->PaintRectangle ( PlateEvaluationRegion ) ;

  ViewPort->PaintRectangle ( PlateBoundary ) ;


  PlateEvaluationBoundaryMargin = CurrentPlateHeight ;


  PlateEvaluationRegion.top    = PlateBoundary.top  - PlateEvaluationBoundaryMargin ;
  PlateEvaluationRegion.left   = PlateBoundary.left - PlateEvaluationBoundaryMargin ;

  PlateEvaluationRegion.bottom = PlateBoundary.bottom + PlateEvaluationBoundaryMargin ;
  PlateEvaluationRegion.right  = PlateBoundary.right  + PlateEvaluationBoundaryMargin ;

  CurrentPlateULrow = PlateEvaluationBoundaryMargin ;
  CurrentPlateULcol = PlateEvaluationBoundaryMargin ;

  PlateCentroid.Row = ( ( PlateBoundary.top + PlateBoundary.bottom ) >> 1 ) + ( ( PlateBoundary.top + Pla
teBoundary.bottom ) & 0x0001 ) ;
  PlateCentroid.Col = ( ( PlateBoundary.left + PlateBoundary.right ) >> 1 ) + ( ( PlateBoundary.left + Pl
ateBoundary.right ) & 0x0001 ) ;

  OldPlateULrow = NewPlateULrow ;
  OldPlateULcol = NewPlateULcol ;


  gmTRACE_MAC ( "gmKlugeDetector::UpdatePlateParameters() : Exiting\n" ) ;

}

/**********************
 *
 **********************

void gmKlugeDetector::
{
  gmTRACE_MAC ( "gmKlugeDetector::() : Entering\n" ) ;

  gmTRACE_MAC ( "gmKlugeDetector::() : Exiting\n" ) ;
}

/****************************************************************************

                    Following are this class's Public Methods

 ****************************************************************************/

/**********************
 *
 **********************/

gmKlugeDetector::gmKlugeDetector ( gmVideoStream * VideoStream )
{
```

```
  gmTRACE_MAC ( "gmKlugeDetector::gmKlugeDetector() : Entering\n" ) ;


  ViewPort      = VideoStream->GetViewPort() ;
  VideoFrameBuf = VideoStream->GetFrameImageBuf() ;

  kkRawImage    = VideoFrameBuf->Export2KlugeIMAGE ( NULL ) ;

  //
  // here we initialize the licence
  // plate tracking region
  //

  printf ( "   ** Select Initial Licence Plate Evaluation Region :" ) ;

  while ( ! ViewPort->SelectionReady ( ) )
    Sleep ( 50 ) ;

  PlateEvaluationRegion = ViewPort->GetSelectedRegion ( ) ;

  ViewPort->ResetSelection() ;

  printf ( "\n" ) ;

  //
  // here we initialize the rear
  // shadow edge tracking region
  //

  printf ( "   ** Select Initial Rear Shadow Evaluation Region :" ) ;

  while ( ! ViewPort->SelectionReady ( ) )
    Sleep ( 50 ) ;

  RearShadowEvaluationRegion = ViewPort->GetSelectedRegion ( ) ;

  ViewPort->ResetSelection() ;

  printf ( "\n" ) ;


  printf ( "   ** Enter Initial Licence Plate Width : " ) ;
  scanf  ( "%d" , & CurrentPlateWidth ) ;
  printf ( "\n" ) ;

  CurrentPlateULrow = - 1 ;
  CurrentPlateULcol = - 1 ;

  PrevPlateWidth = CurrentPlateWidth ;
  PrevPlateULrow = - 1 ;
  PrevPlateULcol = - 1 ;

  FirstPass = TRUE ;

  AdaptiveFindPlate ( ) ;
  UpdatePlateParameters  ( ) ;
  AdaptiveFindRearShadow ( ) ;


  FirstPass = FALSE ;


  gmTRACE_MAC ( "gmKlugeDetector::gmKlugeDetector() : Exiting\n" ) ;
}

/**********************
 *
 **********************/

void gmKlugeDetector::ReInitialize ( )
{
  gmTRACE_MAC ( "gmKlugeDetector::ReInitialize() : Entering\n" ) ;

  //
```

```
  // here we re-initialize the licence
  // plate tracking parameters
  //

  printf ( "    ** Select New Licence Plate Evaluation Region :" ) ;

  while ( ! ViewPort->SelectionReady ( ) )
    Sleep ( 50 ) ;

  PlateEvaluationRegion = ViewPort->GetSelectedRegion ( ) ;

  ViewPort->ResetSelection() ;


  printf ( "\n" ) ;
  printf ( "    ** Enter New Licence Plate Width : " ) ;
  scanf  ( "%d" , & CurrentPlateWidth ) ;
  printf ( "\n" ) ;


  CurrentPlateULrow = - 1 ;
  CurrentPlateULcol = - 1 ;

  PrevPlateWidth = CurrentPlateWidth ;
  PrevPlateULrow = - 1 ;
  PrevPlateULcol = - 1 ;


  ReInit = TRUE ;


  gmTRACE_MAC ( "gmKlugeDetector::ReInitialize() : Exiting\n" ) ;
}

/********************
 *
 *******************/

int gmKlugeDetector::AdaptiveFindPlate ( BOOL DebugEnb )
{

  gmTRACE_MAC ( "gmKlugeDetector::AdaptiveFindPlate() : Entering\n" ) ;


  int         WidthDelta, br, bc, bi, i, r, c;
  double      BestVal, CorVal;


  SetSubImage ( kkRawImage , PlateEvaluationRegion ) ;

  PrewRes = Prewitt ( kkRawImage , 640 , 480 ) ;

  BestVal = FindPlate ( PrevPlateWidth , & CurrentPlateULrow , & CurrentPlateULcol ) ;



  if ( DebugEnb )
    printf("Img %5d: best UL corner (%3d, %3d) val = %8.2f, width = %2d\n", FrameIdx, CurrentPlateULrow ,
 CurrentPlateULcol , BestVal, PrevPlateWidth);


  PrevPlateULrow = CurrentPlateULrow ;
  PrevPlateULcol = CurrentPlateULcol ;


  WidthDelta = 1;

  /*
   * Only reduce the width if the total perimeter energy goes up
   */

  br = CurrentPlateULrow ;
  bc = CurrentPlateULcol;
```

```
  for (i = PrevPlateWidth - WidthDelta , bi = PrevPlateWidth ; i < PrevPlateWidth ; i ++ )
  {
    CorVal = FindPlate ( i , & r , & c ) ;

    if ( DebugEnb )
      printf ( "   Refine: (%d, %d), val = %f, width = %d\n" , r , c , CorVal , i ) ;


    if (CorVal > BestVal)
    {
      BestVal = CorVal ;
      bi = i ;
      br = r ;
      bc = c ;
    }

  }


  CurrentPlateWidth = bi ;
  CurrentPlateULrow = br ;
  CurrentPlateULcol = bc ;


  /*
   * Only increase the width if the average perimeter energy goes up
   */


  BestVal /= ( 2 * ( CurrentPlateWidth + ( ( CurrentPlateWidth >> 1 ) + ( CurrentPlateWidth && 0x0001 ) )
 - 1 ) ) ;


  for ( i = CurrentPlateWidth , bi = CurrentPlateWidth ; i <= ( CurrentPlateWidth + WidthDelta ) ; i ++ )
  {
    CorVal = FindPlate ( i , & r , & c ) ;

    CorVal /= ( 2 * ( i + ( ( i >> 1 ) + ( i && 0x0001 ) ) - 1 ) ) ;

    if ( DebugEnb )
      printf("   Refine: (%d, %d), val = %f, width = %d\n" , r , c , CorVal , i ) ;

    if ( CorVal > BestVal )
    {
      BestVal = CorVal;
      bi = i ;
      br = r ;
      bc = c ;
    }

  }


  CurrentPlateWidth = bi ;
  CurrentPlateULrow = br ;
  CurrentPlateULcol = bc ;


  PrevPlateWidth = CurrentPlateWidth ;


  CloseKKimage ( PrewRes[0] );
  CloseKKimage ( PrewRes[1] );
  CloseKKimage ( PrewRes[2] );
  CloseKKimage ( PrewRes[3] );
  delete       ( PrewRes );


  gmTRACE_MAC ( "gmKlugeDetector::AdaptiveFindPlate() : Exiting\n" ) ;


  return 0 ;
}
```

```
/**********************
 *
 *********************/

int gmKlugeDetector::AdaptiveFindRearShadow ( BOOL DebugEnb )
{

  gmTRACE_MAC ( "gmKlugeDetector::AdaptiveFindRearShadow() : Entering\n" ) ;


  if ( FrameIdx == 0 )
    FrameIdx = FrameIdx ;


  if ( FirstPass )
  {
    SetSubImage ( kkRawImage , RearShadowEvaluationRegion ) ;
    PrewRes = Prewitt ( kkRawImage , 640 , 480 ) ;
    CurrentRearShadowRow = FindRearShadow ( ) + RearShadowEvaluationRegion.top ;

    RearShadowInitialOffset = CurrentRearShadowRow - PlateCentroid.Row ;

    PrevRearShadowRow = CurrentRearShadowRow ;

    CloseKKimage ( PrewRes[0] );
    CloseKKimage ( PrewRes[1] );
    CloseKKimage ( PrewRes[2] );
    CloseKKimage ( PrewRes[3] );
    delete       ( PrewRes );

    goto MAIN_EXIT ;
  }

  /*
   * we arrive here only when the system is
   * running smoothly and not on a first video
   * frame . we're just going to update the
   * shadow coordinate parameters based on
   * a linear scaling of the initial values .
   */

  CurrentRearShadowRow = PlateCentroid.Row + ( int ) ( ( ( ( double ) RearShadowInitialOffset ) * ( ( dou
ble ) CurrentPlateWidth ) / ( ( double ) PlateInitialWidth ) ) + 0.500  ) ;

  PrevRearShadowRow = CurrentRearShadowRow ;


MAIN_EXIT :

  RearShadowEvaluationRegion.top    = CurrentRearShadowRow ;
  RearShadowEvaluationRegion.left   = PlateCentroid.Col - 15 ;
  RearShadowEvaluationRegion.bottom = CurrentRearShadowRow + 1 ;
  RearShadowEvaluationRegion.right  = PlateCentroid.Col + 15 ;


  if ( DebugEnb )
    printf("Img %5d: best fit row : %3d\n", FrameIdx , CurrentRearShadowRow ) ;


  gmTRACE_MAC ( "gmKlugeDetector::AdaptiveFindRearShadow() : Exiting\n" ) ;


  return 0 ;

}

/**********************
 *
 *********************/

void gmKlugeDetector::ProcessFrame ( int Idx , BOOL DebugEnb )
{
  gmTRACE_MAC ( "gmKlugeDetector::() : Entering\n" ) ;
```

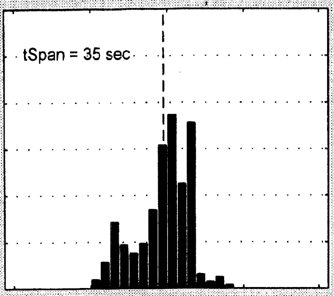
```
  FrameIdx = Idx ;

  AdaptiveFindPlate       ( DebugEnb ) ;
  UpdatePlateParameters   ( ) ;

  AdaptiveFindRearShadow ( DebugEnb ) ;
  ViewPort->PaintRectangle ( RearShadowEvaluationRegion ) ;

  ReInit     = FALSE ;

  gmTRACE_MAC ( "gmKlugeDetector::() : Exiting\n" ) ;
}

/**********************
 *
 **********************

void gmKlugeDetector::
{
  gmTRACE_MAC ( "gmKlugeDetector::() : Entering\n" ) ;

  gmTRACE_MAC ( "gmKlugeDetector::() : Exiting\n" ) ;
}

/**********************
 *
 **********************/
```

Appendix C

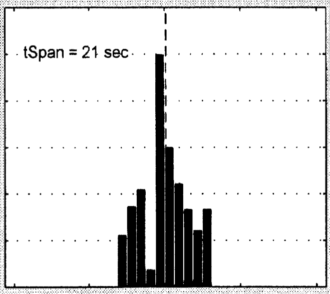
ID	**Channel Name**	**Data Type**
1	xTimeStamp	Double Precision
2	PlateCentroidRow	Integer
3	PlateCentroidCol	Integer
4	PlateWidth	Integer
5	LeftNearRow	Integer
6	LeftNearCol	Integer
7	RightNearRow	Integer
8	RightNearCol	Integer
9	LeftMidRow	Integer
10	LeftMidCol	Integer
11	RightMidRow	Integer
12	RightMidCol	Integer
13	LeftFarRow	Integer
14	LeftFarCol	Integer
15	ShadowRow	Integer
16	LaneWidth	Integer
17	LaneOffset	Integer
18	LaneCenter	Integer
19	PixelWidthFt	Double Precision
20	y	Double Precision
21	Headway	Double Precision
22	InValid	Byte

Appendix D

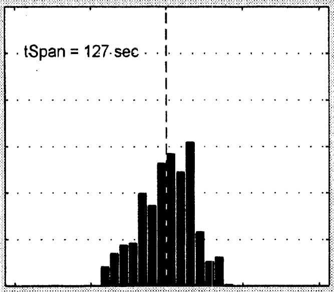
Ensembles 1 - 15



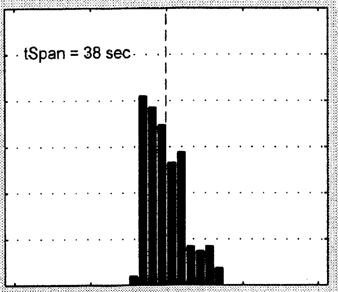
t0_02537_c1_04



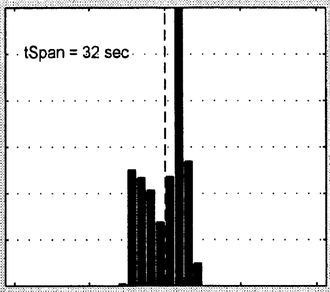
t0_02950_c1_00



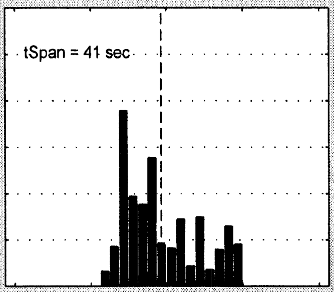
t0_03113_c1_00



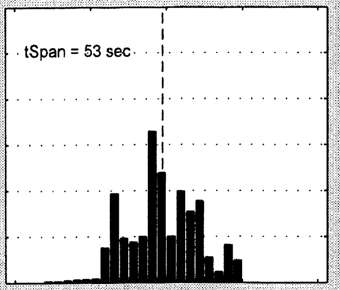
t0_03608_c1_01



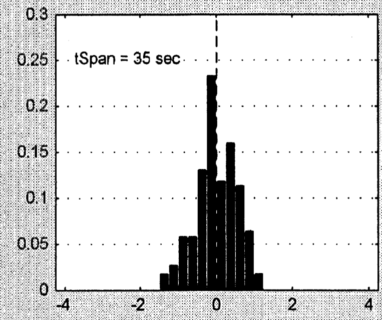
t0_04000_c1_03



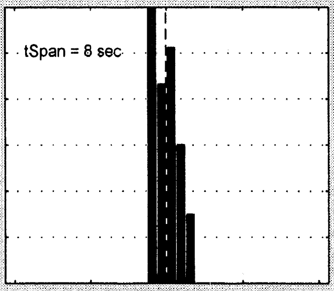
t0_10326_c1_00



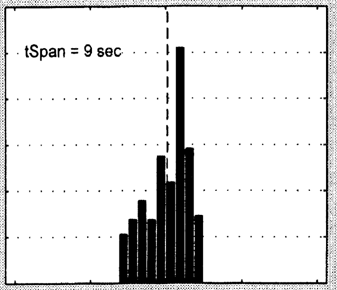
t0_10658_c1_00



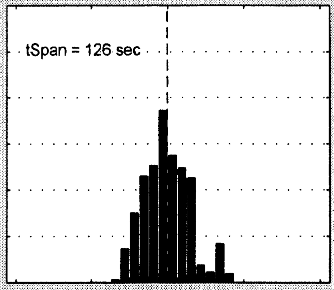
t2_12040_c3_02



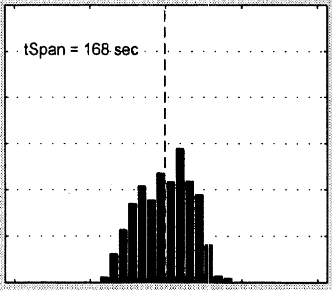
t3_02703_c1_00



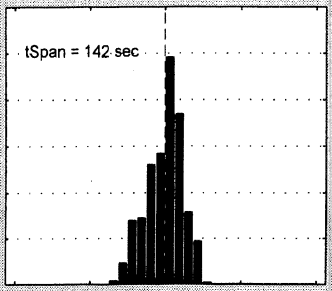
t3_02738_c1_00



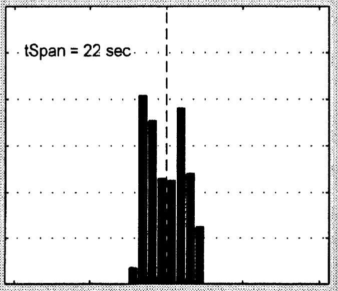
t3_03652_c1_34



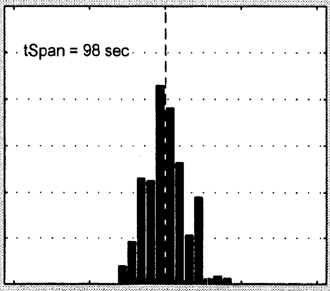
t3_05040_c1_01



t3_10900_c1_07

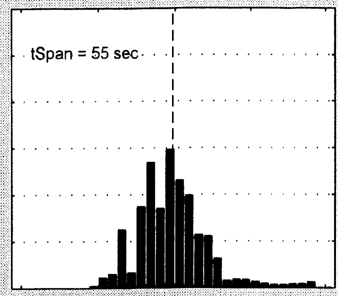


t3_11740_c1_02

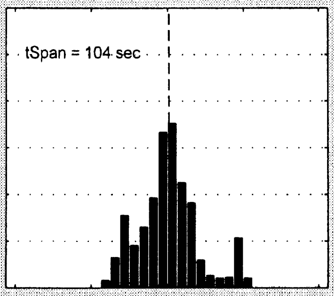


t3_12245_c1_00

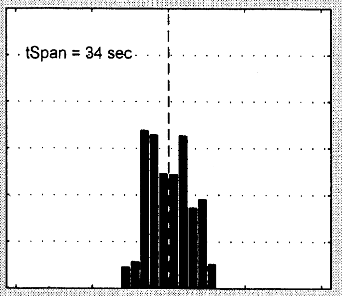
Ensembles 16 - 30



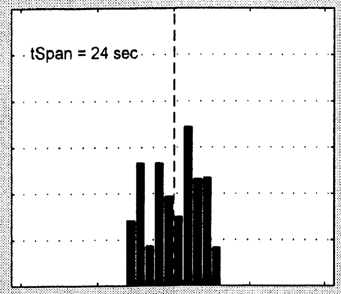
t0_00301_c1_00



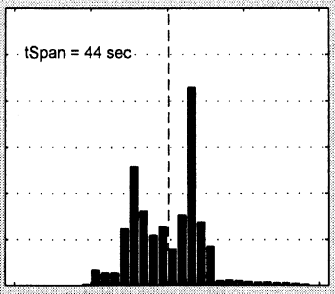
t0_00520_c1_00



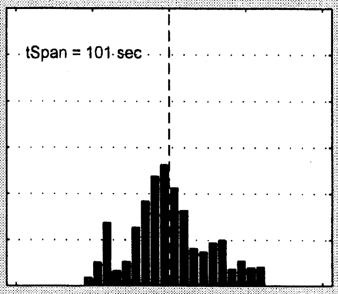
t0_01607_c1_00



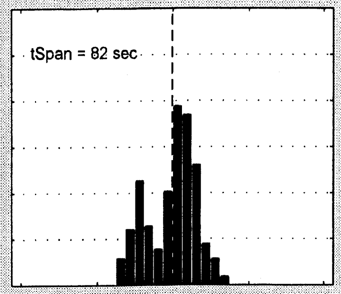
t0_01730_c1_00



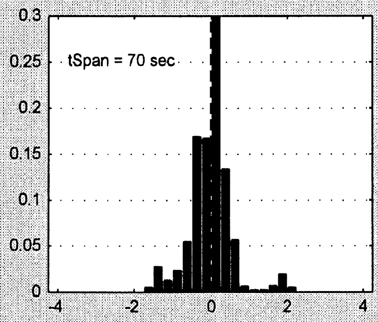
t0_01953_c1_00



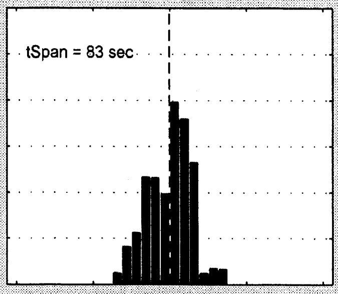
t0_02844_c1_00



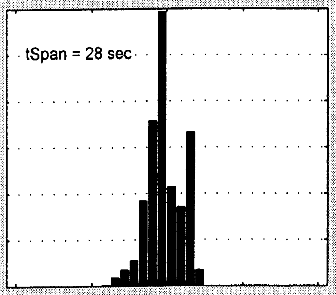
t0_04605_c1_00



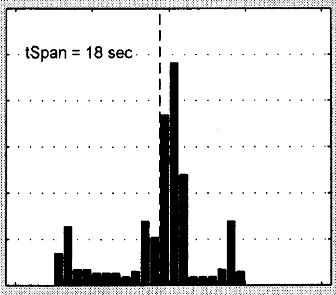
t0_04940_c1_00



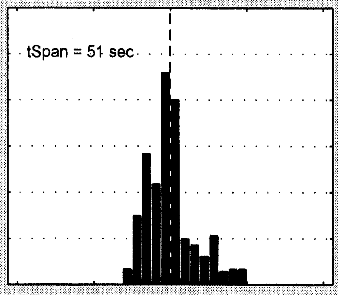
t0_05343_c1_01



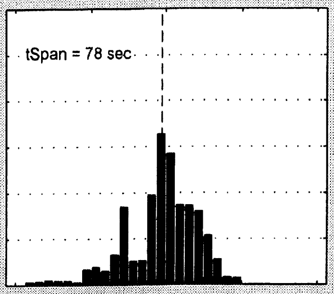
t1_01700_c1_02



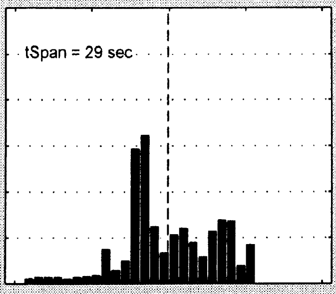
t1_03019_c1_01



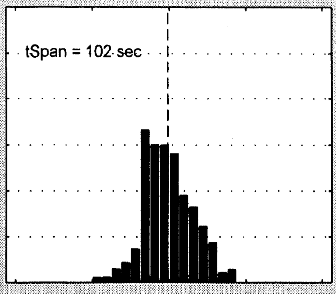
t1_03415_c1_00



t1_05414_c1_00

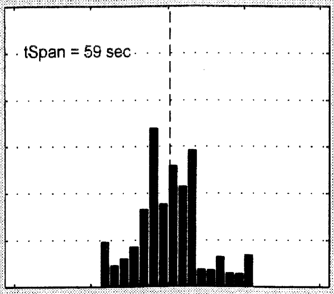


t1_05721_c1_01

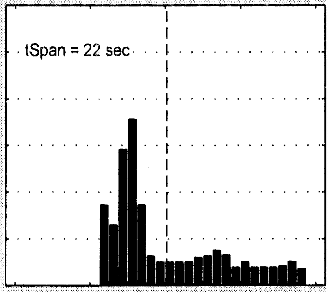


t1_10220_c1_06

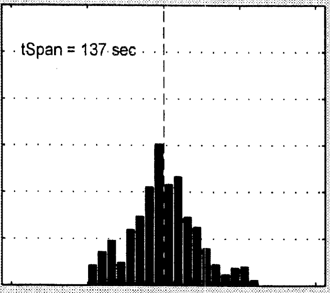
Ensembles 31 - 45



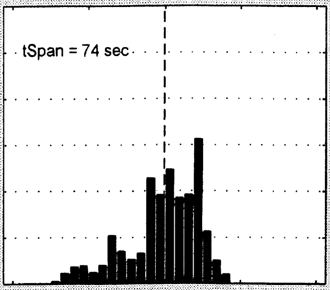
t1_10704_c1_00



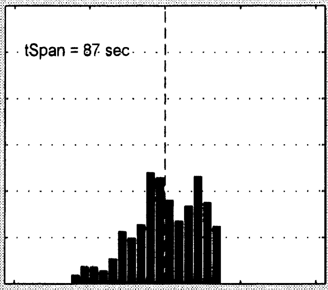
t1_12146_c1_02



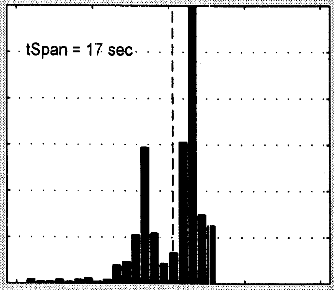
t1_13542_c1_00



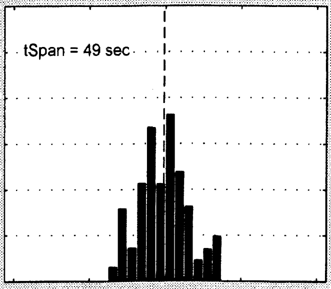
t1_13845_c1_00



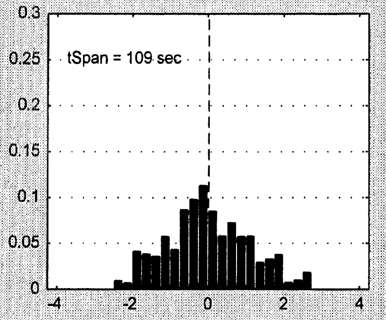
t1_14647_c1_00



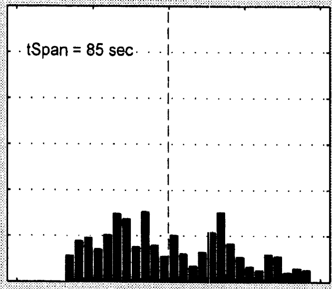
t1_15047_c1_00



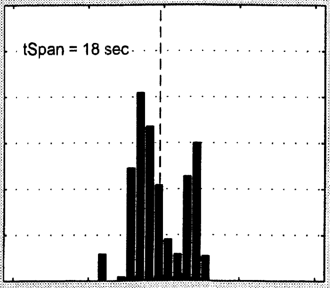
t1_15215_c1_00



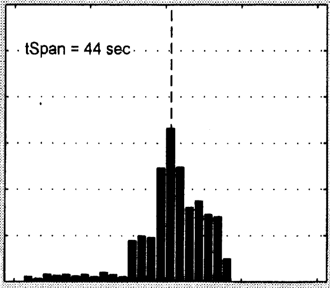
t2_04905_c1_00



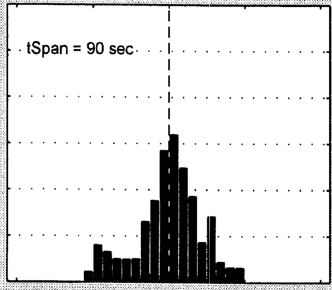
t2_05214_c1_00



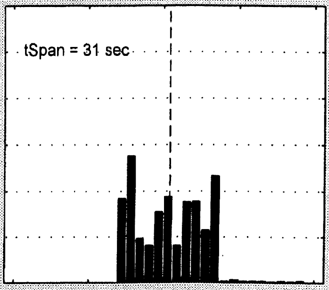
t2_05600_c1_00



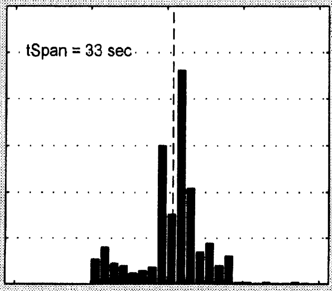
t2_10508_c1_00



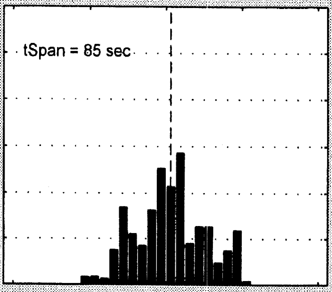
t2_13735_c1_02



t3_10000_c1_00



t3_10826_c1_02



t3_11100_c1_00