Question(s):	12	**Meeting, date:**	Geneva, February 2-6, 2009
Study Group:	9 **Working Party:**	**Intended type of document** (R-C-TD):	C
Source:	USA (Proposed)		
Title:	Fast Low Bandwidth Video Quality Model (VQM) Description and Reference Code		

Contact:	Stephen Wolf	Tel: +1 303-497-3771
	NTIA USA	Fax: +1 303-497-5969 Email: swolf@its.bldrdoc.gov
Contact:	Margaret Pinson	Tel: +1 303-497-3579
	NTIA USA	Fax: +1 303-497-5969 Email: mpinson@its.bldrdoc.gov

Introduction

Study Group 9 has been working on finalizing Draft New Recommendation J.redref which specifies Reduced Reference (RR) methods for performing video quality measurements of digital television systems. The U.S. National Telecommunications and Information Administration (NTIA) recently submitted two Video Quality Models (VQMs) to the recent Video Quality Experts Group (VQEG) RR Television (RRTV) evaluation tests. One of the VQMs (i.e., the Fast Low Bandwidth VQM) was in the top performing group for both 525-line and 625-line video systems and also significantly outperformed Peak Signal to Noise Ratio (PSNR), a full reference VQM. This contribution describes the NTIA Fast Low Bandwidth VQM algorithm and provides reference code for its implementation. While NTIA holds multiple U.S. patents on this algorithm, NTIA/ITS has made and is willing to make these freely available to all interested parties for both non-commercial and commercial purposes.

Proposal

We propose that this contribution be included as a normative Annex in Draft New Recommendation J.redref.

1. Background

In the 2003-2004 time frame, the U.S. National Telecommunications and Information Administration (NTIA) developed two video quality models (VQMs) with a Reduced Reference (RR) bandwidth of approximately 12 to 14 kbits/sec for ITU-R Recommendation BT.601 (Rec. 601) sampled video [1]. These models were called the "Low Bandwidth VQM" and "Fast Low Bandwidth VQM". The Fast Low Bandwidth VQM was a computationally efficient version of the Low Bandwidth VQM. The Fast Low Bandwidth VQM is about 4 times faster since it extracts spatial features from video frames that are first pre-averaged, rather than extracting spatial features directly from the Rec. 601 video frames. Additional computational savings for the Fast Low Bandwidth VQM resulted from computing temporal information (i.e., motion) features based on a random sub-sampling of pixels in the luminance Y channel rather than using all pixels in all three video channels (Y, Cb, and Cr). Both VQMs have been available in our VQM software tools for several years and may be freely used for both commercial and non-commercial applications. Binary executable versions of these VQM tools and their associated source code are available for download [2].

Since NTIA wanted to submit both the Low Bandwidth and Fast Low Bandwidth VQMs to the Reduced Reference TV (RRTV) tests for independent evaluation by the Video Quality Experts Group (VQEG), NTIA choose to submit them to different bit rate categories even though they have identical RR bit rate requirements. NTIA chose to submit the Low Bandwidth VQM to the 256k category and the Fast Low Bandwidth VQM to the 80k category since the performance of the Low Bandwidth VQM was expected to be superior to that of the Fast Low Bandwidth VQM. Both VQMs utilized the NTIA RR calibration algorithm which is included in ITU-T Recommendation J.244 [3]. This calibration algorithm requires approximately 22 to 24 kbits/sec of RR bandwidth to produce estimates for temporal delay, spatial shift, spatial scaling, gain, and level offset.

An interesting result from the VQEG RRTV evaluation tests [4] was that the Fast Low Bandwidth VQM outperformed the Low Bandwidth VQM for both the 525-line and 625-line test. This is an interesting result since it implies that extracting spatial features from averaged frames is superior to extracting them from non-averaged frames. Whether or not this result will prove true for other data sets is an area for further research. At this time, NTIA does not see a reason to standardize both models so this Annex only describes the Fast Low Bandwidth VQM.

2. Introduction

This Annex presents a description and reference code for the NTIA Fast Low Bandwidth VQM. The NTIA Fast Low Bandwidth VQM utilizes techniques similar to those of the NTIA General VQM, a description of which can be found in both ITU-T Recommendation J.144 [5] and ITU-R Recommendation BT.1683 [6]. The Fast Low Bandwidth VQM uses RR features with much less bandwidth than the NTIA General VQM, but the feature extraction and comparison process is similar for both. For ITU-R Recommendation BT.601 sampled video [1], the Fast Low Bandwidth VQM uses RR features that require approximately 12 to 14 kbits/sec of transmission bandwidth. This Annex only describes the Fast Low Bandwidth VQM since its complementary low bandwidth calibration algorithms are fully documented in ITU-T Recommendation J.244 [3]. However, for completeness, the reference code in this Annex includes both the Fast Low Bandwidth VQM and its associated low bandwidth calibration algorithms. The reference code also contains example quantization functions for the features used by the low bandwidth calibration (these quantization functions are not part of ITU-T Recommendation J.244).

3. Fast Low Bandwidth VQM Description

3.1 VQM Overview

The VQM description will encompass three primary areas: (1) the low bandwidth features that are extracted from the original and processed video streams, (2) the parameters that result from comparing like original and processed feature streams, and (3) the VQM calculation that combines the various parameters, each of which measures a different aspect of video quality. This description makes use of readily available references for the technical details.

3.2 Feature Description

3.2.1 Feature Overview

Three types of features are used by the Fast Low Bandwidth VQM: color, spatial, and temporal. Each of these feature types quantify perceptual distortions in their respective domains. The reference code subroutine "model_fastlowbw_features" provides a complete mathematical description of the features used by the Fast Low Bandwidth VQM.

3.2.2 Color Features

The color features are the same f_{COHER_COLOR} features that are used by the NTIA General VQM. These features are described in detail in Annex D.7.3 of ITU-T Recommendation J.144. The f_{COHER_COLOR} features provide a 2-dimensional vector measurement of the amount of blue and red chrominance information (C_B, C_R) in each S-T region. Thus, the f_{COHER_COLOR} features are sensitive to color distortions. The f_{COHER_COLOR} features of the NTIA Fast Low Bandwidth VQM are extracted from spatial-temporal (S-T) region sizes of 30 vertical lines by 30 horizontal pixels by 1 second of time (i.e., 30 x 30 x 1s) whereas the NTIA General VQM used S-T region sizes of 8 x 8 x 1 frame.

3.2.3 Spatial Features

The spatial features are the same f_{SI13} and f_{HV13} features that are used by the NTIA General VQM. These features are described in detail in Annex D.7.2.2 of ITU-T Recommendation J.144. The f_{SI13} and f_{HV13} features measure the amount and angular distribution of spatial gradients in spatial-temporal (S-T) sub-regions of the luminance (Y) images. Thus, the f_{SI13} and f_{HV13} features are sensitive to spatial distortions such as blurring and blocking.

The f_{SI13} and f_{HV13} features of the NTIA Fast Low Bandwidth VQM are extracted from spatial-temporal (S-T) region sizes of 30 vertical lines by 30 horizontal pixels by 1 second of time (i.e., 30 x 30 x 1s) whereas the NTIA General VQM used S-T region sizes of 8 x 8 x 0.2s. In addition, to save computations, the 1 second of luminance Y images are first pre-averaged across time before applying the two-dimensional 13 x 13 edge enhancement filters given in Annex D.7.2.1.

One additional spatial feature is extracted from the 1 second of pre-averaged luminance (Y) images. This feature is the *mean* luminance (Y) level of each 30 x 30 x 1s S-T region (denoted here as f_{MEAN}). The purpose of the f_{MEAN} feature is to provide a luminance-level perceptual weighting function for spatial information (SI) loss as measured by the f_{SI13} and f_{HV13} features. This will be described in the Parameter Description Section.

3.2.4 Temporal Features

Powerful estimates of perceived video quality can be obtained from the color and spatial feature set described above. However, since the S-T regions from which these features are extracted span many video frames (i.e., 1 second of video frames), they tend to be insensitive to brief temporal disturbances in the picture. Such disturbances can result from noise or digital transmission errors; and, while brief in nature, they can have a significant impact on the perceived picture quality. Thus, a temporal-based RR feature is used to quantify the perceptual effects of temporal disturbances. This feature measures the absolute temporal information (ATI), or motion, in the luminance Y image plane and is computed as:

$$f_{ATI} = rms\left\{rand5\%\left|Y(t) - Y(t-0.2s)\right|\right\}$$

For computational efficiency, *Y* is randomly sub-sampled to contain only 5% of the image pixels (represented here by the *rand5%* function). The sub-sampled *Y* image at time *t*-0.2s is subtracted from the identically sub-sampled *Y* image at time *t* and the root mean square error (*rms*) of the result is used as a measure of ATI. Using the convention found in Annex D.8 of ITU-T Recommendation J.144, this will also be denoted as:

$$f_{ATI} \cong \text{Y\_rand5\%\_ati0.2s\_rms}$$

The feature f_{ATI} is sensitive to temporal disturbances. For 30 fps video, 0.2s is 6 video frames while for 25 fps video, 0.2s is 5 video frames. Subtracting images 0.2s apart makes the feature insensitive to real time 30 fps and 25 fps video systems that have frame update rates of at least 5 fps. The quality aspects of these low frame rate video systems, common in multimedia applications, are sufficiently captured by the f_{SI13}, f_{HV13}, and f_{COHER_COLOR} features. The 0.2s spacing is also more closely matched to the peak temporal response of the human visual system than differencing two images that are one frame apart in time.

Figure 1 provides an example plot of the f_{ATI} feature for a original (solid blue) and processed (dashed red) video scene from a digital video system with transient burst errors in the digital transmission channel. Transient errors in the processed picture create spikes in the f_{ATI} feature. The bandwidth required to transmit the f_{ATI} feature is extremely low since it requires only 30 samples per second for 30 fps video. Other types of additive noise in the processed video, such as might be generated by an analog video system, will appear as a positive DC shift in the time history of the processed feature stream with respect to the original feature stream. Video coding systems that eliminate noise will cause a negative DC shift.

Before extracting a transient error parameter from the f_{ATI} feature streams shown in Figure 1, it is advantageous to increase the width of the motion spikes (red spikes in Figure 1). The reason is that short motion spikes from transient errors do not adequately represent the perceptual impact of these types of errors. One method for increasing the width of the motion spikes is to apply a maximum filter to both the original and processed feature streams before calculation of the error parameter function between the two waveforms. For the f_{ATI} based error parameter, a 7-point wide maximum filter (which will be denoted here as the *max7pt* function) was used that produces an output sample at each frame that is the maximum of itself and the 3 nearest neighbors on each side (i.e., earlier and later time samples).

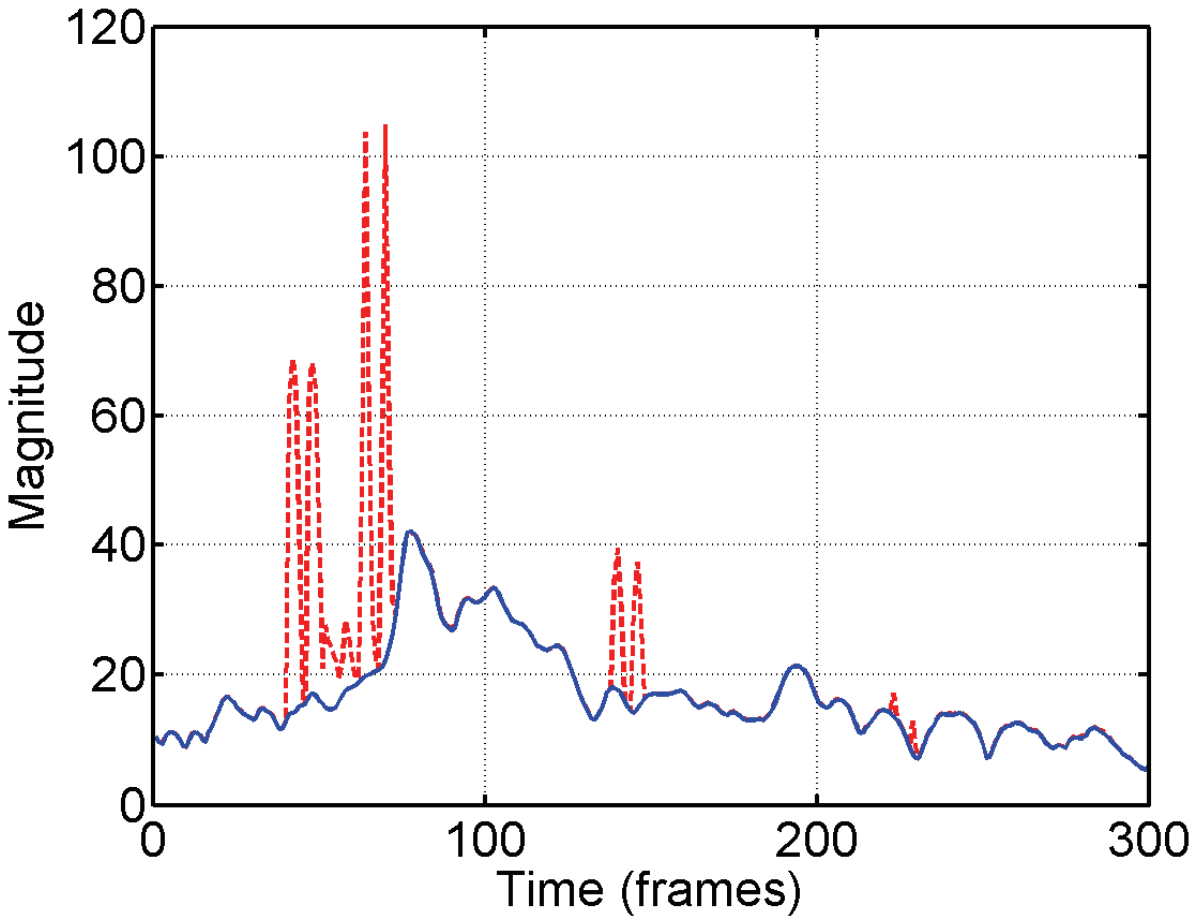

Figure 1. Example time history of f_{ATI} feature.

3.2.5 Quantization of Features

Quantization to 9 bits of accuracy is sufficient for the Y_{MEAN}, f_{SI13}, f_{HV13}, and f_{COHER_COLOR} features, while the f_{ATI} feature should be quantized to 10 bits. To have minimal effect on the video quality parameter calculations, a non-linear quantizer design should be used where the quantizer error is proportional to the magnitude of the signal being quantized. Very low values are uniformly quantized to some cutoff value, below which there is no useful quality assessment information. Such a quantizer design minimizes the error in the corresponding parameter calculations because these calculations are normally based on an error ratio or log ratio of the processed and original feature streams (see the Parameter Description Section given below).

Figure 2 provides a plot of the 9-bit non-linear quantizer used for the f_{SI13} original feature. The reference code subroutine "model_lowbw_compression" provides a complete mathematical description of the recommended quantizers used by the Fast Low Bandwidth VQM. If the features fall outside the range of the recommended quantizers on the low or high end (highly unlikely), then the S-T parameters derived from these features are zeroed so they do not influence the overall VQM.

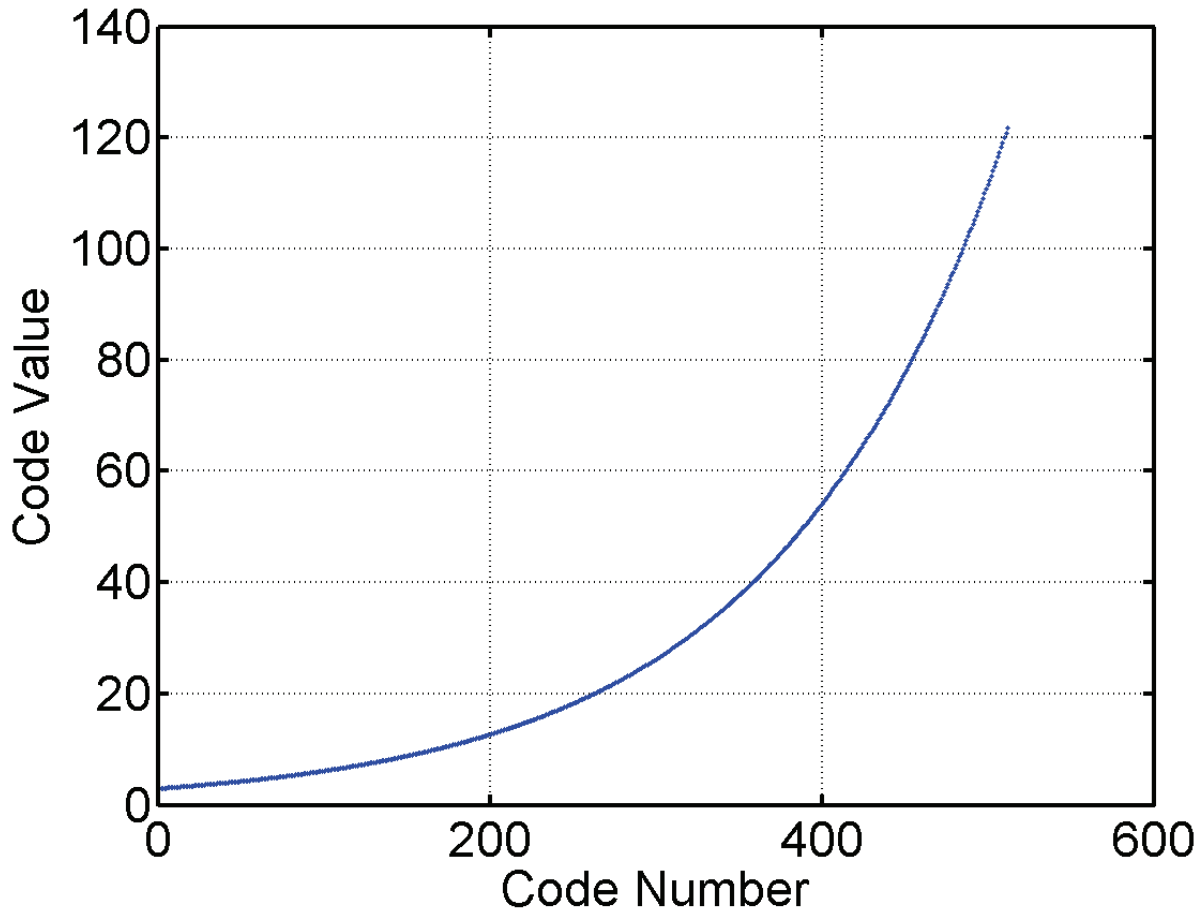

Figure 2. Non-linear 9-bit quantizer for the f_{SI13} feature.

3.3 Parameter Description

3.3.1 Parameter Overview

Several steps are involved in the calculation of parameters that track the various perceptual aspects of video quality. The steps may involve (1) applying a perceptual threshold to the extracted features from each S-T sub-region, (2) calculating an error function between processed features and corresponding original features, and (3) pooling the resultant error over space and time. See Annex D.8 of ITU-T Recommendation J.144 for a detailed description of these techniques and their accompanying mathematical notation for parameter names, which will also be used here. The reference code subroutine "model_fastlowbw_parameters" provides a complete mathematical description of the parameters used by the Fast Low Bandwidth VQM. For simplicity, the description of the parameters in this section does not consider the effects of feature quantization (e.g., handling feature values that might lie outside of the recommended quantization ranges).

3.3.2 New Methods

This section will summarize new methods that have been found to improve the objective to subjective correlation of parameters based on RR features with very low transmission bandwidths

such as those utilized for the NTIA Fast Low Bandwidth VQM (i.e., new methods not found in ITU-T Recommendation J.144). It is worth noting that no improvements have been found for the basic form of the parameter error functions given in Annex D.8.2.1 of ITU-T Recommendation J.144. The two error functions that consistently produce the best parameter results (for spatial and temporal parameters) are a logarithmic ratio $\{\log10\ [f_p(s,t) / f_o(s,t)]\}$ and an error ratio $\{[f_p(s,t) - f_o(s,t)] / f_o(s,t)\}$, where $f_p(s,t)$ and $f_o(s,t)$ are the processed feature and corresponding original feature extracted from the S-T region with spatial coordinates s and temporal coordinates t, respectively. Errors must be separated into gains and losses, since humans respond differently to additive (e.g., blocking) and subtractive (e.g., blurring) impairments. Applying a lower perceptual threshold to the features before application of these two error functions prevents division by zero.

After computation of the S-T parameters using one of the error functions, then the S-T parameters must be pooled over space and time to produce a parameter value for the video clip. This error pooling can occur in multiple stages (e.g., over space and then over time). One new error pooling method that is utilized by the Fast Low Bandwidth VQM is called Macro-Block (MB) error pooling. MB error pooling groups a contiguous number of S-T sub-regions and applies an error pooling function to this set. For instance, the function denoted as "MB(3,3,2)max" will perform a max function over parameter values from each group of 18 S-T sub-regions that are stacked 3 vertical by 3 horizontal by 2 temporal. For the 32 x 32 x 1s S-T sub-regions of the f_{SI13}, f_{HV13}, and f_{COHER_COLOR} features described above, each MB(3,3,2) region would encompass a portion of the video stream that spans 96 vertical lines by 96 horizontal pixels by 2 seconds of time. MB error pooling has been found to be useful in tracking the perceptual impact of impairments that are localized in space and time. Such localized impairments often dominate the quality decision process. MB error pooling can also be implemented as a filtering process so that instead of producing a single output value for each MB, each S-T sample is replaced with its MB filtered value, where the MB is centered on the S-T sample. This is called Overlapped MB (OMB) error pooling.

A second error pooling method is a generalized Minkowski(P,R) summation, defined as:

$$Minkowski(P,R) = \sqrt[R]{\frac{1}{N}\sum_{i=1}^{N}|v_i|^P}$$

Here v_i represents the parameter values that are included in the summation. This summation might, for instance, include all parameter values at a given instance in time (spatial pooling), or may be applied to the macro-blocks described above. The Minkowski summation where the power P is equal to the root R has been used by many developers of video quality metrics for error pooling. The generalized Minkowski summation, where $P \neq R$, provides additional flexibility for linearizing the response of individual parameters to changes in perceived quality. This is a necessary step before combining multiple parameters into a single estimate of perceived video quality, which is performed with a linear fit.

3.3.3 Color Parameters

Two parameters are extracted from the f_{COHER_COLOR} features. One of these parameters, called *color_extreme*, measures extreme color distortions that might be caused by colored blocks from transmission errors. The other parameter, called *color_spread*, provides an indication of the variance or spread in the color errors. Rather than using the Euclidean distance measure to quantify distortions (as in Annex D.8.2.2 of ITU-T Recommendation J.144), both of these parameters use the square root of the Manhattan distance. Following the mathematical notation in Annex D.8.2.2 of ITU-T Recommendation J.144, where $f_p(s,t)$ and $f_o(s,t)$ represent the 2-dimensional f_{COHER_COLOR}

feature extracted from a S-T region of the processed and original video streams, this feature comparison function is given by:

$$sqrtmanhat(s,t) = \sqrt{\sum_{C_B,C_R} \left| \underline{f}_p(s,t) - \underline{f}_o(s,t) \right|}$$

The Manhattan distance measure seems to be better than the Euclidean distance measure and the square root function is required to linearize the parameter's response to quality changes. Following the mathematical notations in Annex D.8 of ITU-T Recommendation J.144, the color parameters are given by:

$$color\_extreme = \text{color\_coher\_color\_30x30\_1s\_mean\_sqrtmanhat\_OMB(3,3,2)above99\%\_Minkosski(0.5,1)}$$

$$color\_spread = \text{color\_coher\_color\_30x30\_1s\_mean\_sqrtmanhat\_OMB(3,3,2)Minkosski(2,4)\_90\%}$$

A combined color parameter (*color_comb*) that contains the optimal combination of both the *color_extreme* and *color_spread* parameters is then computed as:

$$color\_comb = 0.691686 * color\_extreme - 0.617958 * color\_spread$$

This positive valued *color_comb* parameter is then clipped at the low end, which is represented mathematically by (following the notation in Annex D.8.5 of ITU-T Recommendation J.144):

$$color\_comb = color\_comb\_clip\_0.114$$

This *color_comb* parameter is included in the linear combination for the VQM calculation.

3.3.4 Spatial Parameters

Two spatial parameters are computed from the f_{SI13} feature, one that measures a loss of spatial information (*si_loss*) and one that measures a gain of spatial information (*si_gain*). Following the mathematical notation in Annex D.8 of ITU-T Recommendation J.144, these parameters are given by:

$$si\_loss = \text{avg1s\_Y\_si13\_30x30\_std\_3\_ratio\_loss\_OMB(3,3,2)Minkosski(1,2)\_Minkosski(1.5,2.5)\_clip\_0.12}$$

$$si\_gain = \text{avg1s\_Y\_si13\_30x30\_std\_3\_log\_gain\_clip\_0.1\_above95\%tail\_Minkosski(1.5,2)}$$

As the mean luminance (Y) level of the S-T sub-region increases (i.e., as measured by the f_{MEAN} feature), the ability to perceive changes in spatial detail (e.g., such as blurring measured by *si_loss*) decreases. This can be accommodated by introducing a weighting function (*Y_weight*) as shown in Figure 3 to the *si_loss* values from each S-T sub-region (i.e., the *si_loss* values after the ratio loss comparison function is performed on each S-T sub-region but before the spatial and temporal collapsing functions). The weighting function *Y_weight* is equal to one (i.e., full weighting) until an average luminance level of 175 is reached and then it decreases linearly to zero as the luminance values increase from 175 to 255. This intermediate correction is applied only to the *si_loss* values, not the *si_gain* values.

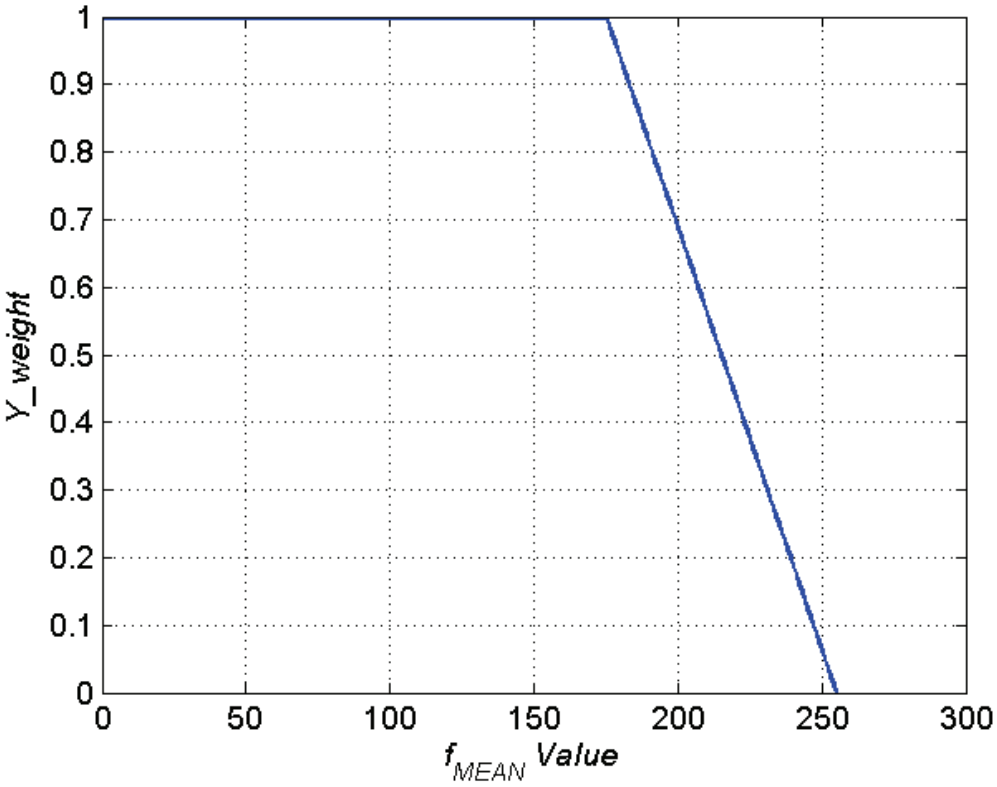

Figure 3. Weighting function *Y_weight* for modifying *si_loss* S-T parameters.

Two spatial parameters are computed from the f_{HV13} feature, one that measures a loss of relative Horizontal and Vertical (HV) spatial information (*hv_loss*) and one that measures a gain (*hv_gain*). Following the mathematical notation in Annex D.8 of ITU-T Recommendation J.144, these parameters are given by:

$$hv\_loss = \text{avg1s\_Y\_hv13\_angle0.225\_rmin20\_30x30\_mean\_4\_ratio\_loss\_...}$$
$$\text{OMB(3,3,2)below1\%\_Minkosski(1,1.5)\_clip\_0.08}$$

$$hv\_gain = \text{avg1s\_Y\_hv13\_angle0.225\_rmin20\_30x30\_mean\_4\_log\_gain\_...}$$
$$\text{clip\_0.06\_OMB(3,3,2)above99\%tail\_Minkosski(1.5,3)}$$

Not shown in the above equations is that the *Y_weight* function shown in Figure 3 is also applied to both the *hv_loss* and *hv_gain* values from each S-T sub-region before the spatial and temporal collapsing functions (after the ratio_loss and log_gain computations, respectively). An additional weighting function (*SI_weight* as shown in Figure 4) is applied to the *hv_loss* values from each S-T sub-region. This was necessary to reduce the sensitivity of *hv_loss* for S-T regions that have very little spatial information (i.e., low f_{SI13} original feature values).

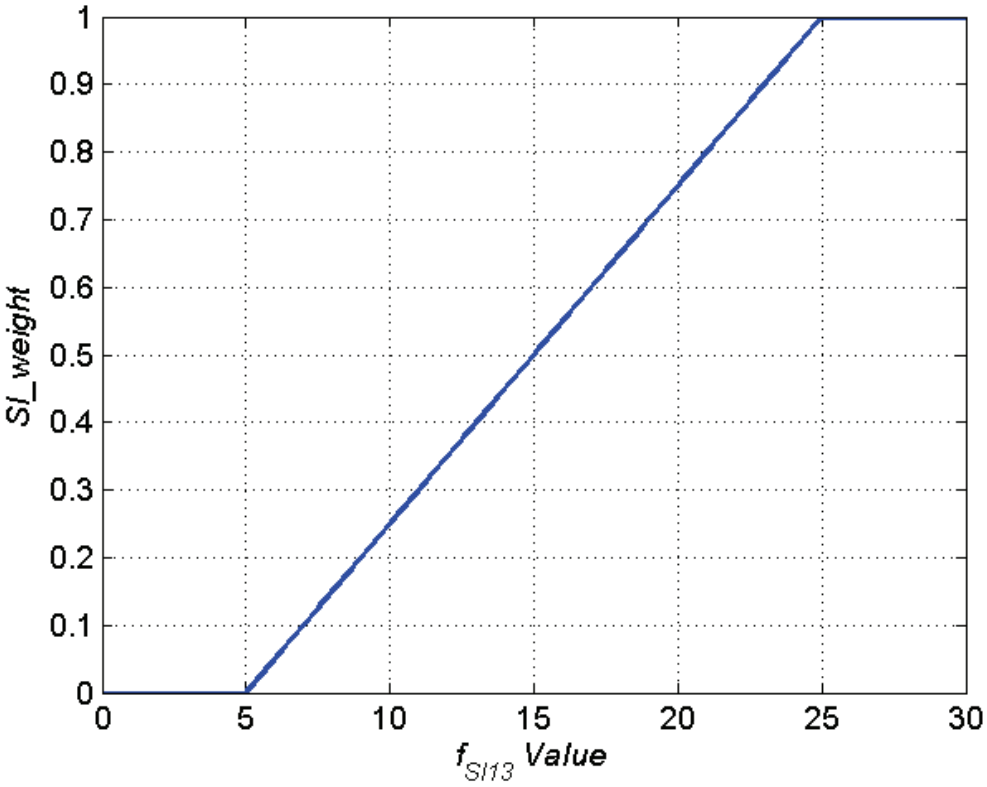

Figure 4. Weighting function *SI_weight* for modifying *hv_loss* S-T parameters.

The spatial distortion parameters can be crushed (i.e., excessive excursions beyond those found in the training data are limited or compressed) using functions like the VQM crushing function found in the VQM Calculation Section.

3.3.5 Temporal Parameters

Two temporal parameters are computed from the f_{ATI} feature, one that measures added random noise (*ati_noise*) and one that measures motion disturbances caused by transmission errors (*ati_error*). Following the mathematical notation in Annex D.8 of ITU-T Recommendation J.144, these parameters are given by:

$$ati\_noise = \text{Y\_rand5\%\_ati0.2s\_rms\_5\_ratio\_gain\_between25\%50\%}$$

$$ati\_error = \text{Y\_rand5\%\_ati0.2s\_rms\_max7pt\_12\_ratio\_gain\_above90\%}$$

To make the *ati_noise* and *ati_error* parameters more robust against temporal misalignments, the parameters are computed for all temporal alignments of the processed video that are within plus or minus 0.4 seconds of the best estimated temporal alignment to the original video, and then the minimum parameter value is selected.

3.4 VQM Calculation

Similar to the NTIA General VQM in Annex D of ITU-T Recommendation J.144, the Fast Low Bandwidth VQM calculation linearly combines two parameters from the f_{HV13} feature (*hv_loss* and *hv_gain*), two parameters from the f_{SI13} feature (*si_loss* and *si_gain*), and two parameters from the f_{COHER_COLOR} feature (except that the two parameters have been combined into a single color

distortion parameter called *color_comb*). The one noise parameter in the NTIA General VQM has been replaced with 2 parameters based on the low bandwidth f_{ATI} feature described here (*ati_noise* and *ati_error*).

Thus, VQM_{FLB} (abbreviation for Fast Low Bandwidth VQM) consists of a linear combination of eight parameters. VQM_{FLB} is given by:

$$VQM_{FLB} = \{\ 0.38317338378290 * hv\_loss + 0.37313218013131 * hv\_gain +$$
$$0.58033514546526 * si\_loss + 0.95845512360511 * si\_gain +$$
$$1.07581708014998 * color\_comb +$$
$$0.17693274495002 * ati\_noise + 0.02535903906351 * ati\_error\ \}$$

The total VQM (after the contributions of all the parameters are added up) is clipped at a lower threshold of 0.0 to prevent negative VQM numbers. Finally, a crushing function that allows a maximum of 50% overshoot is applied to VQM values over 1.0 to limit VQM values for excessively distorted video that falls outside the range of the training data.

If $VQM_{FLB} > 1.0$, then $VQM_{FLB} = (1 + c)*VQM_{FLB} / (c + VQM_{FLB})$, where $c = 0.5$.

VQM_{FLB} computed in the above manner will have values greater than or equal to zero and a nominal maximum value of one. VQM_{FLB} may occasionally exceed one for video scenes that are extremely distorted.

To make VQM_{FLB} more robust against spatial misalignments, VQM_{FLB} is computed for all spatial alignments of the processed video that are within plus or minus 1 pixel of the best estimated spatial alignment to the original video, and then the minimum VQM_{FLB} is selected.

4. References

[1] ITU-R Recommendation BT.601-6 (01/07), "*Studio encoding parameters of digital television for standard 4:3 and wide screen 16:9 aspect ratios*."

[2] Video Quality Model (VQM) Software Tools - Binary Executables and Source Code, available from the National Telecommunications and Information Administration (NTIA) at http://www.its.bldrdoc.gov/n3/video/VQM_software.php.

[3] ITU-T Recommendation J.244 (04/08), "*Full reference and reduced reference calibration methods for video transmission systems with constant misalignment of spatial and temporal domains with constant gain and offset*."

[4] VQEG Final Report of MM Phase I Validation Test (2008), "*Final report from the Video Quality Experts Group on the validation of objective models of multimedia quality assessment, phase I*", Video Quality Experts Group (VQEG), http://www.its.bldrdoc.gov/vqeg/projects/multimedia, ITU-T Study Group 9 TD923, Study Period 2005-2008.

[5] ITU-T Recommendation J.144 (03/04), "*Objective Perceptual Video Quality Measurement Techniques for Digital Cable Television in the Presence of a Full Reference*."

[6] ITU-R Recommendation BT.1683 (06/04), "*Objective Perceptual Video Quality Measurement Techniques for Standard Definition Digital Broadcast Television in the Presence of a Full Reference*."

5. Reference Code for Implementing the Fast Low Bandwidth VQM

The purpose of this reference code is to assist the user with proper implementation of the Fast Low Bandwidth VQM. While MATLAB® code is used for the reference code, any software code can be used that reproduces the results given here. Each subsection in section 5 contains MATLAB code for the function named in the section header (e.g., save the contents of section 5.1 to a file called "fastlowbw_ref.m"). Execute fastlowbw_ref with no arguments to receive help information on how to call the routine. This code contains the flexibility of running the model on a short video clip (i.e., 5 to 15 seconds) within a larger video sequence (e.g., 1-minute sequence). This is done by shifting the short video clip by one second and re-computing the model for each temporal shift. While this functionality is not demonstrated below, comments within the code and returned arguments from "model_fastlowbw_parameters.m" will refer to this capability. This hidden capability may be useful for implementing an in-service video quality monitoring system.

When the sample test vectors (i.e., video clips) are processed with the Fast Low Bandwidth VQM reference code (function "fastlowbw_ref.m"), text files are produced that contain the calibration and model results. For the following example MATLAB function calls, output files similar to those given below should be obtained (due to random processes used by the Fast Low Bandwidth VQM, results may vary slightly from those presented here):

fastlowbw_ref '"rrtv_flogar_original.yuv"' '"rrtv_flogar_hrc1.yuv"' '"QCIF30"' '"none"'

"rrtv_flogar_hrc1.yuv_calibration.txt" file with results similar to this:

```
none
    0  Horizontal Shift
    0  Vertical Shift
    1  Valid Region Top
    1  Valid Region Left
  144  Valid Region Bottom
  176  Valid Region Right
1.000 Luminance Gain
0.000 Luminance Offset
 1000  Horizontal Scale
 1000  Vertical Scale
    0  Temporal Delay
```

"rrtv_flogar_hrc1.yuv_model.txt" file with results similar to this:

```
0.512650 fastlowbw
0.156115 hv_loss
0.072374 hv_gain
0.174293 si_loss
0.028922 si_gain
0.080944 color_comb
0.000000 noise
0.000002 error
0.000000 vshift
0.000000 hshift
```

fastlowbw_ref '"rrtv_flogar_original.yuv"' '"rrtv_flogar_hrc1.yuv"' '"QCIF30"' '"rrcal2"'

"rrtv_flogar_hrc1.yuv_calibration.txt" file with results similar to this:

```
rrcal2
    0  Horizontal Shift
    0  Vertical Shift
    1  Valid Region Top
    1  Valid Region Left
  144  Valid Region Bottom
  176  Valid Region Right
0.998 Luminance Gain
0.539 Luminance Offset
 1000  Horizontal Scale
 1000  Vertical Scale
    1  Temporal Delay
1.005 Cb Gain
0.218 Cb Offset
0.968 Cr Gain
0.098 Cr Offset
```

"rrtv_flogar_hrc1.yuv_model.txt" file with results similar to this:

```
0.507892 fastlowbw
0.151204 hv_loss
0.071144 hv_gain
0.178214 si_loss
0.028708 si_gain
```

```
0.078499 color_comb
0.000000 noise
0.000124 error
0.000000 vshift
0.000000 hshift
```

fastlowbw_ref '"f:\sg9_rrtv\rrtv_calmob_original.yuv"' '"f:\sg9_rrtv\rrtv_calmob_hrc2.yuv"' '"QCIF30"' '"rrcal2scale"'

“rrtv_calmob_hrc2.yuv_calibration.txt” file with results similar to this:

```
rrcal2scale
    0  Horizontal Shift
    0  Vertical Shift
    1  Valid Region Top
    1  Valid Region Left
  144  Valid Region Bottom
  176  Valid Region Right
0.998 Luminance Gain
-0.185 Luminance Offset
 1000  Horizontal Scale
 1000  Vertical Scale
    0  Temporal Delay
0.857 Cb Gain
-2.043 Cb Offset
0.868 Cr Gain
-2.727 Cr Offset
```

“rrtv_calmob_hrc2.yuv_model.txt” file with results similar to this:

```
0.291195 fastlowbw
0.051529 hv_loss
0.056814 hv_gain
0.084128 si_loss
0.004541 si_gain
0.092818 color_comb
0.000000 noise
0.001365 error
0.000000 vshift
0.000000 hshift
```

Function Listing

5.1 Function "fastlowbw_ref.m"

```
function fastlowbw_ref (original_file, processed_file, file_type, calibration);
% wrapper for fastlowbw_ref.  This is not the typical MATLAB interface, but rather
% an alternate interface required for compilation.
% 1) call with no arguments for help information.
% 2) if running inside MATLAB, add three quotes (''') both before and after
% every argument.

if nargin == 0,
```

```
fprintf('FASTLOWBW_REF -- Version 1.0\n');
fprintf('  Take original and processed test sequences in raw BIG-YUV \n');
fprintf('  file format.  Calculate NTIA Fast Low Bandwidth model and calibration.\n');
fprintf('  This is reference software.\n');
fprintf('SYNTAX\n');
fprintf('  fastlowbw_ref original_file processed_file video_standard calibration model\n');
fprintf('DESCRIPTION\n');
fprintf('  Takes the name of a UYVY formatted file (''original_file'') containing an\n');
fprintf('  original video sequence and UYVY formatted file (''processed_file'') containing\n');
fprintf('  a processed video sequence.  Both files must contain uncompressed video in\n');
fprintf('  either the UYVY or RGB color space.\n');
fprintf('  ''video_standard'' indicates the frame rate and video size:\n');
fprintf('    ''525''          525-line, 30fps video (720 pixels by 486 rows), "NTSC"\n');
fprintf('                   Interlaced fields, lower field presented earlier in time\n');
fprintf('    ''625''          625-line, 25fps video (720 pixels by 576 rows), "PAL"\n');
fprintf('                   Interlaced fields, upper field presented earlier in time\n');
fprintf('    ''VGA30''        VGA (480 lines x 640 pixels), 30fps video, progressive.\n');
fprintf('    ''VGA25''        VGA (480 lines x 640 pixels), 25fps video, progressive.\n');
fprintf('    ''CIF30''        CIF (288 lines x 352 pixels), 30fps video, progressive.\n');
fprintf('    ''CIF25''        CIF (288 lines x 352 pixels), 25fps video, progressive.\n');
fprintf('    ''QCIF30''        CIF (144 lines x 176 pixels), 30fps video, progressive.\n');
fprintf('    ''QCIF25''        CIF (144 lines x 176 pixels), 25fps video, progressive.\n');
fprintf('  ''calibration'' indicates the calibration options desired, and must\n');
fprintf('  be one of the following:\n');
fprintf('    ''none''       No calibration will be performed.\n');
fprintf('    ''rrcal2''      Reduced Reference Bandwidth Calibration Version 2.0 (J.Cal)\n');
fprintf('    ''rrcal2scale'' Reduced Reference Bandwidth Calibration Version 2.0 (J.Cal),\n');
fprintf('                  including estimation of spatial scaling\n');
fprintf('EXAMPLE CALL:\n');
fprintf('  fastlowbw_ref ''original.yuv'' ''processed.yuv'' ''525'' ''rrcal2''\n');
fprintf('RESTRICTIONS:\n');
fprintf('  If the video sequences (after calibration) are longer than 15 seconds, then\n');
fprintf('  only the first 15 seconds will be used for model calculation\n');
fprintf('  Temporal registration uncertainty will +/- 1 sec.\n');
fprintf('NOTES:\n');
fprintf('  These algorithms are unchanged from those previously released to the public\n');
fprintf('  (i.e., CVQM, BVQM).  Source code and binary executables are available for\n');
fprintf('  download at www.its.bldrdoc.gov  These algorithms can be freely used for\n');
fprintf('  commercial and non-commercial applications.\n');
```

```
    return;
end

% NOTES FOR PROGRAMMERS:
%
% 1. To modify the temporal delay search range, change variable "uncert"
% passed into function dll_lowbw_temporal.m
%
% 2. To modify spatial shift and scaling search limits, change variables
% "max_shift_horiz", "max_shift_vert", "max_scale_horiz" and
% "max_scale_vert" at the beginning of function dll_lowbw_calib_processed.m
% and dll_lowbw_calib_original.m.  These values must be identical for both
% functions, and are currently set automatically based on image size.
% Choosing larger values may make the algorithm unstable.
%
% 3. The original file is referred to by an ID of "1" throughout, and the
% processed file is referred to by an ID of "2".  These are referred to
% within functions by the variable name "fn" (i.e., "F" for "file" and "N" for
% "number").
%
% 4. To separate into upstream and downstream parts, simply call each
% function requiring the original file ("1") into one piece, and the
% functions requiring the processed file ("2") into another piece.  This
% code presumes downstream monitoring or bi-directional communication.
% Some of the functions require both processed video file and the original
% features and results from the previous steps.
%
% 5. If implementing a system that is entirely down-stream (i.e., no
% communication from processed to original), then the original will need to
% assume a valid region.  Discarding the overscan is recommended (see also
% dll_default_vr.m).
%
% 6. The reduced reference calibration routines and model presented herein
% use some randomized processies.  As a result, the results of this program
% will differ slightly from one run to the next.  This variability can be
% prevented by initializeing the random number generator with a constant
% value.  This will aid debugging.
%
```

```
% 7. When comparing results from this implementation to results from
% BVQM, slight differences will be obverved.  These stem from the random
% number generator, yet also because BVQM performs filtering over the
% results from multiple video sequences.  The places where filtering should
% be performed is mentioned in comments, below.



% strip off the extra single quotes '' for Windows compile, comment these
% eval lines out for Linux compile
original_file = eval(original_file);
processed_file = eval(processed_file);
file_type = eval(file_type);
calibration = eval(calibration);

% create names for output files & temporary file
temporary_file = sprintf('%s_temp.txt', processed_file);
model_file = sprintf('%s_model.txt', processed_file);
calibration_file = sprintf('%s_calibration.txt', processed_file);
error_file = sprintf('%s_errors.txt', processed_file);
cvqm_error(error_file, 0);

if strcmpi(file_type, '525'),
    video_standard = 'interlace_lower_field_first';
    rows = 486;
    cols = 720;
    fps = 30;
elseif strcmpi(file_type, '625'),
    video_standard = 'interlace_upper_field_first';
    rows = 576;
    cols = 720;
    fps = 25;
elseif strcmpi(file_type, 'VGA30'),
    video_standard = 'progressive';
    rows = 480;
    cols = 640;
    fps = 30;
elseif strcmpi(file_type, 'VGA25'),
    video_standard = 'progressive';
```

```
    rows = 480;
    cols = 640;
    fps = 25;
elseif strcmpi(file_type, 'CIF30'),
    video_standard = 'progressive';
    rows = 288;
    cols = 352;
    fps = 30;
elseif strcmpi(file_type, 'CIF25'),
    video_standard = 'progressive';
    rows = 288;
    cols = 352;
    fps = 25;
elseif strcmpi(file_type, 'QCIF30'),
    video_standard = 'progressive';
    rows = 144;
    cols = 176;
    fps = 30;
elseif strcmpi(file_type, 'QCIF25'),
    video_standard = 'progressive';
    rows = 144;
    cols = 176;
    fps = 25;
else
    error('fastlowbw_ref called with an invalid string for ''video_standard''.  Aborting');
end

if strcmpi(calibration,'rrcal2') || strcmpi(calibration,'rrcal2scale') || strcmpi(calibration,'none'),
    % fine!
else
    cvqm_error(error_file, 1, 'Calibration request string not recognized.');
    return;
end

model = 'fastlowbw';

try
    % for debugging, you'll want to comment out this try/catch loop.
    % the following print is to remind you to re-insert the try/catch back.
```

```
% fprintf('COMMENT IN TRY!\n\n');

% initialize file read
dll_video('initialize', 1, original_file, 'uyvy', video_standard, rows, cols, fps);
dll_video('initialize', 2, processed_file, 'uyvy', video_standard, rows, cols, fps);
dll_calib_video('initialize', 1);

% calculate seconds of video to be used.
[total_sec] = min( dll_video('total_sec',1), dll_video('total_sec',2) );
if total_sec > 15,
    total_sec = 15;
    cvqm_error(error_file, 3, 'File is longer than 15 seconds.');
    cvqm_error(error_file, 3, 'Results will be calculated using first 15 seconds only.');
end
if total_sec < 4,
    cvqm_error(error_file, 3, 'Video files are too brief.  Aborting.');
    return;
end

warning off;
delete(calibration_file);
warning on;

if strcmpi(calibration, 'none'),
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Use "no calibration" values
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    shift.horizontal = 0;
    shift.vertical = 0;
    pvr = dll_default_vr(1);
    y_gain = 1.0;
    y_offset = 0.0;
    scale.horizontal = 1000;
    scale.vertical = 1000;
    delay = 0;

    % We don't need to do anything with the calibration values (above),
    % because these are the defaults set by dll_calib_video.m.
```

```
    % write results.
    [sucess] = cvqm_save_calibration(calibration_file, calibration, ...
        shift, pvr, y_gain, y_offset, scale, delay);
    if sucess == 0,
        cvqm_error(error_file, 1, 'Cannot open file to write calibration results.');
        return;
    end

else

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Start Calibration
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Determine whether or not to estimate scaling.
    if strcmpi(calibration,'rrcal2'),
        no_scaling = 1;
    elseif strcmpi(calibration,'rrcal2scale'),
        no_scaling = 0;
    end

    total_sec = floor(total_sec);

    % Assume the default low bandwidth valid region as a starting point
    [vr] = dll_default_vr(1);

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Initial temporal registration
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % calcualte low bandwidth temporal registration features on the
    % original video sequence.
    [ti2_orig, ti10_orig, ymean_orig, orig_is_white_clip, orig_is_black_clip] = ...
        dll_lowbw_temporal_features(1, total_sec, vr);

    % Quantize the original temporal registration features and encode.
    % Return variables are suitable for low bandwidth transmission
    % (e.g., when saved to a *.mat file).  This save/load step is not
```

```
% demonstrated.
[ti2_index, ti10_index, y_index] = ...
    dll_lowbw_temporal_quant(1, ti2_orig, ti10_orig, ymean_orig);

% Reverse: go from encoded variables back into quantized original features.
[ti2_orig, ti10_orig, ymean_orig] = ...
    dll_lowbw_temporal_quant(0, ti2_index, ti10_index, y_index);

% calculate low bandwidth temporal registration features on the
% processed video sequence.
[ti2_proc, ti10_proc, ymean_proc, proc_is_white_clip, proc_is_black_clip] = ...
    dll_lowbw_temporal_features(2, total_sec, vr);

% Print errors and warnings.  Note white & black level clipping, if any
if orig_is_white_clip,
    cvqm_error(error_file, 2, ...
        'White level clipping detected on original video sequence may cause VQM errors.');
end
if orig_is_black_clip,
    cvqm_error(error_file, 2,...
        'Black level clipping detected on original video sequence may cause VQM errors.');
end
if proc_is_white_clip,
    cvqm_error(error_file, 2, ...
        'White level clipping detected on processed video sequence may cause VQM errors.');
end
if proc_is_black_clip,
    cvqm_error(error_file, 2, ...
        'Black level clipping detected on processed video sequence may cause VQM errors.');
end


% Compute temporal registration, using original and processed features.
% variable "uncert" is the uncertainty in seconds that should be
% searched by this temporal registration algorithmm.
uncert = 1;
[delay, sucess, is_still] = ...
    dll_lowbw_temporal (1, ti2_orig, ti2_proc, ti10_orig, ti10_proc, ymean_orig, ...
    ymean_proc, uncert, 'field');
```

```
if sucess == 0 && is_still,
    cvqm_error(error_file, 2, ...
        'Still or nearly still sequence; initial temporal registration cannot be computed.');
elseif sucess == 0,
    cvqm_error(error_file, 2, 'Initial temporal registration algorithm failed.');
end

% note if re-framing is indicated
if mod(delay,1) == 0.5,
    dll_calib_video('set_reframe',1);
    delay = delay - 0.5;
end

% re-calculate seconds of video to be used (may have changed)
[total_sec] = min( dll_calib_video('total_sec',1), dll_calib_video('total_sec',2) );
if total_sec > 15,
    total_sec = 15;
end

% apply the temporal registration to all future video read calls.
dll_lowbw_temporal_original(1, delay);
dll_lowbw_temporal_processed(2, delay);

[total_sec] = min( dll_calib_video('total_sec',1), dll_calib_video('total_sec',2) );
if total_sec > 15,
    total_sec = 15;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Spatial Shift and [optional] spatial scaling
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[seed_state] = dll_lowbw_calib_initialize;

% compute original features.
[orig_pixels, orig_horiz_profile, orig_vert_profile] = ...
    dll_lowbw_calib_original(1, seed_state, total_sec);
```

```
% Quantize the original spatial shift/scaling features and encode.
% Return variables are suitable for low bandwidth transmission
% (e.g., when saved to a *.mat file).  This save/load step is not
% demonstrated.
[index_horiz_profile, index_vert_profile] = ...
    dll_lowbw_calib_quant(1, orig_horiz_profile, orig_vert_profile);

% Reverse: go from encoded variables back into quantized original features.
[orig_horiz_profile, orig_vert_profile] = ...
    dll_lowbw_calib_quant(0, index_horiz_profile, index_vert_profile);

% Compute processed features
[shift, scale,status] = ...
    dll_lowbw_calib_processed(2, seed_state, total_sec, orig_pixels, ...
    orig_horiz_profile, orig_vert_profile, no_scaling);

% print errors and warnings.
if status.scale && ~no_scaling,
    cvqm_error(error_file, 2, 'Actual spatial scaling may be beyond search limits.');
end
if status.shift,
    cvqm_error(error_file, 2, 'Actual spatial shift may be beyond search limits.');
end
if (strcmp(video_standard,'interlace_lower_field_first') || ...
        strcmp(video_standard,'interlace_upper_field_first')) && mod(shift.vertical,2),
    cvqm_error(error_file, 2, 'Processed video will be reframed.');
    cvqm_error(error_file, 2, 'Actual temporal delay is 0.5 frames greater than reported value.');
end
if abs(shift.horizontal) > 8 || abs(shift.vertical) > 5,
    cvqm_error(error_file, 2, 'Extreme spatial shift detected.');
end
if scale.vertical ~= 1000 || scale.horizontal ~= 1000,
    cvqm_error(error_file, 2, 'Video scaling detected; please examine other scenes.');
end


% Apply above estimates, to all future video read calls.
%
% If 2+ video sequences are available for the same system, ideally the
```

```
% spatial shift and spatial scaling values would be filtered across the
% results from all video sequences (e.g., sort horizontal shifts and use
% the median horizontal shift for all video sequences).  This increases
% the shift and scaling estimation accuracy, and is particularly
% important for scaling factors.
%
% Since this reference code uses one original/processed video
% sequence pair, this cannot be demonstrated.
y_gain = 1.0;
y_offset = 0.0;
dll_calib_video('calibration', shift.horizontal, shift.vertical, vr, ...
    y_gain, y_offset, scale.horizontal, scale.vertical);

% re-calculate seconds of video to be used (may have changed)
[total_sec] = min( dll_calib_video('total_sec',1), dll_calib_video('total_sec',2) );
if total_sec > 15,
    total_sec = 15;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Valid Video Estimation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% calculate original video valid region.  This returns 4 integers, so
% quantization is unnecessary.
[ovr] = dll_orig_valid_region;

% calculate processed video valid region.
[pvr] = dll_proc_valid_region(ovr);

% print errors and warnings.
if (pvr.bottom - pvr.top + 1) / rows < 0.55 || ...
        (pvr.right - pvr.left + 1) / cols < 0.80,
    cvqm_error(error_file, 2, 'Greatly reduced valid region detected.');
end

% Apply processed valid region estimate to all future video read calls.
%
% If 2+ systems will be compared, ideally the same valid region should
```

```
% be used for that scene for all systems (PVSs).  Choose the smallest
% valid region, and apply to all PVSs.
%
% Since this reference code uses one original/processed video
% sequence pair, this cannot be demonstrated.

dll_calib_video('calibration', shift.horizontal, shift.vertical, pvr, ...
    y_gain, y_offset, scale.horizontal, scale.vertical);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Luminance, Cb and Cr gain and level offset estimation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Calculate low bandwidth gain/offset features on the original video.
[orig_y, orig_cb, orig_cr, yesno] = dll_lowbw_gain_v2_original(1, total_sec);

% Quantize the original gain/offset features and encode.
% Return variables are suitable for low bandwidth transmission
% (e.g., when saved to a *.mat file).  This save/load step is not
% demonstrated.
[index_y, index_cb, index_cr] = dll_lowbw_gain_v2_quant(1, orig_y, orig_cb, orig_cr); % quantize

% Reverse: go from encoded variables back into quantized original features.
[orig_y, orig_cb, orig_cr] = dll_lowbw_gain_v2_quant(0, index_y, index_cb, index_cr); % reconstruct

% calculate Y, Cb, and Cr gain and level offset from the original video
% features and the processed video sequence file.
[y_gain, y_offset, cb_gain, cb_offset, cr_gain, cr_offset, sucess] = ...
    dll_lowbw_gain_v2_processed(2, total_sec, orig_y, orig_cb, orig_cr, yesno);
cvqm_error(error_file, 2, 'Color Gain & Offset estimated but not removed.');

% print errors and warnings.
if y_gain < 0.9 || y_gain > 1.1,
    cvqm_error(error_file, 2, 'Extreme Luminance Gain detected.');
end
if y_offset < -20 || y_offset > 20,
    cvqm_error(error_file, 2, 'Extreme Luminance Offset detected.');
end
if sucess == 0,
```

```
    cvqm_error(error_file, 2, ...
        'Warning: algorithm used to estimate Luminance Gain & Offset may have failed.');
end
if sucess == -1,
    cvqm_error(error_file, 2, ...
        'Luminance gain & offset algorithm detected extreme values or failed; results discarded.');
end
if isnan(cb_gain),
    cvqm_error(error_file, 2, 'Warning: algorithm used to estimate Cb Gain & Offset failed.');
end
if isnan(cr_gain),
    cvqm_error(error_file, 2, 'Warning: algorithm used to estimate Cr Gain & Offset failed.');
end

% Apply the luminance gain & offset estimates to all future video read
% calls.  Do NOT apply the Cb and Cr estimates -- these are considered
% errors that the viewer may object to.
%
% If 2+ video sequences are available for the same system, ideally the
% luma gain and offset values would be filtered across the results from
% all video sequences (e.g., sort values and use the median gain for
% all video sequences).  This increases the luma gain/offset accuracy.
%
% Since this reference code uses one original/processed video
% sequence pair, this cannot be demonstrated.
dll_calib_video('calibration', shift.horizontal, shift.vertical, pvr, ...
    y_gain, y_offset, scale.horizontal, scale.vertical);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Second temporal registration -- improved estimate
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% calcualte low bandwidth temporal registration features on the
% original video sequence.
[ti2_orig, ti10_orig, ymean_orig, orig_is_white_clip, orig_is_black_clip] = ...
    dll_lowbw_temporal_features(1, total_sec, pvr);

% Quantize the original temporal registration features and encode.
```

```
% Return variables are suitable for low bandwidth transmission
% (e.g., when saved to a *.mat file).  This save/load step is not
% demonstrated.
[ti2_index, ti10_index, y_index] = ...
    dll_lowbw_temporal_quant(1, ti2_orig, ti10_orig, ymean_orig);

% Reverse: go from encoded variables back into quantized original features.
[ti2_orig, ti10_orig, ymean_orig] = ...
    dll_lowbw_temporal_quant(0, ti2_index, ti10_index, y_index);

% calcualte low bandwidth temporal registration features on the
% processed video sequence.
[ti2_proc, ti10_proc, ymean_proc, proc_is_white_clip, orig_is_black_clip] = ...
    dll_lowbw_temporal_features(2, total_sec, pvr);

% calculate temporal registration from the original and processed
% features.
uncert = 1;
[delay2, sucess, is_still] = ...
    dll_lowbw_temporal (1, ti2_orig, ti2_proc, ti10_orig, ti10_proc, ymean_orig, ymean_proc, uncert);

% print errors and warnings.
if sucess == 0 && is_still,
    cvqm_error(error_file, 2, ...
        'Still or nearly still sequence; temporal registration cannot be computed.');
elseif sucess == 0,
    cvqm_error(error_file, 2, 'Final temporal registration algorithm failed.');
end

% apply the improved delay estimate to all future video read calls.
dll_lowbw_temporal_original(1, delay2);
dll_lowbw_temporal_processed(2, delay2);

% print errors and warnings.
if abs(delay+delay2) >= round(fps),
    cvqm_error(error_file, 2, 'Temporal mis-registration exceeds 1 second uncertainty limit.');
end

% write results.
```

```
    [sucess] = cvqm_save_calibration(calibration_file, calibration, shift, pvr, y_gain, ...
        y_offset, scale, delay+delay2, cb_gain, cb_offset, cr_gain, cr_offset);
    if sucess == 0,
        cvqm_error(error_file, 1, 'Cannot open file to write lowbw calibration results.');
        return;
    end

end


% re-calculate seconds of video to be used (may have changed)
[total_sec] = min( dll_calib_video('total_sec',1), dll_calib_video('total_sec',2) );
if total_sec > 15,
    total_sec = 15;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% calculate FastLowBW model
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
warning off;
delete(model_file);
warning on;

% calculate features on the original video sequences, and save to file
% "temporary_file"
dll_features('Fast', 1, total_sec, temporary_file);

% calculate features on the processed video sequence, and hold in
% variable "proc_features"
proc_features = dll_features('Fast', 2, total_sec);

% Compute video quality metric, and return in variable "VQM"
[vqm, pars, par_names] = dll_model('vqm',temporary_file, proc_features);

% delete the temporary file with original features.
delete(temporary_file);

% Save model results to file.
[sucess] = cvqm_save_model(model_file, model, vqm, pars, par_names);
```

```
catch
    cvqm_error(error_file, 1, lasterr);
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [sucess] = cvqm_save_calibration(file_name, calibration,...
    shift, pvr, y_gain, y_offset,scale, delay, cb_gain, cb_offset, cr_gain, cr_offset);
% [sucess] = cvqm_save_calibration(file_name, calibration, shift, pvr, y_gain, y_offset,scale, delay);
%   % Open for writing 'file_name' & record type of calibration requested.
%   % 'sucess' is 1 if this function can write; 0 if fail to write.
%   % Record to file values produced by calibration


% open file & remember pointer
fid = fopen(file_name,'w');
if fid <= 0,
    sucess = 0;
    return;
else
    sucess = 1;
end

% write out type of calibration performed.
fprintf(fid, '%s\r\n', calibration);

% write out calibration values
fprintf(fid,'%5d  Horizontal Shift\r\n', shift.horizontal);
fprintf(fid,'%5d  Vertical Shift\r\n', shift.vertical);
fprintf(fid,'%5d  Valid Region Top\r\n', pvr.top);
fprintf(fid,'%5d  Valid Region Left\r\n', pvr.left);
fprintf(fid,'%5d  Valid Region Bottom\r\n', pvr.bottom);
fprintf(fid,'%5d  Valid Region Right\r\n', pvr.right);
fprintf(fid,'%5.3f Luminance Gain\r\n', y_gain);
fprintf(fid,'%5.3f Luminance Offset\r\n', y_offset);
fprintf(fid,'%5d  Horizontal Scale\r\n', scale.horizontal);
```

```
fprintf(fid,'%5d  Vertical Scale\r\n', scale.vertical);
fprintf(fid,'%5d  Temporal Delay\r\n', delay);

if exist('cr_offset'),
    fprintf(fid,'\r\n%5.3f Cb Gain\r\n', cb_gain);
    fprintf(fid,'%5.3f Cb Offset\r\n', cb_offset);
    fprintf(fid,'%5.3f Cr Gain\r\n', cr_gain);
    fprintf(fid,'%5.3f Cr Offset\r\n', cr_offset);
end

fclose(fid);



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [sucess, shift, pvr, y_gain, y_offset,scale, delay] = cvqm_load_calibration(file_name);
% [sucess, shift, pvr, y_gain, y_offset,scale, delay] = cvqm_save_calibration(file_name);
%   % Open 'file_name' & read calibration values.
%   % 'sucess' is 2 if this function can read sucessfully; 0 if failed.
%   % 'sucess' is 1 if values look to be unreasonably extreme.


% open file & remember pointer
fid = fopen(file_name,'r');
if fid <= 0,
    sucess = 0;
    return;
end
sucess = 2;

% write out type of calibration performed.
fgets(fid);

% write out calibration values
shift.horizontal = fscanf(fid,'%d');
fgets(fid);
shift.vertical = fscanf(fid,'%d');
fgets(fid);
pvr.top = fscanf(fid,'%d');
```

```
fgets(fid);
pvr.left = fscanf(fid,'%d');
fgets(fid);
pvr.bottom = fscanf(fid,'%d');
fgets(fid);
pvr.right = fscanf(fid,'%d');
fgets(fid);
y_gain = fscanf(fid,'%f');
fgets(fid);
y_offset = fscanf(fid,'%f');
fgets(fid);
scale.horizontal = fscanf(fid,'%d');
fgets(fid);
scale.vertical = fscanf(fid,'%d');
fgets(fid);
delay = fscanf(fid,'%d');
fgets(fid);

fclose(fid);

% error check
if shift.horizontal < -50 || shift.horizontal > 50 || shift.vertical < -50 || shift.vertical > 50,
    % shift really unreasonable.
    status = 1;
end
[rows,cols,fps,durration] = dll_video('size', 1);
if pvr.top < 1 || pvr.left < 1 || pvr.bottom > rows || pvr.right > cols,
    % pvr really unreasonable
    status = 1;
end
if y_gain < 0.2 || y_gain > 3.0 || y_offset < -100 || y_offset > 100,
    % gain and/or offset really unreasonable
    status = 1;
end
if scale.horizontal < 500 || scale.horizontal > 2000 || scale.vertical < 500 || scale.vertical > 2000,
    % scale really unreasonable.
    status = 1;
end
delay_sec = delay / fps;
```

```
if abs(delay_sec) > durration/2,
    % delay is longer than half the file durration!  really unreasonable.
    status = 1;
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function cvqm_error(file_name, code, message);
% cvqm_error(file_name, code, message);
%   % Append an error message to file 'file_name'.
%   % each line should start with a number (in "code") and then contain
%   % message describing the problem or issue.
%   % if code == 0, delete the previous file (if any) and ignore message.
%   % code=1    Data Input/Output invalid.  Operation impossible.
%   % code=2    Calibration values should be examined; a problem may exist.
%   % code=3    Fatal error


% open file & remember pointer
if code == 0,
    try
        warning off;
        delete(file_name);
        warning on;
    catch
    end
    return;
end

fid = fopen(file_name,'a');
fprintf(fid, '%d %s\r\n', code, message);
fclose(fid);



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [sucess] = cvqm_save_model(file_name, model,vqm, pars, par_names);
% [sucess] = cvqm_save_model(file_name, model,vqm, pars, par_names);
```

```
%    % Open for writing 'file_name' & record type of model requested.
%    % 'sucess' is 1 if this function can write; 0 if fail to write.
%    % Record to file values produced by model


% open file & remember pointer
fid = fopen(file_name,'w');
if fid <= 0,
    sucess = 0;
    return;
else
    sucess = 1;
end

% write out type of model performed.
fprintf(fid, '%f %s\r\n', vqm, model);

% write out calibration values
for cnt=1:length(pars)
    fprintf(fid, '%f %s\r\n', pars(cnt), par_names{cnt});
end

fclose(fid);
```

5.2 Function “adaptive_filter.m”

```
function [filter_size, extra] = adaptive_filter (image_size);
% ADAPTIVE_FILTER
%   Given an image size, return adaptive filter size.
%   This function adapts the filter size of the spatial filters SI and HV
%   described in SPIE 1999 paper
% SYNTAX
%   [filter_size, extra] = adaptive_filter (image_size);
% DESCRIPTION
%   Given an image size (image_size.rows, image_size.cols), return the
%   optionimal SI & HV filter length ('filter_size') and the number of extra
%   pixels needed on all sides of the image ('extra').
%
```

```
%   Filter size adjusts automatically for the image size as follows:
%       QCIF, QSIF      SI5
%       CIF, SIF        SI9
%       VGA, 601, HDTV  SI13

% find adaptive filter size
if image_size.rows <= 216,
    filter_size = 5;
    extra = 2;
elseif image_size.rows <= 384,
    filter_size = 9;
    extra = 4;
else
    filter_size = 13;
    extra = 6;
end
```

5.3 Function “adjust_requested_sroi.m”

```
function [sroi,vert,horiz] = adjust_requested_sroi (struct, varargin)
% ADJUST_REQUESTED_SROI
%  Adjust the requested Spatial Region of Interest (SROI) as specified.
% SYNTAX
%  [sroi] = adjust_requested_sroi (struct)
%  [sroi] = adjust_requested_sroi (...,'PropertyName',PropertyValue,...);
%  [sroi,vert,horiz] = adjust_requested_sroi (...);
% DESCRIPTION
%  Given clip 'struct' (in the same format as GClips or Gsscqe), return the
%  adjusted spatial region of interest ('sroi').  Return variable is a
%  structure, with four elements, 'roi.top', 'roi.left', 'roi.bottom',
%  and 'roi.right'.  NOTE: variable 'struct' uses only image_size and cvr.
%
%  'sroi',...       Requested spatial region of interest (SROI), overriding
%                   default values of SROI.  Must be followed by 4 values,
%                   specifying the region of interest, in the order: top,
%                   bottom, left, right.  SROI will be adjusted.  Default
%                   SROI given by function 'default_sroi'.
%  'hsize', value,  Horizontal size of S-T blocks.  SROI must evenly divide
```

```
%                   by this value, horizontally.
%  'vsize', value,  Vertically size of S-T blocks.  SROI must evenly divide
%                   by this value, vertizontally.
%  'extra', value,  This many valid pixels are required on all sides of the
%                   SROI, extra pixels for filtering.  Returned SROI will
%                   NOT include those extra pixels!
%  'yxextra', y, x, Number 'y' indicates the number of valid pixels required
%                   on top and bottom sides of the SROI; number 'x' indicates
%                   the number of valid pixels required on the left and right
%                   of the SROI.  Returned SROI will NOT include those extra pixels!
%
%  Optional return arguments 'vert' and 'horiz' will, if present, be filled
%  with the number of abutting blocks that fit vertically and
%  horizontally within the SROI.
% REMARKS
%  The top-left coordinate will be odd, and the bottom-right coordinate
%  even.  Thus, an equal number of pixels will be used from both fields.
%
%  Functionality tested.

% read values from struct that can be over written by variable argument
% list.
roi = default_sroi(struct.image_size);
xextra = 0;
yextra = 0;
hsize = 1;
vsize = 1;

% parse variable argument list (property values)
cnt = 1;
while cnt <= nargin - 1,
    if strcmp(lower(varargin(cnt)),'sroi') == 1,
        roi.top = varargin{cnt+1};
        roi.left = varargin{cnt+2};
        roi.bottom = varargin{cnt+3};
        roi.right = varargin{cnt+4};
        whole_image = 0;
        cnt = cnt + 5;
    elseif strcmp(lower(varargin(cnt)),'hsize') == 1,
```

```
        hsize = varargin{cnt+1};
        cnt = cnt + 2;
    elseif strcmp(lower(varargin(cnt)),'vsize') == 1,
        vsize = varargin{cnt+1};
        cnt = cnt + 2;
    elseif strcmp(lower(varargin(cnt)),'extra') == 1,
        xextra = varargin{cnt+1};
        yextra = varargin{cnt+1};
        cnt = cnt + 2;
    elseif strcmp(lower(varargin(cnt)),'yxextra') == 1,
        yextra = varargin{cnt+1};
        xextra = varargin{cnt+2};
        cnt = cnt + 3;
    else
        error('Property value passed into adjust_requestetd_sroi not recognized');
    end
end

% set minimum argument values.
if hsize <= 0 | (hsize > 2 & mod(hsize,2)),
    error('hsize must be one, or a postive even number');
end
if vsize <= 0 | (vsize > 2 & mod(vsize,2)),
    error('vsize must be one, or a postive even number');
end
if xextra < 0 | yextra < 0,
    error('Number of extra pixels for filtering must be zero or positive');
end

% make sure are within CVR and have the extra pixels for filtering.
if roi.top < struct.cvr.top + yextra,
    roi.top = struct.cvr.top + yextra;
end
if roi.left < struct.cvr.left + xextra,
    roi.left = struct.cvr.left + xextra;
end
if roi.bottom > struct.cvr.bottom - yextra,
    roi.bottom = struct.cvr.bottom - yextra;
end
```

```
if roi.right > struct.cvr.right - xextra,
    roi.right = struct.cvr.right - xextra;
end


% % We agreed to remove this restriction on Dec 13, 2005.
% % make sure top-left coordinates are odd, and bottom-right even
% if mod(roi.top,2) == 0,
%     roi.top = roi.top + 1;
% end
% if mod(roi.left,2) == 0,
%     roi.left = roi.left + 1;
% end
% if mod(roi.bottom,2) ~= 0,
%     roi.bottom = roi.bottom - 1;
% end
% if mod(roi.right,2) ~= 0,
%     roi.right = roi.right - 1;
% end


% make sure region evenly divides by vsize & vsize.
while mod((roi.bottom - roi.top + 1),vsize),
    if roi.top < struct.image_size.rows - roi.bottom,
        roi.top = roi.top + 1;
    else
        roi.bottom = roi.bottom - 1;
    end
end
while mod((roi.right - roi.left + 1),hsize),
    if roi.left < struct.image_size.cols - roi.right,
        roi.left = roi.left + 1;
    else
        roi.right = roi.right - 1;
    end
end

sroi = roi;
```

```
vert = (sroi.bottom-sroi.top+1)/vsize;
horiz = (sroi.right-sroi.left+1)/hsize;
```

5.4 Function “block_statistic.m”

```
function [s1,s2,s3] = block_statistic(y,vsize,hsize,varargin)
% BLOCK_STATISTIC
%  Extract feature from each spatial-temporal (S-T) region, producing one number
%  for each S-T block.  Takes a block of perceptually filtered images
%  and produces features.
% SYNTAX
%  [s1] = block_statistic(y,vsize,hsize,'Stat1');
%  [s1,s2] = block_statistic(y,vsize,hsize,'Stat1','Stat2');
%  [s1,s2,s3] = block_statistic(y,vsize,hsize,'Stat1','Stat2','Stat3');
% DEFINITION
%  [s1] = block_statistic(y,'Stat1'); divides time-slice of images 'y' into
%  abutting Spatial-Temporal (S-T) regions that contain 'vsize' pixels
%  vertically and 'hsize' pixels horizontally.  Then, statstic 'Stat1' is
%  computed over each S-T region, and the results are returned in 's1'.
%  When called with the names of two statistics, two statistics are
%  computed and returned, and so forth.  Available statistics are:
%  'mean' ==> compute the mean over each S-T region.
%  'std' ==> compute the standard deviation over each S-T region.
%  'rms' ==> compute the RMS over each S-T region.
%  'fraction' ==> compute fraction of pixels that are greater than or
%                 equal to 1.0
% REMARKS
%  Functionality tested pretty well.

want_mean = 0;
want_std = 0;
want_rms = 0;
want_fraction = 0;

for cnt = 1:nargin-3,
    if strcmp(lower(varargin(cnt)),'mean'),
        want_mean = 1;
```

```
        want_mean_at = cnt;
    elseif strcmp(lower(varargin(cnt)),'std'),
        want_std = 1;
        want_std_at = cnt;
    elseif strcmp(lower(varargin(cnt)),'rms'),
        want_rms = 1;
        want_rms_at = cnt;
    elseif strcmp(lower(varargin(cnt)),'fraction'),
        want_fraction = 1;
        want_fraction_at = cnt;
    end
end

if want_mean + want_std + want_rms + want_fraction ~= nargin - 3,
    error('block_statistic ''Stat'' not recognized or repeated.');
end

% check block size request.
[row,col,time] = size(y);
if mod(row,vsize) ~= 0,
    error('vertical size of block must evenly divide the SROI.');
end

if mod(col,hsize) ~= 0,
    error('horizontal size of block must evenly divide the SROI.');
end

if want_mean | want_std,
    temp = sum(y,3);  % sum over time
    temp = sum( reshape(temp,vsize,row*col/vsize)); % sum block vertically
    temp = reshape(temp,row/vsize,col)';
    temp = sum(reshape(temp,hsize,row*col/(vsize*hsize))); % sum block horizontally
    y_sum = reshape(temp,col/hsize,row/vsize)' ./ (hsize*vsize*time); % reshape
end

if want_std | want_rms,
    temp = sum(y.^2,3);
    temp = sum(reshape(temp,vsize,row*col/vsize));
```

```
    temp = reshape(temp,row/vsize,col)';
    temp = sum(reshape(temp,hsize,row*col/(vsize*hsize)));
    y_square = reshape(temp,col/hsize,row/vsize)' ./ (hsize*vsize*time);
end

if want_fraction,
    y(find(y >= 1.0)) = 1.0;
    y(find(y < 1.0)) = 0.0;

    temp = sum(y,3);  % sum over time
    temp = sum( reshape(temp,vsize,row*col/vsize)); % sum block vertically
    temp = reshape(temp,row/vsize,col)';
    temp = sum(reshape(temp,hsize,row*col/(vsize*hsize))); % sum block horizontally
    y_fraction = reshape(temp,col/hsize,row/vsize)' ./ (hsize*vsize*time); % reshape
end



if want_mean,
    switch want_mean_at,
        case 1,
            s1 = y_sum;
        case 2,
            s2 = y_sum;
        case 3,
            s3 = y_sum;
        otherwise
            error('Code defect')
    end
end

if want_std,
    y_std = sqrt( max(y_square - y_sum .^ 2,0));
    switch want_std_at,
        case 1,
            s1 = y_std;
        case 2,
            s2 = y_std;
        case 3,
```

```
            s3 = y_std;
        otherwise
            error('Code defect')
    end
end

if want_rms,
    y_rms = sqrt( y_square );
    switch want_rms_at,
        case 1,
            s1 = y_rms;
        case 2,
            s2 = y_rms;
        case 3,
            s3 = y_rms;
        otherwise
            error('Code defect')
    end
end


if want_fraction,
    switch want_fraction_at,
        case 1,
            s1 = y_fraction;
        case 2,
            s2 = y_fraction;
        case 3,
            s3 = y_fraction;
        otherwise
            error('Code defect')
    end
end
```

5.5 Function “block_statistic_shift.m”

```
function [data] = block_statistic_shift(y,vsize,hsize,varargin)
```

```
% BLOCK_STATISTIC_SHIFT
%  Extract feature from each spatial-temporal (S-T) region.  Produce one set
%  of features for each of 9 shifts:  +/- 1 horizontally & vertically.
%  See also function block_statistic.
% SYNTAX
%  [data] = block_statistic_shift(y,vsize,hsize,'Stat1');
%  [data] = block_statistic_shift(y,vsize,hsize,'Stat1','Stat2');
%  [data] = block_statistic_shift(y,vsize,hsize,'Stat1','Stat2','Stat3');
% DEFINITION
%  [s1] = block_statistic_shift(y,'Stat1'); divides time-slice of images 'y' into
%  abutting Spatial-Temporal (S-T) regions that contain 'vsize' pixels
%  vertically and 'hsize' pixels horizontally.  'y' must contain an extra 1
%  pixels on all sides, to be used for shifts.  Statstic 'Stat1' is
%  computed over each S-T region, and the results are returned in 's1'.
%  When called with the names of two statistics, two statistics are
%  computed and returned, and so forth.  Available statistics are:
%  'mean' ==> compute the mean over each S-T region, in data(:).mean
%  'std' ==> compute the standard deviation over each S-T region, in data(:).std.
%  'rms' ==> compute the RMS over each S-T region, in data(:).rms
%  'fraction' ==> compute fraction of pixels that are greater than or
%                 equal to 1.0, , in data(:).fraction
%  Return value 'data' is an array length 9, with one element for each
%  shift.

want_mean = 0;
want_std = 0;
want_rms = 0;
want_fraction = 0;

for cnt = 1:nargin-3,
    if strcmp(lower(varargin(cnt)),'mean'),
        want_mean = 1;
    elseif strcmp(lower(varargin(cnt)),'std'),
        want_std = 1;
    elseif strcmp(lower(varargin(cnt)),'rms'),
        want_rms = 1;
    elseif strcmp(lower(varargin(cnt)),'fraction'),
        want_fraction = 1;
    else
```

```
        error('block_statistic_shift ''Stat'' not recognized.');
    end
end


% check block size request.
[row,col,time] = size(y);
row = row - 2;
col = col - 2;
if mod(row,vsize) ~= 0,
    error('vertical size of block must evenly divide the SROI.');
end

if mod(col,hsize) ~= 0,
    error('horizontal size of block must evenly divide the SROI.');
end

if want_mean | want_std,
    tempT = sum(y,3);  % sum over time
    loop = 1;
    for cnt1=1:3,
        rng1=(cnt1+row-1);
        for cnt2=1:3,
            rng2=(cnt2+col-1);
            temp = sum( reshape(tempT(cnt1:rng1,cnt2:rng2),vsize,row*col/vsize)); % sum block vertically
            temp = reshape(temp,row/vsize,col)';
            temp = sum(reshape(temp,hsize,row*col/(vsize*hsize))); % sum block horizontally
            data(loop).mean = reshape(temp,col/hsize,row/vsize)' ./ (hsize*vsize*time); % reshape
            loop = loop + 1;
        end
    end
end

if want_std | want_rms,
    tempT = sum(y.^2,3);
    loop = 1;
    for cnt1=1:3,
        for cnt2=1:3,
```

```
            rng1=(cnt1+row-1);
            rng2=(cnt2+col-1);
            temp = sum(reshape(tempT(cnt1:rng1,cnt2:rng2),vsize,row*col/vsize));
            temp = reshape(temp,row/vsize,col)';
            temp = sum(reshape(temp,hsize,row*col/(vsize*hsize)));
            data(loop).rms = reshape(temp,col/hsize,row/vsize)' ./ (hsize*vsize*time);
            loop = loop + 1;
        end
    end
end

if want_fraction,
    y(find(y >= 1.0)) = 1.0;
    y(find(y < 1.0)) = 0.0;

    tempT = sum(y,3);  % sum over time
    loop = 1;
    for cnt1=1:3,
        for cnt2=1:3,
            rng1=(cnt1+row-1);
            rng2=(cnt2+col-1);
            temp = sum( reshape(tempT(cnt1:rng1,cnt2:rng2),vsize,row*col/vsize)); % sum block vertically
            temp = reshape(temp,row/vsize,col)';
            temp = sum(reshape(temp,hsize,row*col/(vsize*hsize))); % sum block horizontally
            data(loop).fraction = reshape(temp,col/hsize,row/vsize)' ./ (hsize*vsize*time); % reshape
            loop = loop + 1;
        end
    end
end

if want_std,
    for loop=1:9,
        data(loop).std = sqrt( max(data(loop).rms - data(loop).mean .^ 2,0));
    end
end

if want_rms | want_std,
    for loop=1:9,
        data(loop).rms = sqrt( data(loop).rms );
```

```
    end
end
```

5.6 Function “default_sroi.m”

```
function [roi] = default_sroi (image_size)
% DEFAULT_SROI
%  Return the default spatial region of interest (SROI) for a given image
%  size.
% SYNTAX
%  [roi] = default_sroi (image_size)
% DESCRIPTION
%  [roi] = default_sroi (image_size); takes an image size structure with
%  two elements, 'image_size.rows' and 'image_size.cols', and returns the
%  default SROI for that image size, 'roi'.  The returned variable is also
%  a structure, with four elements, 'roi.top', 'roi.left', 'roi.bottom',
%  and 'roi.right'.
% REMARKS
%  If an image size is requested that does not NTSC / 525-line or PAL / 625-line,
%  then the default SROI encompasses the entire image.
%

if (image_size.rows == 486 | image_size.rows == 480) & image_size.cols == 720,
    % NTSC / 525-line
    roi.top =    21;
    roi.left =   25;
    roi.bottom = 20+448;
    roi.right =  24+672;
elseif image_size.rows == 576 & image_size.cols == 720,
    % PAL / 625-line
    roi.top =    17;
    roi.left =   25;
    roi.bottom = 16+544;
    roi.right =  24+672;
elseif image_size.rows == 720 & image_size.cols == 1280,
    % initialize maximum valid region.
    roi.top = 7;
    roi.left = 17;
```

```
    roi.bottom = image_size.rows - 6;
    roi.right = image_size.cols - 16;
elseif image_size.rows == 1080 & image_size.cols == 1920,
    % initialize maximum valid region.
    roi.top = 7;
    roi.left = 17;
    roi.bottom = image_size.rows - 6;
    roi.right = image_size.cols - 16;
else
    roi.top = 1;
    roi.left = 1;
    roi.bottom = image_size.rows;
    roi.right = image_size.cols;
end
```

5.7 Function “dll_calib_video.m”

```
function [one,two,three,four] = dll_calib_video(control, fn, varargin);
% DLL_CALIB_VIDEO
%   This function implements calibrated video file read.  Understands
%   model's SROI.
% SYNTAX
%   [...] = dll_calib_video(control, ...);
%   [...] = dll_calib_video(control, fn, ...);
% DESCRIPTION
%   'fn' is either 1 for original, or 2 for processed (when required)
%   'control' is one of the following strings.  Additional parameters may
%   be required, as specified below:
%
% dll_calib_video('initialize', fn);
%           % initialize calibration.  dll_video(fn) must have been
%           % initialized on this computer, for either original or processed.
%
% [pvr] = dll_calib_video('pvr');
%           % get PVR.
%
% dll_calib_video('sroi', roi, extra);
%           % set Spatial Region of Interest required by model, 'extra' is
```

```
%           % the extra pixels needed on all sides for spatial filtering.
%
% dll_calib_video('max_roi');
%           % maximize SROI and PVR, given shifts.
%
% dll_calib_video('calibration', horiz, vert, pvr, gain, offset, horiz_stretch, vert_stretch);
%           % Set calibration values for processed video.  horiz & vert are
%           % spatial registration; PVR is Destination valid region.
%           % WARNING: if fn=1 and fn=2 calculated on two different
%           % computers, this call must be made on BOTH computers.
%
% [y,cb,cr] = dll_calib_video('sec', fn, durration);
%           % read [Y,Cb,Cr] images, 'durration' seconds, and calibrate.
%           % Return entire image, pixels outside of PVR replace with black.
%
% [y,cb,cr] = dll_calib_video('tslice', fn);
%           % read the next tslice of [Y,Cb,Cr] images, and calibrate!
%           % Return pixels within SROI, only.
%
% dll_calib_video('clear');
%           % Clear calibration values.
%
% [y] = dll_calib_video('peek', fn, durration);  Get the Y
%               images without removing them from the buffer.  So, the next
%               call with 'sec' or 'tslice' will get these same frames.
%               Perform shift & scaling & valid region but NOT gain &
%               offset!
%
% dll_calib_video('luma', gain, offset);
%           % Set luminance gain & offset values for processed video.
%           % WARNING: if fn=1 and fn=2 calculated on two different
%           % computers, this call must be made on BOTH computers.
%
% dll_calib_video('set_reframe', value);
%           % set reframe to yes (value=1) or no (values=0), ignoring
%           % spatial shift.  Next set of spatial shift will over-ride.
%
% value = dll_calib_video('get_reframe');
%           % get whether reframe.  yes (value=1) or no (values=0).
```

```
%
% value = dll_calib_video('total_sec', fn)
%           % return the total number of seconds of CALIBRATED video left
%           % in the file, after the current "read" point.


persistent CALIB;

if strcmp(control, 'initialize'),
    CALIB.do_reframe = 0;
    CALIB.do_calibration = 0;
    CALIB.horizontal = 0;
    CALIB.vertical = 0;
    CALIB.pvr = dll_default_vr(fn);
    CALIB.gain = 1.0;
    CALIB.offset = 0.0;
    CALIB.horiz_stretch = 1000;
    CALIB.vert_stretch = 1000;
    CALIB.sroi = [];

elseif strcmp(control,'set_reframe'),
    CALIB.do_reframe = fn;
    CALIB.do_calibration = 1;

elseif strcmp(control,'get_reframe'),
    one = CALIB.do_reframe;

elseif strcmp(control,'pvr'),
    if isfield(CALIB,'pvr'),
        [one] = CALIB.pvr;
    else
        error('PVR must be defined (default or actual) prior to model calculation');
    end

elseif strcmp(control,'print'),
    CALIB
    CALIB.sroi
    CALIB.pvr
```

```
elseif strcmp(control,'max_roi'),
    [rows,cols] = dll_video('size', 2);
    CALIB.sroi.top = 1 - min(0, CALIB.vertical);
    CALIB.sroi.left = 1 - min(0, CALIB.horizontal);
    CALIB.sroi.bottom = rows - max(0, CALIB.vertical);
    CALIB.sroi.right = cols - max(0, CALIB.horizontal);
    CALIB.pvr = CALIB.sroi;

elseif strcmp(control,'sroi'),
    CALIB.sroi = fn;
    CALIB.sroi.top = CALIB.sroi.top - varargin{1};
    CALIB.sroi.left = CALIB.sroi.left - varargin{1};
    CALIB.sroi.bottom = CALIB.sroi.bottom + varargin{1};
    CALIB.sroi.right = CALIB.sroi.right + varargin{1};

elseif strcmp(control,'sec'),
    [one,two,three] = gciC_tslice(fn, CALIB, varargin{1});

elseif strcmp(control, 'tslice'),
    [one,two,three] = gciC_tslice(fn, CALIB);
    one = one(CALIB.sroi.top:CALIB.sroi.bottom, CALIB.sroi.left:CALIB.sroi.right,:);
    if length(two) > 0,
        two = two(CALIB.sroi.top:CALIB.sroi.bottom, CALIB.sroi.left:CALIB.sroi.right,:);
        three = three(CALIB.sroi.top:CALIB.sroi.bottom, CALIB.sroi.left:CALIB.sroi.right,:);
    end

elseif strcmp(control, 'calibration'),
    CALIB.do_calibration = 1;
    CALIB.horizontal = fn;
    CALIB.vertical = varargin{1};
    CALIB.pvr = varargin{2};
    CALIB.gain = varargin{3};
    CALIB.offset = varargin{4};
    CALIB.horiz_stretch = varargin{5};
    CALIB.vert_stretch = varargin{6};

    % get buffer image if needed for reframing
```

```
    CALIB.do_reframe = 0;
    if mod(abs(CALIB.vertical),2),
        if (dll_video('exist',2) && ~strcmp('progressive',dll_video('get_video_standard', 2))) | ...
                (dll_video('exist',2) && ~strcmp('progressive',dll_video('get_video_standard', 1))),
            CALIB.do_reframe = 1;
        end
    end

    % set default sroi
    CALIB.sroi = CALIB.pvr;

elseif strcmp(control, 'luma'),
    CALIB.gain = fn;
    CALIB.offset = varargin{1};

elseif strcmp(control, 'clear'),
    CALIB.do_calibration = 0;

elseif strcmp(control, 'peek'),
    [one] = gciC_tslice_yonly_nogain(fn, CALIB, varargin{1});

elseif strcmp(control, 'total_sec'),
    [one] = dll_video('total_sec',fn);
    if CALIB.do_reframe && fn == 2,  % processed only.
        one = one - 1 / dll_video('fps', fn);
    end

end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [y,cb,cr] = gciC_tslice(fn, CALIB, durration);
% get requested images.  Do calibration.  Keep & handle reframing buffer.

    % Get time-slice of images & do calibration.
    if fn == 2 & CALIB.do_calibration,
```

```
        if exist('durration', 'var'),
            [uy,ucb,ucr] = dll_video('sec', fn, CALIB.do_reframe, durration);
        else
            [uy,ucb,ucr] = dll_video('tslice', fn, CALIB.do_reframe);
        end

        y = do_calibration_on_tslice(fn, uy, CALIB.horizontal, CALIB.vertical, ...
            CALIB.pvr, CALIB.gain, CALIB.offset, CALIB.do_reframe, CALIB.horiz_stretch, ...
            CALIB.vert_stretch, 0);
        cb = do_calibration_on_tslice(fn, ucb, CALIB.horizontal, CALIB.vertical, ...
            CALIB.pvr, 1.0, 0.0, CALIB.do_reframe, CALIB.horiz_stretch, CALIB.vert_stretch, 1);
        cr = do_calibration_on_tslice(fn, ucr, CALIB.horizontal, CALIB.vertical, ...
            CALIB.pvr, 1.0, 0.0, CALIB.do_reframe, CALIB.horiz_stretch, CALIB.vert_stretch, 1);

    else
        % Get time-slice of images only (no calibration)
        if exist('durration', 'var'),
            [y,cb,cr] = dll_video('sec', fn, 0, durration);
        else
            [y,cb,cr] = dll_video('tslice', fn);
        end
    end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function c_image = do_calibration_on_tslice(fn, u_image, horizontal, vertical, pvr, ...
    gain, offset, do_reframe, horiz_stretch, vert_stretch, is_color);

if strcmp('interlace_lower_field_first',dll_video('get_video_standard',fn)),
    f1 = 2;
    f2 = 1;
elseif strcmp('interlace_upper_field_first', dll_video('get_video_standard',fn)),
    f1 = 1;
    f2 = 2;
elseif strcmp('progressive', dll_video('get_video_standard',fn)),
end
```

```
[rows,cols, frames] = size(u_image);

% do reframing & shift, if required
if do_reframe,
    % reshape the images
    u_image = reshape(u_image, 2,rows/2,cols,frames);

    c_image = u_image(:,:,:,2:frames);
%    c_image = zeros(2,rows/2,cols,frames-1);
    %
    if strcmp('interlace_lower_field_first',dll_video('get_video_standard',fn)),
        c_image(f2,1:rows/2,:,:) = u_image(f1,1:rows/2,:,2:frames);
        c_image(f1,1:rows/2-1,:,:) = u_image(f2,2:rows/2,:,1:(frames-1));
        vertical = vertical - 1;
    else % strcmp('interlace_upper_field_first',dll_video('get_video_standard',fn))
        c_image(f1,2:rows/2,:,:) = u_image(f2,1:((rows/2)-1),:,1:(frames-1));
        c_image(f2,1:rows/2,:,:) = u_image(f1,1:rows/2,:,2:frames);
        vertical = vertical + 1;
    end

    u_image = reshape(c_image, rows, cols, frames-1);
end

if horiz_stretch ~= 1000 || vert_stretch ~= 1000,
    [row,col,time] = size(u_image);
    if strcmp('progressive', dll_video('get_video_standard',fn)),
        for cnt = 1:time,
            if is_color,
                u_image2(:,:,cnt) = resample_image(double(u_image(:,:,cnt)), ...
                    vert_stretch, horiz_stretch, 'Fast');
            else
                u_image2(:,:,cnt) = resample_image(double(u_image(:,:,cnt)), ...
                    vert_stretch, horiz_stretch);
            end
        end
    else
        for cnt = 1:time,
            if is_color,
                u_image2(:,:,cnt) = resample_image(double(u_image(:,:,cnt)), ...
```

```
                    vert_stretch, horiz_stretch, 'Fast', 'Interlace');
            else
                u_image2(:,:,cnt) = resample_image(double(u_image(:,:,cnt)), ...
                    vert_stretch, horiz_stretch, 'Interlace');
            end
        end
    end
    u_image = u_image2;
    clear u_image2;
end

% undo shift.
c_image = circshift(u_image,[-vertical, -horizontal, 0]);
% undo gain & offset
if gain ~= 1.0 && offset ~= 0.0,
    c_image = double(c_image);
    c_image =  (c_image - offset) / gain;
end

% zero area voided by PVR
c_image(1:pvr.top-1,:) = 0;
c_image(pvr.bottom+1:rows,:) = 0;
c_image(:,1:pvr.left-1) = 0;
c_image(:,pvr.right+1:cols) = 0;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [y] = gciC_tslice_yonly_nogain(fn, CALIB, durration);
% get requested images.  Do calibration.  Keep & handle reframing buffer.

    % Get time-slice of images.
    [y] = dll_video('peek', fn, CALIB.do_reframe, durration);

    % Do calibration if appropriate.  Skip gain/offset.
    % perform in groups of 10 images, because this can be memory intensive.
    if fn == 2 && CALIB.do_calibration,
        [r,c,t]=size(y);
```

```
        y2 = zeros(r,c,t-CALIB.do_reframe,'single');
        for i=1:10:(t-CALIB.do_reframe),
            j=min(i+9,t-CALIB.do_reframe);
            y2(:,:,i:j) = do_calibration_on_tslice(fn, y(:,:,i:(j+CALIB.do_reframe)), ...
                CALIB.horizontal, CALIB.vertical, ...
                CALIB.pvr, 1.0, 0.0, CALIB.do_reframe, CALIB.horiz_stretch, CALIB.vert_stretch, 0);
        end
        y = y2;
        clear y2;
    end
```

5.8 Function “dll_default_vr.m”

```
function [roi] = dll_default_vr (fn)
% DLL_DEFAULT_VR
%  Return the default valid region for a given image size.
% SYNTAX
%  [roi] = dll_default_vr (fn)
% DESCRIPTION
%  This function takes fn, initialized in dll_video, and returns the default valid
%  region for that image size, 'roi'.  The returned variable is also
%  a structure, with four elements, 'roi.top', 'roi.left', 'roi.bottom',
%  and 'roi.right'.

[image_size.rows,image_size.cols] = dll_video('size',fn);

if (image_size.rows == 486 | image_size.rows == 480) & image_size.cols == 720,
    % NTSC / 525-line
    roi.top =    19;
    roi.left =   23;
    roi.bottom = image_size.rows - 18;
    roi.right =  image_size.cols - 22;
elseif image_size.rows == 576 & image_size.cols == 720,
    % PAL / 625-line
    roi.top =    15;
    roi.left =   23;
    roi.bottom = image_size.rows - 14;
```

```
    roi.right =  image_size.cols - 22;
elseif image_size.rows == 720 & image_size.cols == 1280,
    % initialize maximum valid region.
    roi.top = 7;
    roi.left = 17;
    roi.bottom = image_size.rows - 6;
    roi.right = image_size.cols - 16;
elseif image_size.rows == 1080 & image_size.cols == 1920,
    % initialize maximum valid region.
    roi.top = 7;
    roi.left = 17;
    roi.bottom = image_size.rows - 6;
    roi.right = image_size.cols - 16;
else
    roi.top = 1;
    roi.left = 1;
    roi.bottom = image_size.rows;
    roi.right = image_size.cols;
end
```

5.9 Function “dll_features.m”

```
function [features] = dll_features (model_name, fn, durration, compressed_file);
% DLL_FEATURES
%   Calculate features for a model.
% SYNTAX
%   [features] = dll_features(model_name, fn, durration);
%   [features] = dll_features(model_name, fn, durration, compressed_features_file);
% DESCRIPTION
%   Calculate original or processed features needed to calculate one model.
%   Function 'dll_video' must be initialized for (fn).  'model_name' is the
%   name of the model to be run:
%       'Low'        Low-Bandwidth Model
%       'General'    NTIA General Model
%       'Developers' Developer's Model
%   'fn' is 1 for original and 2 for processed.
%   'durration' is the durration of the video sequence for which the
%   features are to be calculated, in seconds (from 5 to 30 seconds)
```

```
%   'compressed_features_file' is the name of a file where the compressed
%   features should be written.  Currently, this option is only available
%   for fn=1 and model_name='Low' (low-bandwidth model, original features).
%
%   Return variable 'features' is a structure holding the uncompressed
%   feature data.
%

warning off MATLAB:max:mixedSingleDoubleInputs

% initialize model.
[model_tslice_sec, model_planes] = dll_model('initialize', model_name, durration, fn);
dll_video('set_tslice', fn, model_tslice_sec);

% run tslices through features
if strcmp(model_planes, 'y'),
    ready_for_vqm = 0;
    while ~ready_for_vqm,
        [y] = dll_calib_video('tslice', fn);
        [ready_for_vqm] = dll_model('tslice', y);
    end
elseif strcmp(model_planes, 'ycbcr'),
    [fps] = dll_video('fps', fn);
    ready_for_vqm = 0;
    while ~ready_for_vqm,
        [y, cb, cr] = dll_calib_video('tslice', fn);
        [ready_for_vqm] = dll_model('tslice', y, cb, cr, fps);
    end
end

% retrieve the features.
[features] = dll_model('get');

if (strcmp(model_name,'Low') | strcmp(model_name,'Fast')) & fn==1 & exist('compressed_file','var'),
    model_lowbw_compression ('compress', compressed_file, features.si_std, features.hv_ratio, ...
        features.y_mean, features.cb_mean, features.cr_mean, features.ati_rms );
end
```

5.10 Function “dll_lowbw_calib_initialize.m”

```
function [seed_state] = dll_lowbw_calib_initialize;
% DLL_LOWBW_CALIB_INITIALIZE
%   Initialize low bandwidth calibration.
% SYNTAX
%   [seed_state] = dll_lowbw_calib_initialize;
% DESCRIPTIONS
%   Initialize low bandwidth calibration.  The value returned by this
%   function ('seed_state') must be passed IDENTICALLY to
%   dll_lowbw_calib_original and then dll_lowbw_calib_processed.  The next
%   time these two functions are again required (i.e., the next video
%   sequence), this initialization function should be called again.

rand('seed',sum(100*clock));
seed_state = round(rand * 255);
seed_state = uint8(seed_state);
```

5.11 Function “dll_lowbw_calib_original.m”

```
function [orig_pixels, orig_horiz_profile, orig_vert_profile] = ...
    dll_lowbw_calib_original(fn, seed_state, num_sec)
% DLL_LOWBW_CALIB_ORIGINAL
%   Calcualte original features needed for low bandwidth calibration.
% SYNTAX
%  [orig_pixels, orig_horiz_profile, orig_vert_profile] = ...
%       fast_calibration(fn, seed_state, num_sec)
% DESCRIPTION
%  Calcualte original features needed for low bandwidth calibration:
%  estimate spatial registration and scaling registration for each processed
%  clip.  Video clip must be temporally registered first.
%  'fn' is the file identifier from dll_video, which should always be fn=1.
%  'seed_state' is as returned by dll_lowbw_initialize, which should be
%  called anew each time this function is called.
%  'num_sec' is the number of seconds of video from file fn=1 that should
%  be used for calibration.
%
%  Return values 'orig_pixels', 'orig_horiz_profile', and
```

```
%  'orig_vert_profile' are required by function dll_lowbw_calib_processed.


num_sec = floor(num_sec);

% set up constants.
max_scale = 100; % maximum scaling search, 10%

[rows,cols, fps] = dll_video('size', fn);
durration = floor( dll_video('total_frames',1) / fps );
if durration < num_sec,
    num_sec = durration;
end

if rows <= 216,
    max_shift_horiz = 4;  % maximum search in any direction, in # pixels
    max_shift_vert = 4;
    max_scale_horiz = 60; % maximum scaling search,
    max_scale_vert = 40;
elseif rows <= 384,
    max_shift_horiz = 8;
    max_shift_vert = 8;
    max_scale_horiz = 60;
    max_scale_vert = 40;
else
    max_shift_horiz = 20;
    max_shift_vert = 20;
    max_scale_horiz = 100;
    max_scale_vert = 60;
end

% error checks & corrections
if mod(max_shift_horiz,2),
    max_shift_horiz = max_shift_horiz + 1;
end
if mod(max_shift_vert,2),
    max_shift_vert = max_shift_vert + 1;
end
```

```
% compute PVR and OROI given the above.
max_pixels_horiz = max_shift_horiz + (max_scale_horiz / 1000) * cols;
max_pixels_horiz = ceil(max_pixels_horiz);
max_pixels_horiz = max_pixels_horiz + mod(max_pixels_horiz,2);

max_pixels_vert = max_shift_vert + (max_scale_vert / 1000) * rows;
max_pixels_vert = ceil(max_pixels_vert);
max_pixels_vert = max_pixels_vert + mod(max_pixels_vert,2);

[pvr, oroi] = find_pvr_oroi_guess(rows, cols, max_pixels_horiz, max_pixels_vert);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Algorithm:
%
%   Use one frame every second (approx).
%       horizontal and vertical profiles AND
%       80% as many randomly subsampled pixels as there are profile pixels
%           - randomly distributed over all frames
%   Search over ALL frames simultaneously.
%   Search original +- 0 second (yes! ZERO);
%   shift processed +- 20 pixels/lines for NTSC (as specified else)
%   scale processed by +/- 10% for NTSC (as specified else)
%       rescale using nearest neighbor
%
%   When have final scale & shift, compute luminance gain & offset with
%   those pixels & profiles, too.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



% Choose the % of pixels to be used.
rows = oroi.bottom - oroi.top + 1;
cols = oroi.right - oroi.left + 1;
[list_row, list_col, list_time, list_o] = ...
```

```
    sas_choose_pixels (seed_state, rows, cols, num_sec, max_pixels_horiz, max_pixels_vert);

% load frames
dll_video('set_rewind', fn);
for loop = 1:num_sec,
    y(:,:,loop) = dll_video('sec', fn, 0, 1.0/fps); % don't reframe
    if loop ~= num_sec,
        dll_video('discard', fn, (fps-1)/fps);
    end
end
dll_video('rewind', fn);

y = double(y);

% cut out OROI
orig = y(oroi.top:oroi.bottom, oroi.left:oroi.right, :);

% Compute original profiles.
[orig_horiz_profile, orig_vert_profile] = sas_profile_images(orig);

% reshape rows & columns into one dimension.
[rows,cols,time] = size(orig);
orig_y = reshape(orig, rows*cols*time,1);
clear orig;

%list of coordinates for profiles
list_horiz_profile = (1:cols) + max_pixels_horiz;
list_vert_profile = (1:rows) + max_pixels_vert;

% pick our original pixels
orig_pixels = orig_y(list_o);


% % % Limit precision on return variables
% % orig_pixels = char(orig_pixels);
% % orig_horiz_profile = uint16( round(orig_horiz_profile * 257));
% % orig_vert_profile = uint16( round(orig_vert_profile * 257));
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [pvr] = find_pvr_guess(rows, cols);
% return the best guess for pvr based only on image size.
% i.e., discard overscan.
% 'fchoice' is 1 for 'field' or 0 for 'frame' depending on which is desired.

if (rows == 486 | rows == 480 ) ...
        & cols == 720,
    pvr.top = 19;
    pvr.bottom = 486 - 18;
    pvr.left = 23;
    pvr.right = 720 - 22;
elseif rows == 576 & cols == 720,
    pvr.top = 15;
    pvr.bottom = 576 - 14;
    pvr.left = 23;
    pvr.right = 720 - 22;
else
    pvr.top = 1;
    pvr.bottom = rows;
    pvr.left = 1;
    pvr.right = cols;
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [pvr, oroi] = find_pvr_oroi_guess(rows, cols, max_pixels_horiz, max_pixels_vert);
% Find best guess at PVR and OROI, given image size and
% size of maximum search ('max_pixels_horiz'), in pixels.

pvr = find_pvr_guess(rows, cols);
oroi.top = pvr.top + max_pixels_vert;
oroi.bottom = pvr.bottom - max_pixels_vert;
oroi.left = pvr.left + max_pixels_horiz;
```

```
oroi.right = pvr.right - max_pixels_horiz;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [list_row, list_col, list_time, list_o] = ...
    sas_choose_pixels (seed_state, rows, cols, time, max_pixels_horiz, max_pixels_vert);

% Choose 80% as many random points as profile points.
need = ceil(0.80 * (rows+cols)*time );

rand('state', double(seed_state));
list_row = round(rand(1,need) * (rows) + 0.5);
list_col = round(rand(1,need) * (cols) + 0.5);
list_time = round(rand(1,need) * (time) + 0.5);

% limit to range available.  'rand' is unlikely to exceed that range, but
% it is possible.
list_row = max( min(list_row,rows), 1);
list_col = max( min(list_col,cols), 1);
list_time = max( min(list_time,time), 1);

list_o = (list_time-1)*rows*cols + (list_col-1)*rows + list_row;

% change coordinates to be for processed, where there are more rows &
% columns.
list_row = list_row + max_pixels_vert;
list_col = list_col + max_pixels_horiz;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [horiz_profile, vert_profile] = sas_profile_images(y);
% Compute the horizontal profile for each frame (averaging each column).
% Put all such together into one giant image.
% Return that image in profile_image.
```

```
[rows,cols,time] = size(y);

horiz_profile = zeros(cols, time);
vert_profile = zeros(rows, time);
for cnt = 1:time,
    horiz_profile(:,cnt) = mean(y(:,:,cnt))';
    vert_profile(:,cnt) = mean(y(:,:,cnt), 2);
end
```

5.12 Function “dll_lowbw_calib_processed.m”

```
function [shift, scale, status] = ...
    dll_lowbw_calib_processed(fn, seed_state, num_sec, orig_pixels, ...
    orig_horiz_profile, orig_vert_profile, no_scaling);
% DLL_LOWBW_CALIB_PROCESSED
%   Calculate processed features and low bandwidth calibration.
% SYNTAX
%   [shift, scale, status] = ...
%       dll_lowbw_calib_processed(fn, seed_state, num_sec, orig_pixels, ...
%       orig_horiz_profile, orig_vert_profile, no_scaling);
% DESCRIPTION
%  Calcualte processed features needed for low bandwidth calibration, then
%  estimate spatial registration and scaling registration for each processed
%  clip.  Video clip must be temporally registered first.
%  'fn' is the file identifier from dll_video, which should always be fn=2.
%  'seed_state' is as returned by dll_lowbw_initialize, which should be
%  called anew each time this function is called.
%  'num_sec' is the number of seconds of video from file fn=1 and 2 that should
%  be used for calibration.
%  'orig_pixels','orig_horiz_profile', and 'orig_vert_profile' are results
%  from dll_lowbw_calib_original, for the original video (i.e., fn=1).
%   'no_scaling' is 1 to pre sume no spatial scaling,
%   'no_scaling' is 0 to calculate spatial scaling
%
%   status.error of 1 indicates error,
%   status.scale of 1 indicates scale returned is equal to search limit,
%   status.shift of 1 indicates shift returned is equal to search limit,
```

```
num_sec = floor(num_sec);

% uncompress input arguments.
seed_state = double(seed_state);
% % orig_pixels = single(orig_pixels);
% % orig_horiz_profile = double( orig_horiz_profile ) / 257;
% % orig_vert_profile = double( orig_vert_profile ) / 257;

%
status.error = 1;
status.scale = 0;
status.shift = 0;
status.luminance = 0;
shift.horizontal = 0;
shift.vertical = 0;
scale.horizontal = 1000;
scale.vertical = 1000;



% try
    % set up constants.
    max_scale = 100; % maximum scaling search, 10%


    [rows,cols, fps] = dll_video('size', fn);
    durration = floor( dll_calib_video('total_sec',2) );
    if durration < num_sec,
        num_sec = durration;
    end

    if rows <= 216,
        max_shift_horiz = 4;  % maximum search in any direction, in # pixels
        max_shift_vert = 4;
        max_scale_horiz = 60; % maximum scaling search,
        max_scale_vert = 40;
    elseif rows <= 384,
        max_shift_horiz = 8;
```

```
    max_shift_vert = 8;
    max_scale_horiz = 60;
    max_scale_vert = 40;
else
    max_shift_horiz = 20;
    max_shift_vert = 20;
    max_scale_horiz = 100;
    max_scale_vert = 60;
end

% error checks & corrections
if mod(max_shift_horiz,2),
    max_shift_horiz = max_shift_horiz + 1;
end
if mod(max_shift_vert,2),
    max_shift_vert = max_shift_vert + 1;
end

% compute PVR and OROI given the above.
max_pixels_horiz = max_shift_horiz + (max_scale_horiz / 1000) * cols;
max_pixels_horiz = ceil(max_pixels_horiz);
max_pixels_horiz = max_pixels_horiz + mod(max_pixels_horiz,2);

max_pixels_vert = max_shift_vert + (max_scale_vert / 1000) * rows;
max_pixels_vert = ceil(max_pixels_vert);
max_pixels_vert = max_pixels_vert + mod(max_pixels_vert,2);

[pvr, oroi] = find_pvr_oroi_guess(rows, cols, max_pixels_horiz, max_pixels_vert);

% run the calibration algorithm
[shift, scale, status] = sas_core_algorithm( ...
    num_sec, fps, max_shift_horiz, max_shift_vert, max_scale_horiz, ...
    max_scale_vert, max_pixels_horiz, max_pixels_vert, status, pvr, oroi, ...
    seed_state, orig_pixels, orig_horiz_profile, orig_vert_profile, no_scaling, fn);

% check for results next to the search limit.
if shift.horizontal == max_shift_horiz | shift.vertical == max_shift_vert,
    status.shift = status.shift + 1;
```

```
    end
    if abs(scale.horizontal - 1000) == max_scale_horiz | abs(scale.vertical - 1000) == max_scale_horiz,
        status.scale = status.scale + 1;
    end

    status.error = 0;

% catch
%     status.error = 1;
% end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Algorithm:
%
%   Use one frame every second (approx).
%       horizontal and vertical profiles AND
%       80% as many randomly subsampled pixels as there are profile pixels
%           - randomly distributed over all frames
%   Search over ALL frames simultaneously.
%   Search original +- 0 second (yes! ZERO);
%   shift processed +- 20 pixels/lines for NTSC (as specified else)
%   scale processed by +/- 10% for NTSC (as specified else)
%       rescale using nearest neighbor
%
%   When have final scale & shift, compute luminance gain & offset with
%   those pixels & profiles, too.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [new_spatial, new_scale, status] = sas_core_algorithm( ...
    num_sec, fps, max_shift_horiz, max_shift_vert, max_scale_horiz, max_scale_vert, ...
    max_pixels_horiz, max_pixels_vert, status, pvr, oroi, ...
    seed_state, orig_pixels, orig_horiz_profile, orig_vert_profile, no_scaling, fn);
% Compute spatial registration for one clip of fields..
```

```
new_spatial.horizontal = 0;
new_spatial.vertical = 0;
new_scale.horizontal = 1000;
new_scale.vertical = 1000;


% Choose the % of pixels to be used.
rows = oroi.bottom - oroi.top + 1;
cols = oroi.right - oroi.left + 1;
[list_row, list_col, list_time, list_o] = ...
    sas_choose_pixels (seed_state, rows, cols, num_sec, max_pixels_horiz, max_pixels_vert);


% load frames
dll_video('set_rewind', fn);
for loop = 1:num_sec,
    y(:,:,loop) = dll_calib_video('sec', fn, 1.0/fps);
    if loop ~= num_sec,
        dll_video('discard', fn, (fps-1)/fps);
    end
end
dll_video('rewind', fn);

y = double(y);

% cut out PVR
proc = y(pvr.top:pvr.bottom, pvr.left:pvr.right, :);

% Compute processed profiles.
[proc_horiz_profile, proc_vert_profile] = sas_profile_images(proc);

% reshape rows & columns into one dimension.
[rowp,colp,timep] = size(proc);
proc_y = reshape(proc, rowp*colp*timep,1);
clear proc;

%list of coordinates for profiles
list_horiz_profile = (1:cols) + max_pixels_horiz;
```

```
list_vert_profile = (1:rows) + max_pixels_vert;


if no_scaling,
    max_scale_vert = 0;
    max_scale_horiz = 0;
end

length_of_list = length(list_o);

% random search.
loop = 1;
best_scale_horiz = 1000;
best_scale_vert = 1000;
best_shift_horiz = 0;
best_shift_vert = 0;
best_value = inf;
best_loop = -1;

values = zeros(2*max_scale_horiz+1,2*max_scale_vert+1,2*max_shift_horiz+1,2*max_shift_vert+1);
values(:,:,:,:) = NaN;

random_tries = 15000; % hope it needn't be that large!
loop = 0;
cnt_used = 0;
while loop <= random_tries,

    % randomly choose stretch, shift, and time.

    % for the first 10% of tries, do a flat random search over all possibilities.
    if loop < random_tries / 10,
        scale_horiz = round( -max_scale_horiz + (2 * max_scale_horiz + 1) * rand );
        scale_vert =  round( -max_scale_vert + (2 * max_scale_vert + 1) * rand );
        shift_horiz = round( -max_shift_horiz + (2 * max_shift_horiz + 1) * rand );
        shift_vert =  round( -max_shift_vert + (2 * max_shift_vert + 1) * rand );
    % weight more near best stretch/shift/time found so far.
    else
        scale_horiz = best_scale_horiz + round( 2 * randn );
```

```
    scale_vert = best_scale_vert + round( 2 * randn );
    shift_horiz = best_shift_horiz + round( 2 * randn );
    shift_vert = best_shift_vert + round( 2 * randn );
end

if max_scale_horiz == 0,
    scale_horiz = 0;
end
if max_scale_vert == 0,
    scale_vert = 0;
end

% If this point is out of the legal range, choose again.
if abs(scale_horiz) > max_scale_horiz | abs(scale_vert) > max_scale_vert | ...
        abs(shift_horiz) > max_shift_horiz | abs(shift_vert) > max_shift_vert,
    continue;
end
% check whether this stretch/shift/time has been computed already.
if ~isnan( values(scale_horiz + max_scale_horiz + 1, scale_vert + max_scale_vert + 1, ...
    shift_horiz + max_shift_horiz + 1, shift_vert + max_shift_vert + 1) ),
    loop = loop + 1;
    continue;
end
cnt_used = cnt_used + 1;

% convert from chosen original coordinates (in smaller image) into
% processed coordinates.
% First, scale.  The "-0.4" is because the resampling routine we are
% training for his this factor.
curr_list_row = list_row * 1000 / (scale_vert+1000) + (rowp/2 - (1000/(scale_vert+1000)) * rowp/2);
curr_list_col = list_col * 1000 / (scale_horiz+1000) + (colp/2 - (1000/(scale_horiz+1000)) * colp/2);

% Second, shift.  Round to nearest pixel (use floor of +0.5 for speed)
curr_list_row = floor(curr_list_row + shift_vert + 0.5);
curr_list_col = floor(curr_list_col + shift_horiz + 0.5);
list_p = (list_time-1)*rowp*colp + (curr_list_col-1)*rowp + curr_list_row;

% compute the difference value of the random pixels.
```

```
pixel_list = orig_pixels - proc_y(list_p);

% scale the profiles.
curr_list_row = list_vert_profile * 1000 / ...
    (scale_vert+1000) + (rowp/2 - (1000/(scale_vert+1000)) * rowp/2);
curr_list_col = list_horiz_profile * 1000 / ...
    (scale_horiz+1000) + (colp/2-(1000/(scale_horiz+1000)) * colp/2);

% Second, shift the profiles.  Round to nearest pixel (use floor of +0.5 for speed)
curr_list_row = floor(curr_list_row + shift_vert + 0.5);
curr_list_col = floor(curr_list_col + shift_horiz + 0.5);

% compute the difference value of the profiles.  Append that to random
% pixels' differneces.
vert_diff = orig_vert_profile - proc_vert_profile(curr_list_row,:);
horiz_diff = orig_horiz_profile - proc_horiz_profile(curr_list_col,:);
pixel_list = [ pixel_list; reshape(vert_diff,rows*num_sec,1); reshape(horiz_diff,cols*num_sec,1)];

% Compute and record standard deviation of difference.
curr_value = std(pixel_list);
values(scale_horiz + max_scale_horiz + 1, scale_vert + max_scale_vert + 1, ...
    shift_horiz + max_shift_horiz + 1, shift_vert + max_shift_vert + 1) = curr_value;

% keep track of the best one!
% if find a tie, change to it if the scaling factor is closer to no
% scaling & no shifting
want = 0;
if curr_value < best_value,
    want = 1;
elseif curr_value == best_value,
    if abs(scale_horiz) <= abs(best_scale_horiz) & abs(scale_vert) <= abs(best_scale_vert) & ...
            abs(shift_horiz) <= abs(best_shift_horiz) & abs(shift_vert) <= abs(best_shift_vert),
        want = 1;
    end
end
if want,
    best_scale_horiz = scale_horiz;
    best_scale_vert = scale_vert;
```

```
        best_shift_horiz = shift_horiz;
        best_shift_vert = shift_vert;
        best_value = curr_value;
        best_loop = loop;
    end

    loop = loop + 1;
end

scale_horiz = best_scale_horiz;
scale_vert = best_scale_vert;
shift_horiz = best_shift_horiz;
shift_vert = best_shift_vert;


% send back all results.
new_spatial.horizontal = best_shift_horiz;
new_spatial.vertical = best_shift_vert;
new_scale.horizontal = best_scale_horiz+1000;
new_scale.vertical = best_scale_vert+1000;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [pvr] = find_pvr_guess(rows, cols);
% return the best guess for pvr based only on image size.
% i.e., discard overscan.
% 'fchoice' is 1 for 'field' or 0 for 'frame' depending on which is desired.

if (rows == 486 | rows == 480 ) ...
        & cols == 720,
    pvr.top = 19;
    pvr.bottom = 486 - 18;
    pvr.left = 23;
    pvr.right = 720 - 22;
elseif rows == 576 & cols == 720,
    pvr.top = 15;
```

```
    pvr.bottom = 576 - 14;
    pvr.left = 23;
    pvr.right = 720 - 22;
else
    pvr.top = 1;
    pvr.bottom = rows;
    pvr.left = 1;
    pvr.right = cols;
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [pvr, oroi] = find_pvr_oroi_guess(rows, cols, max_pixels_horiz, max_pixels_vert);
% Find best guess at PVR and OROI, given image size and
% size of maximum search ('max_pixels_horiz'), in pixels.

pvr = find_pvr_guess(rows, cols);
oroi.top = pvr.top + max_pixels_vert;
oroi.bottom = pvr.bottom - max_pixels_vert;
oroi.left = pvr.left + max_pixels_horiz;
oroi.right = pvr.right - max_pixels_horiz;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [list_row, list_col, list_time, list_o] = ...
    sas_choose_pixels (seed_state, rows, cols, time, max_pixels_horiz, max_pixels_vert);

% Choose 80% as many random points as profile points.
need = ceil(0.80 * (rows+cols)*time );

rand('state', seed_state);
list_row = round(rand(1,need) * (rows) + 0.5);
list_col = round(rand(1,need) * (cols) + 0.5);
list_time = round(rand(1,need) * (time) + 0.5);

% limit to range available.  'rand' is unlikely to exceed that range, but
```

```
% it is possible.
list_row = max( min(list_row,rows), 1);
list_col = max( min(list_col,cols), 1);
list_time = max( min(list_time,time), 1);

list_o = (list_time-1)*rows*cols + (list_col-1)*rows + list_row;

% change coordinates to be for processed, where there are more rows &
% columns.
list_row = list_row + max_pixels_vert;
list_col = list_col + max_pixels_horiz;



%%%%%%%%%%
function [horiz_profile, vert_profile] = sas_profile_images(y);
% Compute the horizontal profile for each frame (averaging each column).
% Put all such together into one giant image.
% Return that image in profile_image.

[rows,cols,time] = size(y);

horiz_profile = zeros(cols, time);
vert_profile = zeros(rows, time);
for cnt = 1:time,
    horiz_profile(:,cnt) = mean(y(:,:,cnt))';
    vert_profile(:,cnt) = mean(y(:,:,cnt), 2);
end
```

5.13 Function “dll_lowbw_calib_quant.m”

```
function [orig_horiz_profile_out, orig_vert_profile_out] = ...
    dll_lowbw_calib_quant(is_quantize, orig_horiz_profile_in, orig_vert_profile_in)
% DLL_LOWBW_CALIB_QUANT
%   Quantize & reconstruct original features for low bandwidth spatial
%   registration.
```

```
% SYNTAX
%   [orig_horiz_profile, orig_vert_profile] = ...
%       dll_lowbw_calib_quant(is_quantize, orig_horiz_profile, orig_vert_profile);
% DESCRIPTION
%   'is_quantize' is 1 for quantize, 0 to reconstruct.
%   Other two paramters are profile sfrom dll_lowbw_calib_original.m (when quantizing)
%   and indexes from previous call (when reconstructing).
%
% Note: original pixels do not need to be quantized.
%
% Example call to quantize:
%   [index_horiz_profile, index_vert_profile] =
%       dll_lowbw_calib_quant(1, orig_horiz_profile, orig_vert_profile);
% Example call to reconstruct:
%   [orig_horiz_profile, orig_vert_profile] =
%       dll_lowbw_calib_quant(0, index_horiz_profile, index_vert_profile);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Define the quantizers for the two spatial registration features.
%  These designs are for 10 bit linear quantizers.

%  profile feature
start = 0.0;  % first code
last = 255.0;  % 252 is the maximum observed in the training data
high_codes = 2^16;  % number of codes for 16-bit quantizer
code_profile = start:(last-start)/(high_codes-1):last;
%  Generate the partitions, halfway between codes
code_profile_size = size(code_profile,2);
part_profile = (code_profile(2:code_profile_size)+code_profile(1:code_profile_size-1))/2;


if is_quantize,
    %  Quantize original features
    [row,col] = size(orig_horiz_profile_in);
    [orig_horiz_profile_out] = quantiz_fast(reshape(orig_horiz_profile_in,1,row*col),part_profile);
    orig_horiz_profile_out = reshape(orig_horiz_profile_out,row,col);
```

```
    [row,col] = size(orig_vert_profile_in);
    [orig_vert_profile_out] = quantiz_fast(reshape(orig_vert_profile_in,1,row*col),part_profile);
    orig_vert_profile_out = reshape(orig_vert_profile_out,row,col);
else
    %  Undo the quantization
    [row,col] = size(orig_horiz_profile_in);
    orig_horiz_profile_out = code_profile(1+orig_horiz_profile_in);
    orig_horiz_profile_out = reshape(orig_horiz_profile_out,row,col);

    [row,col] = size(orig_vert_profile_in);
    orig_vert_profile_out = code_profile(1+orig_vert_profile_in);
    orig_vert_profile_out = reshape(orig_vert_profile_out,row,col);
end
```

5.14 Function “dll_lowbw_gain_v2_original.m”

```
function [orig_y_blocks, orig_cb_blocks, orig_cr_blocks, yesno] = ...
    dll_lowbw_gain_v2_original(fn, num_sec)
% DLL_LOWBW_GAIN_V2_ORIGINAL
%   Calculate original features needed for low bandwidth YCbCr gain &
%   offset (rrcal version 2).
% SYNTAX
%  [orig_y_blocks, orig_cb_blocks, orig_cr_blocks, yesno] = ...
%       dll_lowbw_gain_v2_original(fn, num_sec)
% DESCRIPTION
%  Calculate original features needed for low bandwidth  gain &
%  offset.  Video clip must be temporally registered first.  Spatial
%  registration & valid region should also be calculated.
%  'fn' is the file identifier from dll_video, preferably fn=1.
%  'num_sec' is the number of seconds of video from file fn=1 that should
%  be used for calibration.
%
%  Return values are required by function dll_lowbw_gain_processed.
%   'orig_y_blocks' averaged Y blocks selected.
%   'orig_cb_blocks' averaged Cb blocks selected
%   'orig_cr_blocks' averaged Cr blocks selected
%   'yesno' logicals (i.e., booleans) indicating for all blocks, which 1/2
%       of blocks were selected.
```

```
num_sec = floor(num_sec);

[rows,cols, fps] = dll_video('size', fn);
durration = floor( dll_video('total_frames',1) / fps );
if durration < num_sec,
    num_sec = durration;
end

if rows <= 216,
    block_size = 10;
elseif rows <= 384,
    block_size = 22;
else
    block_size = 46;
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Algorithm:
%
%   Use one frame every second (approx).
%   Search over ALL frames simultaneously.
%   Search original +- 0 second (yes! ZERO);
%   Use luminance image only, sub-sampled by block_size.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% set SROI given specified block size
[temp.image_size.rows,temp.image_size.cols] = dll_video('size',fn);
[temp.cvr] = dll_calib_video('pvr');
extra = 0;
[sroi,vert,horiz] = adjust_requested_sroi (temp, ...
    'vsize',block_size, 'hsize',block_size, 'extra',extra);
dll_calib_video('sroi', sroi, extra);
```

```
% allocate space for results
orig_y_blocks = zeros(vert, horiz, num_sec);
orig_y_std = zeros(vert, horiz, num_sec);
orig_cb_blocks = zeros(vert, horiz, num_sec);
orig_cr_blocks = zeros(vert, horiz, num_sec);

% loop through frames
dll_video('set_rewind', fn);
dll_video('set_tslice', fn, 1.0/fps);
for loop = 1:num_sec,
    % compute mean of frame
    [y, cb, cr] = dll_calib_video('tslice', fn);
    orig_y_blocks(:,:,loop) = block_statistic(y, block_size, block_size, 'mean');
    orig_y_std(:,:,loop) = block_statistic(y, block_size, block_size, 'std');
    orig_cb_blocks(:,:,loop) = block_statistic(cb, block_size, block_size, 'mean');
    orig_cr_blocks(:,:,loop) = block_statistic(cr, block_size, block_size, 'mean');

    % skip over the rest of the frames in this second of video.
    if loop ~= num_sec,
        dll_video('discard', fn, (fps-1)/fps);
    end
end
dll_video('rewind', fn);


% set SROI to PVR again
dll_calib_video('sroi', temp.cvr, 0);

% pick off 1/2 of blocks with lowest Y stdev

[r1,c1,t1] = size(orig_y_std);

orig_y_std = reshape(orig_y_std, r1*c1*t1, 1);
orig_y_blocks = reshape(orig_y_blocks, r1*c1*t1, 1);
orig_cb_blocks = reshape(orig_cb_blocks, r1*c1*t1, 1);
orig_cr_blocks = reshape(orig_cr_blocks, r1*c1*t1, 1);
```

```
[a,b]=sort(orig_y_std);
b = b(1:floor(length(b) / 2));
yesno = logical(orig_y_std <= orig_y_std(b(length(b))));

orig_y_blocks = orig_y_blocks(yesno);
orig_cb_blocks = orig_cb_blocks(yesno);
orig_cr_blocks = orig_cr_blocks(yesno);
```

5.15 Function “dll_lowbw_gain_v2_processed.m”

```
function [y_gain, y_offset, cb_gain, cb_offset, cr_gain, cr_offset, sucess] = ...
    dll_lowbw_gain_v2_processed(fn, num_sec, orig_y, orig_cb, orig_cr, yesno);
% DLL_LOWBW_GAIN_V2_PROCESSED
%   Calculate processed features and low bandwidth YCbCr gain/offset (rrcal
%   version 2).
% SYNTAX
%   [y_gain, y_offset, cb_gain, cb_offset, cr_gain, cr_offset, status] = ...
%       dll_lowbw_gain_v2_processed(fn, seed_state, num_sec, ...
%           orig_y_blocks, orig_cb_blocks, orig_cr_blocks, yesno);
% DESCRIPTION
%  Calculate processed features needed for low bandwidth gain &
%  offset.  Video clip must be temporally registered first.  Spatial
%  registration & valid region should also be calculated.
%  'fn' is the file identifier from dll_video, preferably fn=2.
%  'num_sec' is the number of seconds of video from file fn=2 that should
%  be used for calibration.
%
%  Other input values are computed by function dll_lowbw_gain_original.
%  Returned values are 'y_gain' the luminance gain, and 'y_offset', the
%  luminance offset; and likewise for Cb and Cr.
%
%  Return value of 'sucess' is 1 if algorithm succeeds, and 0 if algorithm
%  may have failed, and -1 if a catestrophic failure results in a return of
%  gain=1, offset=0.  Cb & Cr values set to 'nan' if algorithm failed,
%  otherwise not checked.
```

```
sucess = 1;


num_sec = floor(num_sec);

y_gain = 1;
y_offset = 0;


%figure out block size
num_sec = floor(num_sec);

[rows,cols, fps] = dll_video('size', fn);
durration = floor( dll_video('total_frames',2) / fps );
if durration < num_sec,
    num_sec = durration;
end

if rows <= 216,
    block_size = 10;
elseif rows <= 384,
    block_size = 22;
else
    block_size = 46;
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Algorithm:
%
%   Use one frame every second (approx).
%   Search over ALL frames simultaneously.
%   Search original +- 0 second (yes! ZERO);
%   Use luminance image only, sub-sampled by block_size.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% set SROI given specified block size
[temp.image_size.rows,temp.image_size.cols] = dll_video('size',fn);
[temp.cvr] = dll_calib_video('pvr');
extra = 0;
[sroi,vert,horiz] = adjust_requested_sroi (temp, ...
    'vsize',block_size, 'hsize',block_size, 'extra',extra);
dll_calib_video('sroi', sroi, extra);

% allocate space for results
proc_y = zeros(vert, horiz, num_sec);
proc_cb = zeros(vert, horiz, num_sec);
proc_cr = zeros(vert, horiz, num_sec);

% loop through frames
dll_video('set_rewind', fn);
dll_video('set_tslice', fn, 1.0/fps);
for loop = 1:num_sec,
    % compute mean of frame
    [y, cb, cr] = dll_calib_video('tslice', fn);
    proc_y(:,:,loop) = block_statistic(y, block_size, block_size, 'mean');
    proc_cb(:,:,loop) = block_statistic(cb, block_size, block_size, 'mean');
    proc_cr(:,:,loop) = block_statistic(cr, block_size, block_size, 'mean');

    % skip over the rest of the frames in this second of video.
    if loop ~= num_sec,
        dll_video('discard', fn, (fps-1)/fps);
    end
end
dll_video('rewind', fn);

% set SROI to PVR again
dll_calib_video('sroi', temp.cvr, 0);

[r1,c1,t1] = size(proc_y);
proc_y = reshape(proc_y, r1*c1*t1, 1);
proc_y = proc_y(yesno);
proc_cb = reshape(proc_cb, r1*c1*t1, 1);
```

```
proc_cb = proc_cb(yesno);
proc_cr = reshape(proc_cr, r1*c1*t1, 1);
proc_cr = proc_cr(yesno);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% eliminate blocks with clipping
% MUST be done second< so that above indicies
% are identical for y, cb, & cr
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
b = find(orig_y >= 2 & orig_y <= 253 & proc_y >= 2 & proc_y <= 253 );
if length(b) ~= length(orig_y),
    orig_y = orig_y(b);
    proc_y = proc_y(b);
end

b = find(orig_cb >= -126 & orig_cb <= 126 & proc_cb >= -126 & proc_cb <= 126 );
if length(b) ~= length(orig_cb),
    orig_cb = orig_cb(b);
    proc_cb = proc_cb(b);
end

b = find(orig_cr >= -126 & orig_cr <= 126 & proc_cr >= -126 & proc_cr <= 126 );
if length(b) ~= length(orig_cr),
    orig_cr = orig_cr(b);
    proc_cr = proc_cr(b);
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% compute gain & offset with final pixels
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if (max(max(max(orig_y)))-min(min(min(orig_y)))) < 10,
    y_gain = 1.0;
    y_offset = 0.0;
    sucess = -1;
```

```
elseif (max(max(max(proc_y)))-min(min(min(proc_y)))) <= 0
    y_gain = 1.0;
    y_offset = 0.0;
    sucess = -1;
else

    % compute initial gain via linear regression
    y = proc_y;
    x = [ones(length(y),1) orig_y];
    b = x\y;
    r = y - x*b;

    done = 0;
    prev_b = b;
    counter = 0;
    while ~done && counter < 10000,
        counter = counter + 1;

        epsilon = 1.0;
        cost = 1.0 ./ (abs(r) + epsilon); % cost vector, reciprocal of errors
        cost = cost ./ sqrt(sum(cost)); % normalize for unity norm
        cost = (cost.^2);

        xp = x' .* repmat(cost,1,2)';
        b = inv(xp*x)*xp*y;
        r = y - x*b;

        if abs(prev_b(2) - b(2)) < 0.0001,
            done = 1;
        else
            prev_b = b;
        end
    end

    if counter < 10000,
        y_gain = b(2);
        y_offset = b(1);
    else
```

```
        y_gain = 1.0;
        y_offset = 0.0;
        sucess = -1;
    end

end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% compute gain & offset with final pixels
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if (max(max(max(orig_cb)))-min(min(min(orig_cb)))) < 10,
    cb_gain = nan;
    cb_offset = nan;
elseif (max(max(max(proc_cb)))-min(min(min(proc_cb)))) <= 0
    cb_gain = nan;
    cb_offset = nan;
else

    % compute initial gain via linear regression
    y = proc_cb;
    x = [ones(length(y),1) orig_cb];
    b = x\y;
    r = y - x*b;

    done = 0;
    prev_b = b;
    counter = 0;
    while ~done && counter < 10000,
        counter = counter + 1;

        epsilon = 1.0;
        cost = 1.0 ./ (abs(r) + epsilon); % cost vector, reciprocal of errors
        cost = cost ./ sqrt(sum(cost)); % normalize for unity norm
        cost = (cost.^2);
```

```
        xp = x' .* repmat(cost,1,2)';
        b = inv(xp*x)*xp*y;
        r = y - x*b;

        if abs(prev_b(2) - b(2)) < 0.0001,
            done = 1;
        else
            prev_b = b;
        end
    end

    if counter < 10000,
        cb_gain = b(2);
        cb_offset = b(1);
    else
        cb_gain = nan;
        cb_offset = nan;
    end
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% compute gain & offset with final pixels
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if (max(max(max(orig_cr)))-min(min(min(orig_cr)))) < 10,
    cr_gain = nan;
    cr_offset = nan;
elseif (max(max(max(proc_cr)))-min(min(min(proc_cr)))) <= 0
    cr_gain = nan;
    cr_offset = nan;
else

    % compute initial gain via linear regression
    y = proc_cr;
    x = [ones(length(y),1) orig_cr];
    b = x\y;
```

```
    r = y - x*b;

    done = 0;
    prev_b = b;
    counter =0;
    while ~done && counter < 10000,
        counter = counter + 1;

        epsilon = 1.0;
        cost = 1.0 ./ (abs(r) + epsilon); % cost vector, reciprocal of errors
        cost = cost ./ sqrt(sum(cost)); % normalize for unity norm
        cost = (cost.^2);

        xp = x' .* repmat(cost,1,2)';
        b = inv(xp*x)*xp*y;
        r = y - x*b;

        if abs(prev_b(2) - b(2)) < 0.0001,
            done = 1;
        else
            prev_b = b;
        end
    end

    if counter < 10000,
        cr_gain = b(2);
        cr_offset = b(1);
    else
        cr_gain = nan;
        cr_offset = nan;
    end
end

% failure causes large offset or small gain.
if y_offset > 20 || y_gain < 0.70,
    sucess = 0;
end
if y_gain < 0.6 || y_gain > 1.6 || y_offset < -80 || y_offset > 80,
```

```
    y_gain = 1.0;
    y_offset = 0.0;
    sucess = -1;
end
```

5.16 Function “dll_lowbw_gain_v2_quant.m”

```
function [out_y, out_cb, out_cr] = dll_lowbw_gain_v2_quant(is_quantize, in_y, in_cb, in_cr);
% DLL_LOWBW_GAIN_V2_QUANT
%   Quantizer & reconstruct original features for lowbw gain & offset.
% SYNTAX
%   [index_y, index_cb, index_cr] = dll_lowbw_gain_v2_quant(1, y, cb, cr); % quantize
%   [y, cb, cr] = dll_lowbw_gain_v2_quant(0, index_y, index_cb, index_cr); % reconstruct
% DESCRIPTION
%   First argument is '1' to quantize, and '0' to reconstruct.
%   'y', 'cb', and 'cr are the original features from
%                       lowbw_gain_v2_original.m (i.e., uncompressed)
%   'index_y', 'index_cb', and 'index_cr' are the quantize indicies to be transmitted.

start = 0.0;  % first code
last = 255.0;  % 255 is the maximum observed in the training data
high_codes = 1024;  % number of codes for 10-bit quantizer, must make epsilon=1.0
code_lgo = start:(last-start)/(high_codes-1):last;

%  Generate the partitions, halfway between codes
code_lgo_size = size(code_lgo,2);
part_lgo = (code_lgo(2:code_lgo_size)+code_lgo(1:code_lgo_size-1))/2;

if is_quantize,

    %  Quantize orig_y feature like this
    [out_y] = quantiz_fast(in_y',part_lgo);

    %  Quantize org_cb feature like this
    [out_cb] = quantiz_fast(in_cb'+128,part_lgo);

    %  Quantize org_cr feature like this
```

```
    [out_cr] = quantiz_fast(in_cr'+128,part_lgo);

else

    %  Look-up the quantized value like this
    orig2 = code_lgo(1+in_y);
    out_y = orig2';

    %  Look-up the quantized value like this
    orig2 = code_lgo(1+in_cb);
    out_cb = orig2'-128;

    %  Look-up the quantized value like this
    orig2 = code_lgo(1+in_cr);
    out_cr = orig2'-128;
end
```

5.17 Function “dll_lowbw_temporal.m”

```
function [delay, sucess, is_still] = dll_lowbw_temporal (fn, ti2_orig, ti2_proc, ...
    ti10_orig, ti10_proc, ymean_orig, ymean_proc, uncert, varargin);
% DLL_LOWBW_TEMPORAL
%   Step 2: Compute delay
% SYNTAX
%   [delay, sucess, is_still] = dll_lowbw_temporal (ti2_orig, ti2_proc,
%       ti10_orig, ti10_proc, ymean_orig, ymean_proc, progressive, uncert);
%   [...] = dll_lowbw_temporal(...,'Flag',...);
% DESCRIPTION
%   Input arguments are 'fn' from dll_video, file ID for either original or
%   processed -- it doesn't matter which.  The next six input arguments are
%   the three features computed on the original video by function
%   dll_lowbw_temporal_features (named ti2_orig, ti10_orig & ymean_orig);
%   and the features computed on the processed video by function
%   dll_lowbw_temporal_features (named ti2_proc, ti10_proc & ymean_proc).
%   And finally, the temporal registration uncertainty, 'uncert', in seconds.
%
%   Optional Flags:
```

```
%       'field'     For interlaced systems, align to field accuracy.
%       'frame'     For interlaced systems, align to frame accuracy.  Default.
%
%   Return 'delay', the temporal registration delay in frames; 'sucess',
%   which is 1 if the algorithm succeeded & 0 if the algorithm failed; and
%   'is_still' which contains 1 if the video sequence appears to be still
%   or nearly still (thus temporal registration will always fail), and 0
%   otherwise.

frame_select = 1;

cnt = 1;
while cnt <= length(varargin),
    if strcmpi(varargin{cnt},'field'),
        frame_select = 0;
        cnt = cnt + 1;
    elseif strcmpi(varargin{cnt},'frame'),
        frame_select = 1;
        cnt = cnt + 1;
    else
        error('optional flag not recognized');
    end
end

[video_standard] = dll_video('get_video_standard', fn);
if strcmp(video_standard,'interlace_upper_field_first'), % e.g., if fps == 25,
    fld_num(1) = 1;
    fld_num(2) = 2;
    progressive = 0;
elseif strcmp(video_standard,'interlace_lower_field_first'),
    fld_num(1) = 2;
    fld_num(2) = 1;
    progressive = 0;
elseif strcmp(video_standard,'progressive'),
    fld_num(1) = 1;
    progressive = 1;
else
    warning('video standard not recognized');
    fld_num(1) = 1;
```

```
    progressive = 1;
end

uncert = ceil(uncert * dll_video('fps', fn));
if ~progressive,
    uncert = uncert * 2;
end

[sucess, is_still, delay] = trc_align_with_three_features(ti2_orig, ti2_proc, ...
    ymean_orig, ymean_proc, ti10_orig, ti10_proc, uncert, progressive, frame_select);


%*******************************************************************************
%*******************************************************************************
%*******************************************************************************
function [diff, is_valid] = trc_correlate_one_feature(feature_orig, feature_proc, ...
    threshold, uncert, is_progressive, frame_select, hold_name, file_name);

    diff = NaN;

    is_valid = 1;

    hold_length = min( length(feature_proc), length(feature_proc)) -2*uncert;
    proc = squeeze(feature_proc(uncert+1:uncert+hold_length));
    hold_std = std(proc);

    if hold_std < threshold,
        is_valid = 0;
        hold_std = 1;
    end
    proc = proc ./ hold_std;

    if is_progressive,
        for delay = 1:uncert*2+1,
            src = squeeze(feature_orig(delay:delay+hold_length-1));
            hold_std = std(src);
            if hold_std < threshold,
                is_valid = 0;
```

```
                return;
            end
            diff(delay) = std(src ./ hold_std - proc);
        end
    else
        if frame_select,
            for delay = 1:2:uncert*2+1,
                src = squeeze(feature_orig(delay:delay+hold_length-1));
                hold_std = std(src);
                if hold_std < threshold,
                    is_valid = 0;
                    hold_std = 1;;
                end
                diff( (delay+1)/2 ) = std(src ./ hold_std - proc);

            end
        else
            % align to nearest field -- may indicate different spatial
            % shift.
            for delay = 1:uncert*2+1,
                src = squeeze(feature_orig(delay:delay+hold_length-1));
                hold_std = std(src);
                if hold_std < threshold,
                    is_valid = 0;
                    hold_std = 1;
                end
                diff( delay ) = std(src ./ hold_std - proc);
            end
        end
    end



%*************************************************************************
function [is_valid, is_still, delay] = trc_align_with_three_features(ti2_orig, ti2_proc, ...
    y_orig, y_proc, ti10_orig, ti10_proc, uncert, is_progressive, frame_select);

    STILL_TI = 0.15;
```

```
    STILL_Y = 0.25;

    [diff_ti2,valid_ti2] = trc_correlate_one_feature(ti2_orig, ti2_proc, STILL_TI, ...
        uncert, is_progressive, frame_select);
    [diff_y,valid_y] = trc_correlate_one_feature(y_orig, y_proc, STILL_Y, uncert, ...
        is_progressive, frame_select);
    [diff_ti10,valid_ti10] = trc_correlate_one_feature(ti10_orig, ti10_proc, STILL_TI, ...
        uncert, is_progressive, frame_select);

    [is_valid, is_still, delay] = trc_align_combine(diff_ti2, diff_y, diff_ti10, uncert, ...
        valid_ti2, valid_y, valid_ti10, ...
        is_progressive, frame_select);



%*******************************************************************************
function [is_valid, is_still, delay] = trc_align_combine(diff_ti2, diff_y, diff_ti10, uncert, ...
    passvalid_ti2, passvalid_y, passvalid_ti10,  ...
    is_progressive, frame_select, test, scene, hrc, clip_number);


    CONST_VALID = 0.25;
    CONST_INVALID = 1.4;
    CONST_DELTA = 0.04;
    CONST_TI_RANGE = 3;
    CONST_Y_RANGE = 4;

    % judge whether each of the three features is valid
    if passvalid_ti2 & min(diff_ti2) < CONST_INVALID,
        range_high = find(diff_ti2 <= min(diff_ti2) + CONST_DELTA);
        range_high = range_high(length(range_high)) - range_high(1) + 1;
        if min(diff_ti2) < CONST_VALID | range_high <= CONST_TI_RANGE,
            valid_ti2 = 1;
        else
            valid_ti2 = 0;
        end
    else
        valid_ti2 = 0;
```

```
end

if passvalid_y & min(diff_y) < CONST_INVALID,
    range_high = find(diff_y <= min(diff_y) + CONST_DELTA);
    range_high = range_high(length(range_high)) - range_high(1) + 1;
    if min(diff_y) < CONST_VALID | range_high <= CONST_Y_RANGE,
        valid_y = 1;
    else
        valid_y = 0;
    end
else valid_y = 0;
end

if passvalid_ti10 & min(diff_ti10) < CONST_INVALID,
    range_high = find(diff_ti10 <= min(diff_ti10) + CONST_DELTA);
    range_high = range_high(length(range_high)) - range_high(1) + 1;
    if min(diff_ti10) < CONST_VALID | range_high <= CONST_TI_RANGE,
        valid_ti10 = 1;
    else
        valid_ti10 = 0;
    end
else
    valid_ti10 = 0;
end


% make composite plot of valid differences/correlations
is_valid = 1;
if valid_ti2 & valid_y & valid_ti10,
    diff = (diff_ti2+diff_y+diff_ti10)/3;
elseif valid_ti2 & valid_y,
    diff = (diff_ti2+diff_y)/2;
elseif valid_y & valid_ti10,
    diff = (diff_y+diff_ti10)/2;
elseif valid_ti2 & valid_ti10,
    diff = (diff_ti2+diff_ti10)/2;
elseif valid_ti2,
    diff = diff_ti2;
elseif valid_y,
```

```
        diff = diff_y;
    elseif valid_ti10,
        diff = diff_ti10;
    else
        is_valid = 0;
        delay = 0;
    end


    if is_valid,
        % find delay that minimizes correlation
        [mindiff,mindelay] = min(diff);
        if is_progressive,
            delay = - (mindelay - 1 - uncert);
        else
            if frame_select,
                delay = - (mindelay - 1 - uncert/2);
            else
                mindelay = (mindelay+1)/2;
                delay = - (mindelay - 1 - uncert/2);
            end
        end
    end
    %

    if ~passvalid_ti2 & ~passvalid_y & ~passvalid_ti10,
        is_still = 1;
    else
        is_still = 0;
    end
```

5.18 Function “dll_lowbw_temporal_features.m”

```
function [ti2, ti10, ymean, is_white_clip, is_black_clip] = ...
    dll_lowbw_temporal_features(fn, durration, pvr);
% DLL_LOWBW_TEMPORAL_FEATURES
%   Step 1: Compute features needed for temporal registration
```

```
% SYNTAX
%   [ti2, ti10, ymean, is_white_clip, is_black_clip] = ...
%       dll_lowbw_temporal_features(fn, durration);
%   [...] = dll_lowbw_temporal_features(fn, durration, pvr);
% DESCRIPTION
%   Input arguments are 'fn' from dll_video, to locate the video file;
%   'durration' in seconds, the durration of video to be used (less than
%   file length).  'pvr' computed by dll_proc_valid_region, if known, should
%   also be specified.  A default PVR will be used if this information is not
%   known.  WARNING:  'pvr' (if present) and 'durration' must be identical
%   for original & processed function calls.
%
%   Return values are three features: ti2, ti10 & ymean.
%   Also whether the video appears to contain white level clipping
%   ('is_white_clip') or black level clipping ('is_black_clip').

if ~exist('pvr','var'),
    [row,col] = dll_video('size', fn);

    % discard 4% if PVR not defined.
    hold = round(0.04*row);
    hold = hold + mod(hold,2); % next even number
    pvr.top = hold + 1;
    pvr.bottom = row - hold;

    hold = round(0.04*col);
    hold = hold + mod(hold,2); % next even number
    pvr.left = hold + 1;
    pvr.right = col - hold;
end

% initialize
[fps] = dll_video('fps', fn);
[video_standard] = dll_video('get_video_standard', fn);

if strcmp(video_standard,'interlace_upper_field_first'), % e.g., if fps == 25,
    fld_num(1) = 1;
    fld_num(2) = 2;
```

```
    progressive = 0;
elseif strcmp(video_standard,'interlace_lower_field_first'),
    fld_num(1) = 2;
    fld_num(2) = 1;
    progressive = 0;
elseif strcmp(video_standard,'progressive'),
    fld_num(1) = 1;
    progressive = 1;
else
    warning('video standard not recognized');
    fld_num(1) = 1;
    progressive = 1;
end

rows = pvr.bottom-pvr.top+1;
cols = pvr.right-pvr.left+1;

% loop through frames
dll_video('set_rewind', fn);
dll_video('set_tslice', fn, 1.0/fps);
dll_calib_video('sroi',pvr,0);
if ~progressive,
    buffer = zeros(rows/2*cols,6,2);
else
    buffer = zeros(rows*cols,6,2);
end

ti2_cnt = 1;
ti10_cnt = 1;
ymean_cnt = 1;
white_cnt = 1;
curr = 1;
for loop = 1:floor(fps*durration),
    % read frame
    y_frames = dll_calib_video('tslice', fn);

    if ~progressive,
        % Reshape frames into fields.
        y_fields = reshape(y_frames,2,rows/2,cols);
```

```
    y_fields = permute(y_fields,[2 1 3]);
    frows = rows / 2;
    buffer(:,curr,1) = reshape(y_fields(:,fld_num(1),:),rows/2*cols,1);
    buffer(:,curr,2) = reshape(y_fields(:,fld_num(2),:),rows/2*cols,1);
else
    % Change name; don't reshape frames into fields.
    [rows,cols,time] = size(y_frames);
    y_fields = reshape(y_frames,rows,1,cols);
    frows = rows;
    buffer(:,curr,1) = reshape(y_fields(:,fld_num(1),:),rows*cols,1);
end
clear y_frames;

% compute feature TI2
for fld = 1:length(fld_num),
    if loop >= 2,
        other = curr-1;
        if other < 1,
            other = other + 6;
        end
        ti2(ti2_cnt) = sqrt(mean( (buffer(:,curr,fld) - buffer(:,other,fld)).^2 ));
        ti2_cnt = ti2_cnt + 1;
    end
end

% compute feature TI10
for fld = 1:length(fld_num),
    if loop >= 6,
        other = curr-5;
        if other < 1,
            other = other + 6;
        end
        ti10(ti10_cnt) = sqrt(mean( (buffer(:,curr,fld) - buffer(:,other,fld)).^2 ));
        ti10_cnt = ti10_cnt + 1;
    end
end

% compute feature Ymean
for fld = 1:length(fld_num),
```

```
        ymean(ymean_cnt) = mean(buffer(:,curr,fld));
        ymean_cnt = ymean_cnt + 1;
    end

    % white & black clipping
    for fld = 1:length(fld_num),
        list_white(white_cnt) = length( find(buffer(:,curr,fld) >= 254) );
        list_black(white_cnt) = length( find(buffer(:,curr,fld) <= 1) );
        white_cnt = white_cnt + 1;
    end

    curr = curr + 1;
    if curr > 6,
        curr = 1;
    end

end
dll_video('rewind', fn);



list_white = list_white ./ (frows * cols) * 100;
is_white_clip = max(max(list_white)) >= 5.0;

list_black = list_black ./ (frows * cols) * 100;
is_black_clip = max(max(list_black)) >= 5.0;



ti2 = single(ti2);
ti10 = single(ti10);
ymean = single(ymean);
```

5.19 Function “dll_lowbw_temporal_original.m”

```
function dll_lowbw_temporal_original(fn, delay);
```

```
% DLL_LOWBW_TEMPORAL_ORIGINAL
%   step 3: Apply delay to original video
% SYNTAX
%   dll_lowbw_temporal_original(fn, delay);
% DESCRIPTION
%   'fn' from dll_video, fn=1, description of original video sequence.
%   'delay' as returned by dll_lowbw_temporal

% Adjust the image buffer according to the temporal registration just computed.
if delay < 0,
    [fps] = dll_video('fps', fn);
    dll_video('discard', fn, floor(-delay)/fps);
end
```

5.20 Function “dll_lowbw_temporal_processed.m”

```
function dll_lowbw_temporal_processed(fn, delay);
% DLL_LOWBW_TEMPORAL_PROCESSED
%   step 4: Apply delay to processed video
% SYNTAX
%   dll_lowbw_temporal_processed(fn, delay);
% DESCRIPTION
%   'fn' from dll_video, fn=2, description of processed video sequence.
%   'delay' as returned by dll_lowbw_temporal


% Adjust the image buffer according to the temporal registration just computed.
if delay > 0,
    [fps] = dll_video('fps', fn);
    dll_video('discard', fn, floor(delay)/fps);
end
```

5.21 Function “dll_lowbw_temporal_quant.m”

```
function [ti2, ti10, y] = dll_lowbw_temporal_quant(is_quantize, orig_ti2, orig_ti10, orig_y);
% DLL_LOWBW_TEMPORAL_QUANT
```

```
%   Quantize & reconstruct original features for low bandwidth temporal
%   registration.
% SYNTAX
%  [index_ti2, index_ti10, index_y] = dll_lowbw_temporal_quant(is_quantize, orig_ti2, orig_ti10, orig_y);
%  [orig_ti2, orig_ti10, orig_y] = dll_lowbw_temporal_quant(is_quantize, index_ti2, index_ti10, index_y);
% DESCRIPTION
%   'is_quantize' is 1 for quantize, 0 to reconstruct.
%   When quantizing, the other three paramters are the features returned by
%   function dll_lowbw_temporal_original (orig_ti2, orig_ti10, orig_y)
%   and the return values are the indexes to be transmitted.  When reconstructing,
%   the other three parameters are the indexes, and the return values are
%   the re-constructed features.
%
% Example call to quantize:
%   [ti2_index, ti10_index, y_index] =
%       dll_lowbw_temporal_quant(1, orig_ti2, orig_ti10, orig_y);
% Example call to reconstruct:
%   [orig_ti2, orig_ti10, orig_y] =
%       dll_lowbw_temporal_quant(0, ti2_index, ti10_index, y_index);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Define the quantizers for the three temporal registration features.
%  These designs are for 10 bit linear quantizers.

%  cont feature
start = 0.0;  % first code
last = 255.0;  % 252 is the maximum observed in the training data
high_codes = 4096;  % number of codes for 12-bit quantizer
code_cont = start:(last-start)/(high_codes-1):last;
%  Generate the partitions, halfway between codes
code_cont_size = size(code_cont,2);
part_cont = (code_cont(2:code_cont_size)+code_cont(1:code_cont_size-1))/2;

%  ti2 feature
start = 0.0;  % first code
last = 210.0;  % 209 is the maximum observed in the training data
high_codes = 4096;  % number of codes for 12-bit quantizer
code_ti2 = start:(last-start)/(high_codes-1):last;
```

```
%  Generate the partitions, halfway between codes
code_ti2_size = size(code_ti2,2);
part_ti2 = (code_ti2(2:code_ti2_size)+code_ti2(1:code_ti2_size-1))/2;

%  ti10 feature
start = 0.0;  % first code
last = 210.0;  % 209 is the maximum observed in the training data
high_codes = 4096;  % number of codes for 12-bit quantizer
code_ti10 = start:(last-start)/(high_codes-1):last;
%  Generate the partitions, halfway between codes
code_ti10_size = size(code_ti10,2);
part_ti10 = (code_ti10(2:code_ti10_size)+code_ti10(1:code_ti10_size-1))/2;


if is_quantize,
    %  Quantize original features
    [ti2] = quantiz_fast((orig_ti2),part_ti2);

    [ti10] = quantiz_fast((orig_ti10),part_ti10);

    [y] = quantiz_fast((orig_y),part_cont);
else
    %  Undo the quantization
    ti2 = code_ti2(1+orig_ti2);
    ti2 = reshape(ti2,1,length(ti2));

    ti10 = code_ti10(1+orig_ti10);
    ti10 = reshape(ti10,1,length(ti10));

    y = code_cont(1+orig_y);
    y = reshape(y,1,length(y));
end
```

5.22 Function “dll_model.m”

```
function [one, two, three] = dll_model(control, varargin)
```

```
% DLL_MODEL
%   Complete VQM model calculations.
%
% To Initialize:
%   [model_tslice_sec, model_planes] = dll_model('initialize', model_name, durration, fn);
%         'model_name' is the name of the model to be run:
%           'Fast'       Fast Low-Bandwidth Model
%         'fn' is 1 for original and 2 for processed -- either is okay -- where
%         function dll_video has been initialized on this computer for (fn).
%         'fn' presumed for following 'tslice' and 'get' calls, until next 'initialize'.
%         'durration' is the
%
% To Calculate features for next time-slice
%       where y (if 'model_planes' == 'y')
%       or y, cb, cr, & fps (if 'model_planes' == 'ycbcr'):
%   [ready_for_vqm] = dll_model('tslice', y);
%   [ready_for_vqm] = dll_model('tslice', y, cb, cr, fps);
%
% To Get features
%   [features] = dll_model('get');
%
% To Complete VQM model calculations.
%   [vqm, pars, par_names] = dll_model('vqm', source_features, proc_features);
%       'source_features' is the 'features' return value from dll_features(fn=1)
%           for general & developer's models.  For Lowbw & Fast models,
%           'source_features' is the file name containing compressed
%           features.
%       'proc_features' is the 'features' return value from dll_features(fn=2)
%       Function 'dll_features' must already have been run & retreived with
%       dll_model('get') for fn=1 (source_features) and fn=2 (processed
%       featues).


persistent data;


one = NaN;
two = NaN;
three = NaN;
```

```
if strcmp(control,'initialize'),
    if strcmp(varargin{1}, 'Fast'),
        [data,one,two] = model_run_Callback_initialize_fastlowbw(varargin{2}, varargin{3});
    else
        error('model name not recognized');
    end
    data.model = varargin{1};

elseif strcmp(control,'tslice'),
    if strcmp(data.model,'Fast'),
        [data, one] = model_run_Callback_feature_fastlowbw(data, varargin{1}, varargin{2}, ...
            varargin{3}, varargin{4});
    else
        error('model name not recognized');
    end

elseif strcmp(control,'vqm'),
    if strcmp(data.model,'Fast'),
        file_name = varargin{1};
        [orig_features.si_orig, orig_features.part_si_min, orig_features.part_si_max, ...
             orig_features.hv_feat_orig, orig_features.part_hv_min, orig_features.part_hv_max, ...
             orig_features.y_orig, ...
             orig_features.cb_orig, orig_features.cr_orig, orig_features.part_c_min, ...
             orig_features.part_c_max, orig_features.part_c, ...
             orig_features.ati_orig, orig_features.part_ati_min, orig_features.part_ati_max, ...
             orig_features.part_ati, orig_features.code_ati ] ...
             = model_lowbw_compression('uncompress', file_name);

        [fps] = dll_video('fps');

        [one, two, three] = model_run_Callback_vqm_fastlowbw(varargin{2}, orig_features, fps);
    else
        error('model name not recognized');
    end
elseif strcmp(control,'get'),
    one = data;
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%
function [data,model_tslice_sec, model_planes] = model_run_Callback_initialize_fastlowbw(durration, fn);

% figure out side & control option.

data.destination = (fn == 2);

model_tslice_sec = 1.0;
model_planes = 'ycbcr';

data.tslice_total = floor(durration / model_tslice_sec);
data.tslices = 0;
if data.destination,
    model_fastlowbw_features_shift('clear');
else
    model_fastlowbw_features('clear');
end

% set lowbw SROI
[image_size.rows,image_size.cols] = dll_video('size',fn);
[filter_size, extra] = adaptive_filter(image_size);
[pvr] = dll_calib_video('pvr');
[valid, cvr, sroi] = model_lowbw_sroi(extra, pvr.top, pvr.left, pvr.bottom, pvr.right);
dll_calib_video('sroi', sroi, extra+1);



%%%%%%%%%%%%%%%%%%%%%%%%%
function [data, ready_for_vqm] = model_run_Callback_feature_fastlowbw(data, y, cb, cr, fps);
% process next Time-slice

if data.tslices == data.tslice_total,
    data.tslices = 0;
end
```

```
% cut out SROI +/- 6 pixels
[rows,cols,time] = size(y);
image_size.rows = rows;
image_size.cols = cols;
[filter_size,extra] = adaptive_filter(image_size);
[valid, pvr, sroi] = model_lowbw_sroi(extra, 1, 1, rows, cols);
if ~valid,
    report_Callback('add', 'Valid Region too small.  Low Bandwidth Model cannot execute.');
    stop_Callback('button');
    return;
end

[image_size.rows,image_size.cols,junk] = size(y);
y = y(pvr.top:pvr.bottom, pvr.left:pvr.right,:);
cb = cb(pvr.top:pvr.bottom, pvr.left:pvr.right,:);
cr = cr(pvr.top:pvr.bottom, pvr.left:pvr.right,:);

% compute features.
[filter_size, extra] = adaptive_filter(image_size);
if data.destination,
    model_fastlowbw_features_shift ('memory',  y, cb, cr, fps, filter_size, extra);
else
    % discard one pixel on all sides, then compute features
    [row,col,time] = size(y);
    y = y(2:row-1, 2:col-1, :);
    cb = cb(2:row-1, 2:col-1, :);
    cr = cr(2:row-1, 2:col-1, :);
    model_fastlowbw_features ('memory',  y, cb, cr, fps, filter_size, extra);
end


% update number of tslices destination.
data.tslices = data.tslices + 1;

if data.tslices == data.tslice_total,
    if data.destination,
```

```
        [data.data] = model_fastlowbw_features_shift ('eof');
    else
        [data.si_std data.hv_ratio data.y_mean data.cb_mean data.cr_mean data.ati_rms] ...
            = model_fastlowbw_features ('eof');
    end
    ready_for_vqm = 1;
else
    ready_for_vqm = 0;
end



%%%%%%%%%%%%%%%%%%%%%%%%%
function [vqm_value, pars, par_names] = model_run_Callback_vqm_fastlowbw(proc, src, fps);

    [row,col,TIME_DELTA] = size(src.si_orig);
    for loop = 1:9,
        % calculate model
        do_test_print = 0;
        [data(loop).vqm, data(loop).hv_loss_par, data(loop).hv_gain_par,data(loop).si_loss_par, ...
            data(loop).si_gain_par, ...
            data(loop).color_comb_par, data(loop).noise_par, data(loop).error_par] = ...
            model_fastlowbw_parameters (...
                proc.data(loop).si_std, proc.data(loop).hv_ratio, proc.data(loop).y_mean, ...
                proc.data(loop).cb_mean, proc.data(loop).cr_mean, proc.data(loop).ati_rms, ...
                src.si_orig, src.hv_feat_orig, src.y_orig, src.cb_orig, src.cr_orig, src.ati_orig, ...
                fps, src.part_si_min, src.part_si_max, src.part_hv_min, src.part_hv_max, ...
                src.part_c_min, src.part_c_max, src.part_c, src.part_ati_min, src.part_ati_max, ...
                src.part_ati, src.code_ati, 0, TIME_DELTA);
    end

    % select smallest average VQM score shift
    for shift=1:9,
        vqm_mean(shift) = mean(data(shift).vqm);
    end
    [junk shift] = min(vqm_mean);
    row = floor((shift-1)/3)-1;
    col = mod(shift-1,3)-1;
```

```
    % keep only the last sample; above function produces an entire
    % time-history and we only want the ending number.
    num = length(data(shift).vqm);

    % select & copy data to return.
    pars = [data(shift).hv_loss_par(num), data(shift).hv_gain_par(num), ...
            data(shift).si_loss_par(num), data(shift).si_gain_par(num), ...
            data(shift).color_comb_par(num), data(shift).noise_par(num), ...
            data(shift).error_par(num), row, col];

    par_names = {'hv_loss' 'hv_gain' 'si_loss' 'si_gain' 'color_comb' 'noise' 'error' 'vshift' 'hshift'};

    vqm_value = data(shift).vqm(num);
```

5.23 Function “dll_orig_valid_region.m”

```
function [ovr] = dll_orig_valid_region;
% DLL_ORIG_VALID_REGION
%   Calculate original valid region (OVR)
% SYNTAX
%   [ovr] = dll_orig_valid_region;
% DESCRIPTION
%   Compute original valid region.  Use video from dll_video(fn=1).


% fetch control variables.
[rows,cols, fps] = dll_video('size',1);
[frames] = dll_video('total_frames',1);

% compute additional control variables
half_sec_frames = floor(round(fps) / 2);
half_sec_skip = (half_sec_frames - 1) / fps;
curr = 1;
image_size.rows = rows;
image_size.cols = cols;
```

```
[curr_valid_region, max_valid_region, standard] = valid_region_initialize(rows,cols);

% set rewind point
dll_video('set_rewind',1);

% loop through frames, improving valid region estimate.
for cnt = 1:half_sec_frames:(frames - half_sec_frames),
    y = dll_video('sec', 1, 0, 1/fps);
    dll_video('discard', 1, half_sec_skip);
    curr = curr + 1;
    [curr_valid_region] = vr_search (max_valid_region, curr_valid_region, y, standard, image_size);
end

% rewind
dll_video('rewind', 1);



% print result if debugging
% fprintf('VR = (%d,%d) (%d,%d)\n', curr_valid_region.top, curr_valid_region.left, ...
% curr_valid_region.bottom, curr_valid_region.right);

% error check.  override curr_valid_region if results were too small.
if curr_valid_region.bottom - curr_valid_region.top < ...
            (max_valid_region.bottom - max_valid_region.top)/2 || ...
        curr_valid_region.right - curr_valid_region.left < ...
            (max_valid_region.right - max_valid_region.left)/2
    curr_valid_region = max_valid_region;
end

ovr = curr_valid_region;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [curr_valid_region, max_valid_region, standard] = valid_region_initialize(rows, cols);
% initialize two variables, given the image size.
```

```
if rows == 486 & cols == 720,
    % initialize maximum valid region.
    max_valid_region.top = 7;
    max_valid_region.left = 7;
    max_valid_region.bottom = rows - 4;
    max_valid_region.right = cols - 6;
    standard = 1;
elseif rows == 480 & cols == 720,
    % initialize maximum valid region.
    max_valid_region.top = 7;
    max_valid_region.left = 7;
    max_valid_region.bottom = rows - 2;
    max_valid_region.right = cols - 6;
    standard = 1;
elseif rows == 576 & cols == 720,
    % initialize maximum valid region.
    max_valid_region.top = 7;
    max_valid_region.left = 17;
    max_valid_region.bottom = rows - 6;
    max_valid_region.right = cols - 16;
    standard = 1;
elseif rows == 720 & cols == 1280,
    % initialize maximum valid region.
    max_valid_region.top = 7;
    max_valid_region.left = 17;
    max_valid_region.bottom = rows - 6;
    max_valid_region.right = cols - 16;
    standard = 1;
elseif rows == 1080 & cols == 1920,
    % initialize maximum valid region.
    max_valid_region.top = 7;
    max_valid_region.left = 17;
    max_valid_region.bottom = rows - 6;
    max_valid_region.right = cols - 16;
    standard = 1;
else
    max_valid_region.top = 1;
    max_valid_region.left = 1;
```

```
    max_valid_region.bottom = rows;
    max_valid_region.right = cols;
    standard = 0;
end

% initialize current valid region.
curr_valid_region.top = rows/2-1;
curr_valid_region.left = cols/2-1;
curr_valid_region.bottom = rows/2+1;
curr_valid_region.right = cols/2+1;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [new_curr_valid_region] = vr_search_standard (max_valid_region, curr_valid_region, y);
% search bounderies for one image.

% search for left side.
locn = max_valid_region.left + 1;
prev = mean(y(:,locn - 1));
while locn < curr_valid_region.left,
    next = mean(y(:,locn));
    if next < 20 | next - 2 > prev,
        locn = locn + 1;
        prev = next;
    else
        break;
    end
end
curr_valid_region.left = locn;

% search for top side.
locn = max_valid_region.top + 1;
prev = mean(y(locn - 1,:));
while locn < curr_valid_region.top,
    next = mean(y(locn,:));
    if next < 20 | next - 2 > prev,
        locn = locn + 1;
```

```
        prev = next;
    else
        break;
    end
end
curr_valid_region.top = locn;

% search for right side.
locn = max_valid_region.right - 1;
prev = mean(y(:,locn + 1));
while locn > curr_valid_region.right,
    next = mean(y(:,locn));
    if next < 20 | next - 2 > prev,
        locn = locn - 1;
        prev = next;
    else
        break;
    end
end
curr_valid_region.right = locn;

% search for bottom side.
locn = max_valid_region.bottom - 1;
prev = mean(y(locn + 1,:));
while locn > curr_valid_region.bottom,
    next = mean(y(locn,:));
    if next < 20 | next - 2 > prev,
        locn = locn - 1;
        prev = next;
    else
        break;
    end
end
curr_valid_region.bottom = locn;

% return updated CVR
new_curr_valid_region = curr_valid_region;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [new_curr_vr] = vr_search_noborder (max_vr, curr_vr, y, image_size);
% search bounderies for one image.

% search bounderies for one image.
% max_vr MUST BE exactly equal to the image size.  This
% algorithm is intended for CIF, QCIF, VGA, and other video where the
% entire image is displayed.

% don't discard more than 4% of the rows or columns on any one border.
max_discard_rows = ceil(image_size.rows * 0.04);
max_discard_cols = ceil(image_size.cols * 0.04);

% search for left side.  Allow left side not to move in, even by one.
for locn = max_vr.left:max_discard_cols,
    if mean(y(:,locn)) < 20 | mean(y(:,locn)) + 20 < mean(y(:,locn+1)),
        % is invalid -- still increasing
    else
        break;
    end
end
curr_vr.left = locn;

% search for right side. Allow right side not to move in, even by one.
for locn = max_vr.right:-1:image_size.cols - max_discard_cols + 1,
    if mean(y(:,locn)) < 20 | mean(y(:,locn)) + 20 < mean(y(:,locn-1)),
        % is invalid -- still increasing
    else
        break;
    end
end
curr_vr.right = locn;

% search for top side. Allow top side not to move in, even by one.
for locn = max_vr.top:max_discard_rows,
    if mean(y(locn,:)) < 20 | mean(y(locn,:)) + 20 < mean(y(locn+1,:)),
        % is invalid -- still increasing
```

```
    else
        break;
    end
end
curr_vr.top = locn;

% search for bottom side.  Allow the bottom not to move in, even by one.
for locn = max_vr.bottom:-1:image_size.rows - max_discard_rows + 1,
    if mean(y(locn,:)) < 20 | mean(y(locn,:)) + 20 < mean(y(locn-1,:)),
        % is invalid -- still increasing
    else
        break;
    end
end
curr_vr.bottom = locn;

% return updated CVR
new_curr_vr = curr_vr;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [new_curr_vr] = vr_search (max_vr, curr_vr, y, standard, image_size);
% search bounderies for one image.

if standard,
    [new_curr_vr] = vr_search_standard (max_vr, curr_vr, y);
else
    [new_curr_vr] = vr_search_noborder (max_vr, curr_vr, y, image_size);
end
```

5.24 Function “dll_proc_valid_region.m”

```
function [pvr] = dll_proc_valid_region(ovr);
% DLL_PROC_VALID_REGION
%   Stand-alone DLL code.  Calculates PVR - processed valid region
% SYNTAX
```

```
%   [pvr] = dll_proc_valid_region(ovr);
% DESCRIPTION
%   Compute processed valid region.  Use video from dll_video(fn=1).
%   'ovr' is returned by function dll_orig_valid_region

% fetch control variables.
[rows,cols, fps] = dll_video('size', 2);
[frames] = dll_video('total_frames',2);

% compute control variables.
half_sec_frames = floor(round(fps) / 2);
half_sec_skip = (half_sec_frames - 1) / fps;
curr = 1;
image_size.rows = rows;
image_size.cols = cols;

% initialize using OVR and spatial registration results.
[curr_valid_region, junk, standard] = valid_region_initialize(rows,cols);
max_valid_region = ovr;

% set SROI for processed video read
dll_calib_video('max_roi');

% set rewind point
dll_video('set_rewind', 2);
% loop through frames, improving valid region estimate.
for cnt = 1:half_sec_frames:(frames - half_sec_frames),
    y = dll_calib_video('sec', 2, 1/fps);
    dll_video('discard', 2, half_sec_skip);
    curr = curr + 1;
    [curr_valid_region] = vr_search (max_valid_region, curr_valid_region, y, standard, image_size);
end
% rewind
dll_video('rewind', 2);

if standard,
    % go in by safety margin
    curr_valid_region.top = curr_valid_region.top + 1;
```

```
    curr_valid_region.bottom = curr_valid_region.bottom - 1;
    curr_valid_region.left = curr_valid_region.left + 5;
    curr_valid_region.right = curr_valid_region.right - 5;
end

% odd top/left coordinates, even bottom/right coordinates
curr_valid_region.top = curr_valid_region.top + (1-mod(curr_valid_region.top,2));
curr_valid_region.left = curr_valid_region.left + (1-mod(curr_valid_region.left,2));
curr_valid_region.bottom = curr_valid_region.bottom - mod(curr_valid_region.bottom,2);
curr_valid_region.right = curr_valid_region.right - mod(curr_valid_region.right,2);



% print result if debugging
%fprintf('VR = (%d,%d) (%d,%d)\n', curr_valid_region.top, curr_valid_region.left, ...
%  curr_valid_region.bottom, curr_valid_region.right);

% error check.  override curr_valid_region if results were too small.
if curr_valid_region.bottom - curr_valid_region.top < ...
        (max_valid_region.bottom - max_valid_region.top)/2 || ...
        curr_valid_region.right - curr_valid_region.left < ...
        (max_valid_region.right - max_valid_region.left)/2
    curr_valid_region = max_valid_region;
end

pvr = curr_valid_region;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [curr_valid_region, max_valid_region, standard] = valid_region_initialize(rows, cols);
% initialize two variables, given the image size.

if rows == 486 & cols == 720,
    % initialize maximum valid region.
    max_valid_region.top = 7;
    max_valid_region.left = 7;
    max_valid_region.bottom = rows - 4;
    max_valid_region.right = cols - 6;
```

```
    standard = 1;
elseif rows == 480 & cols == 720,
    % initialize maximum valid region.
    max_valid_region.top = 7;
    max_valid_region.left = 7;
    max_valid_region.bottom = rows - 2;
    max_valid_region.right = cols - 6;
    standard = 1;
elseif rows == 576 & cols == 720,
    % initialize maximum valid region.
    max_valid_region.top = 7;
    max_valid_region.left = 17;
    max_valid_region.bottom = rows - 6;
    max_valid_region.right = cols - 16;
    standard = 1;
elseif rows == 720 & cols == 1280,
    % initialize maximum valid region.
    max_valid_region.top = 7;
    max_valid_region.left = 17;
    max_valid_region.bottom = rows - 6;
    max_valid_region.right = cols - 16;
    standard = 1;
elseif rows == 1080 & cols == 1920,
    % initialize maximum valid region.
    max_valid_region.top = 7;
    max_valid_region.left = 17;
    max_valid_region.bottom = rows - 6;
    max_valid_region.right = cols - 16;
    standard = 1;
else
    max_valid_region.top = 1;
    max_valid_region.left = 1;
    max_valid_region.bottom = rows;
    max_valid_region.right = cols;
    standard = 0;
end

% initialize current valid region.
curr_valid_region.top = rows/2-1;
```

```
curr_valid_region.left = cols/2-1;
curr_valid_region.bottom = rows/2+1;
curr_valid_region.right = cols/2+1;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [new_curr_valid_region] = vr_search_standard (max_valid_region, curr_valid_region, y);
% search bounderies for one image.

% search for left side.
locn = max_valid_region.left + 1;
prev = mean(y(:,locn - 1));
while locn < curr_valid_region.left,
    next = mean(y(:,locn));
    if next < 20 | next - 2 > prev,
        locn = locn + 1;
        prev = next;
    else
        break;
    end
end
curr_valid_region.left = locn;

% search for top side.
locn = max_valid_region.top + 1;
prev = mean(y(locn - 1,:));
while locn < curr_valid_region.top,
    next = mean(y(locn,:));
    if next < 20 | next - 2 > prev,
        locn = locn + 1;
        prev = next;
    else
        break;
    end
end
curr_valid_region.top = locn;
```

```
% search for right side.
locn = max_valid_region.right - 1;
prev = mean(y(:,locn + 1));
while locn > curr_valid_region.right,
    next = mean(y(:,locn));
    if next < 20 | next - 2 > prev,
        locn = locn - 1;
        prev = next;
    else
        break;
    end
end
curr_valid_region.right = locn;

% search for bottom side.
locn = max_valid_region.bottom - 1;
prev = mean(y(locn + 1,:));
while locn > curr_valid_region.bottom,
    next = mean(y(locn,:));
    if next < 20 | next - 2 > prev,
        locn = locn - 1;
        prev = next;
    else
        break;
    end
end
curr_valid_region.bottom = locn;

% return updated CVR
new_curr_valid_region = curr_valid_region;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [new_curr_vr] = vr_search_noborder (max_vr, curr_vr, y, image_size);
% search bounderies for one image.

% search bounderies for one image.
% max_vr MUST BE exactly equal to the image size.  This
```

```
% algorihtm is intended for CIF, QCIF, VGA, and other video where the
% entire image is displayed.

% don't discard more than 4% of the rows or columns on any one border.
max_discard_rows = ceil(image_size.rows * 0.04);
max_discard_cols = ceil(image_size.cols * 0.04);

% search for left side.  Allow left side not to move in, even by one.
for locn = max_vr.left:max_discard_cols,
    if mean(y(:,locn)) < 20 | mean(y(:,locn)) + 20 < mean(y(:,locn+1)),
        % is invalid -- still increasing
    else
        break;
    end
end
curr_vr.left = locn;

% search for right side. Allow right side not to move in, even by one.
for locn = max_vr.right:-1:image_size.cols - max_discard_cols + 1,
    if mean(y(:,locn)) < 20 | mean(y(:,locn)) + 20 < mean(y(:,locn-1)),
        % is invalid -- still increasing
    else
        break;
    end
end
curr_vr.right = locn;

% search for top side. Allow top side not to move in, even by one.
for locn = max_vr.top:max_discard_rows,
    if mean(y(locn,:)) < 20 | mean(y(locn,:)) + 20 < mean(y(locn+1,:)),
        % is invalid -- still increasing
    else
        break;
    end
end
curr_vr.top = locn;

% search for bottom side.  Allow the bottom not to move in, even by one.
```

```
for locn = max_vr.bottom:-1:image_size.rows - max_discard_rows + 1,
    if mean(y(locn,:)) < 20 | mean(y(locn,:)) + 20 < mean(y(locn-1,:)),
        % is invalid -- still increasing
    else
        break;
    end
end
curr_vr.bottom = locn;

% return updated CVR
new_curr_vr = curr_vr;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [new_curr_vr] = vr_search (max_vr, curr_vr, y, standard, image_size);
% search bounderies for one image.

if standard,
    [new_curr_vr] = vr_search_standard (max_vr, curr_vr, y);
else
    [new_curr_vr] = vr_search_noborder (max_vr, curr_vr, y, image_size);
end
```

5.25 Function “dll_video.m”

```
function [one,two,three,four] = dll_video(control, fn, varargin);
% DLL_VIDEO
%   This function implements video file read.
% SYNTAX
%   [...] = dll_video(control, fn, ...);
% DESCRIPTION
%   'fn' is either 1 for original, or 2 for processed
%   'control' is one of the following strings.  Additional parameters may
%   be required, as specified below:
%
% dll_video('initialize', fn, file_name, 'uyvy', video_standard, rows, cols, fps);
```

```
%       'video_standard' is 'progressive',
%       'interlace_lower_field_first', or 'interlace_upper_field_first'
%
% dll_video('set_tslice', fn, tslice_len);
%          % Set durration retrieved by command 'tslice'
%
% [rows,cols,fps,durration] = dll_video('size', fn);
%          % return image size, frames per second, TOTAL file durration
%
% [fps] = dll_video('fps', fn);
% [fps] = dll_video('fps');
%          % return frames per second
%
% [video_standard] = dll_video('get_video_standard', fn);
%           % return video standard:  'progressive',
%           'interlace_lower_field_first' or 'interlace_upper_field_first'
%
% dll_video('set_rewind', fn);
%           % Set 'rewind'to go to current point in the file
%
% dll_video('rewind', fn);
%           % Go to point in file specified by 'set_rewind'
%
% dll_video('discard', fn, durration);
%           % Discard the next 'durration' seconds of images from the buffer.
%
% [y,cb,cr] = dll_video('tslice', fn, reframe);
%           % read [Y,Cb,Cr] images, the next tslice in durration,
%           % previously specified via 'set_tslice'
%
% [y,cb,cr] = dll_video('sec', fn, reframe, durration);
% [y] = dll_video('sec', fn, reframe durration);
%           % read [Y,Cb,Cr] images, 'durration' seconds.
%
% [y] = dll_video('peek', fn, reframe durration);  Get the Y
%               images without removing them from the buffer.  So, the next
%               call with 'sec' or 'tslice' will get these same frames.
%               'reframe' is 1 if ONE extra farme neede for reframing, 0 otherwise
%
```

```
% [total] = dll_video('total_frames',fn);
%           % return the total number of frames left in the file, after the
%           % current "read" point.
%
% [total] = dll_video('total_sec',fn);
%           % return the total number of seconds left in the file, after the
%           % current "read" point.
%
% [boolean] = dll_video('exist',fn);
%           % return 1 (true) if file defined/exists, 0 (false) otherwise.
%
% [code] = dll_set_align('delay_8s', delay);
%           % Works only for files exactly 8seconds long.
%           % With delay=0, discard first 0.8s start and last 0.2s.
%           % Delays around that allowed, from (-0.2) sec to (0.2sec - 1frame).
%           % For original, delay must be set to 0.
%           % After this option is chosen, 'size' will always return a
%           % durration of '7 seconds'.
%           % "code" will be 0 on success, 1 if file was longer than 8sec
%           % and 2 if file is shorter than 8sec -- in which case, the
%           % request cannot be accomodated!


persistent data1;
persistent data2;

% Don't need 'fn' to get 'fps' value.  Return value from whichever
% structure is defined.  Must always be equal, so doesn't matter.
if strcmp(control,'fps') && ~exist('fn'),
    if exist('data1') && isfield(data1,'fps'),
        one = data1.fps;
    elseif exist('data2') && isfield(data2,'fps'),
        one = data2.fps;
    else
        error('function dll_video: no ''fps'' to retreive');
    end
    return;
end
```

```
% otherwise, always need 'fn' value, to say if working from source or
% processed video.
if fn == 1,
    if exist('data1','var')
        data = data1;
    end
elseif fn == 2,
    if exist('data2','var')
        data = data2;
    end
else
    error('function dll_video: fn must be either 1 or 2');
end

%
if strcmp(control, 'initialize'),
    [data] = dll_video__initialize(varargin);
    data.sec7 = 0;

elseif strcmp(control,'exist'),
    if exist('data','var'),
        one = 1;
    else
        one = 0;
    end

elseif strcmp(control, 'set_tslice'),
    [data.tslice_frames, data.over_sec] = tslice_conversion(varargin{1}, data.fps);

elseif strcmp(control,'size'),
    one = data.rows;
    two = data.cols;
    three = data.fps;
    four = data.durration;
    % if demanding 7-second length, return that length instead of actual
    % length.
    if data.sec7 == 1,
```

```
        four = 7;
    end


elseif strcmp(control,'fps'),
    one = data.fps;

elseif strcmp(control,'get_video_standard'),
    one = data.video_standard;

elseif strcmp(control,'set_rewind'),
    data.rewind_curr_frame = data.curr_frame;
    data.rewind_overby = data.overby;

elseif strcmp(control,'rewind'),
    data.curr_frame = data.rewind_curr_frame;
    data.overby = data.rewind_overby;

elseif strcmp(control,'discard'),
    [tslice_frames, over_sec] = tslice_conversion(varargin{1}, data.fps);
    data.curr_frame = data.curr_frame + tslice_frames;
    data.overby = data.overby + over_sec;
    if data.overby >= 1.0,
        data.curr_frame = data.curr_frame - 1;
        data.overby = data.overby - 1.0;
    end

elseif strcmp(control, 'tslice'),
    if length(varargin) == 0,
        [one, two, three, data] = dll_video__next_tslice(data);
    elseif length(varargin) == 1,
        [one, two, three, data] = dll_video__next_tslice(data, varargin{1});
    else
        error('not sure what the second argument is!?!');
    end

elseif strcmp(control,'peek'),
    [one] = dll_video__peek_tslice(data, varargin{1}, varargin{2});
```

```
elseif strcmp(control, 'sec'),
    if nargout == 1,
        [one, data] = dll_video__next_tslice(data, varargin{1}, varargin{2}, 0);
    else
        [one, two, three, data] = dll_video__next_tslice(data, varargin{1}, varargin{2}, 1);
    end

elseif strcmp(control, 'total_frames'),
    one = data.total_frames - data.curr_frame + 1;

elseif strcmp(control, 'total_sec'),
    one = (data.total_frames - data.curr_frame + 1) / data.fps;


elseif strcmp(control, 'delay_8s'),
    delay_value = varargin{1};

    % can't have delays beyond +/- 0.2
    % leave one extra at end for re-framing
    delay_value = min(round(0.2 * data.fps)-1, delay_value);
    delay_value = max(round(-0.2 * data.fps), delay_value);


    % initialize processed video segment to be used: 7 sec + 1frame for
    % re-framing
    data.curr_frame = round(0.8 * data.fps) + 1 + delay_value;
    data.rewind_curr_frame = data.curr_frame;
    data.overby = 0;
    data.sec7 = 1;

    % set return code.  See help above.
    if data.durration < 8,
        one = 2;
    elseif data.durration > 8,
        one = 1;
    else
        one = 0;
```

```
    end

else
    error('function dll_video:  control value not recognized');
end

% copy back to persistent variable
if fn == 1,
    data1 = data;
elseif fn == 2,
    data2 = data;
end



%%%%%%%%%%%%%%%%%%%%%%%%
function [data] = dll_video__initialize(list);
% Organize all of the data that will be needed.

data.file_name = list{1};
data.file_type = list{2};
data.video_standard = list{3};

data.curr_frame = 1;
data.overby = 0;

if strcmp(data.file_type,'uyvy'),
    data.rows = list{4};
    data.cols = list{5};
    data.fps = list{6};
    fid = fopen(data.file_name,'r');
    fseek(fid,0, 'eof');
    data.total_frames = ftell(fid)./(data.rows*data.cols*2);
    fclose(fid);

else
    error('file type not recognized.  Use ''uyvy'' ');
end
```

```
data.durration = data.total_frames / data.fps;



%%%%%%%%%%%%%%%%%%%%%%%%
function [y] = dll_video__peek_tslice(data, reframe, durration);

if reframe ~=1 & reframe ~= 0,
    error('CALL WRONG -- go back & add reframing argument');
end

[tslice_frames, over_sec] = tslice_conversion(durration, data.fps);
start = data.curr_frame;
stop = data.curr_frame + tslice_frames + reframe - 1;
stop = min(stop, data.total_frames); % don't try to read past end of file!

if strcmp(data.file_type,'uyvy'),
    [y] = read_bigyuv(data.file_name, 'frames', start, stop,  ...
        '128','size',data.rows,data.cols);
end



%%%%%%%%%%%%%%%%%%%%%%%%
function [y, cb, cr, data ] = dll_video__next_tslice( data, reframe, durration, get_color);

% by default, get color & luminance
if nargin < 4,
    get_color = 1;
end
if nargin == 1,
    reframe = 0;
end
if reframe ~=1 & reframe ~= 0,
    error('CALL WRONG -- go back & add reframing argument');
end
```

```
% if optional 'durration' argument exists, override the default values for
% data.tslice_frames and data.over_sec.
if exist('durration', 'var'),
    [tslice_frames, over_sec] = tslice_conversion(durration, data.fps);
else
    tslice_frames = data.tslice_frames;
    over_sec = data.over_sec;
end

if tslice_frames == 0,
    error('Request made to read zero (0) frames of video.');
end

if strcmp(data.file_type,'uyvy'),
    % read video from file.
    if get_color,
        [y,cb,cr] = read_bigyuv(data.file_name, 'frames', data.curr_frame, ...
            data.curr_frame + tslice_frames + reframe - 1,  ...
            '128','size',data.rows,data.cols);
    else
        [y] = read_bigyuv(data.file_name, 'frames', data.curr_frame, ...
            data.curr_frame + tslice_frames + reframe - 1,  ...
            '128','size',data.rows,data.cols);
        cb=0;
        cr=0;
    end
end

% handle overlap
data.curr_frame = data.curr_frame + tslice_frames;
data.overby = data.overby + over_sec;
if data.overby >= 1.0,
    data.curr_frame = data.curr_frame - 1;
    data.overby = data.overby - 1.0;
end

% convert to double -- everything!
y = double(y);
```

```
if get_color,
    cb = double(cb);
    cr = double(cr);
end

if ~get_color,
    cb = data;
end
```

5.26 Function “filter_ati_random.m”

```
function ati = filter_ati_random(y, frames_wide, factor)
% FILTER_ATI_RANDOM
%  Compute absolute value, temporal information (ATI) filter.  Use a random
%  sub-sampling of pixels.
% SYNTAX
%  ati = filter_ati_random(y, frames_wide, factor);
% DEFINITION
%  Compute absolute value, temporal information (ATI) filter on a time-slice of
%  images.  Time-slice must include at least two images temporally!
%  Variable 'y'  must be a three dimensional time-slice (row,col,time);
%
%  Do a 'frames_wide' wide ATI filter (e.g., 'frames_wide' = 2 to use
%  sequential images).  'factor' indicates the fraction of pixels to be used
%  where 0 < factor < 1.0
%
%  Returned variable 'ati' has dimensions (pixels, 1, time) and is
%  appropriate for use in image collapsing but not blocks (e.g.,
%  st_collapse.)


if frames_wide < 1,
    error('''frames_wide'' must be a positive integer');
end
if factor <= 0 | factor >= 1,
    error('''factor'' must be between zero and one');
```

```
end

[rows,cols,time] = size(y);
if time < frames_wide + 1;,
    error('ERROR:  Too few frames in time to compute ATI');
end

need = round(rows * cols * factor);
list_row = round(rand(1,need) * (rows) + 0.5);
list_col = round(rand(1,need) * (cols) + 0.5);
list = (list_col-1)*rows + list_row;

y = reshape(y, rows * cols, 1, time);

hold1 = y(list,1:time-frames_wide);
hold2 = y(list,frames_wide+1:time);
ati = abs(hold1 - hold2);
```

5.27 Function “filter_si_hv_adapt.m”

```
function [si, hv, hvb] = filter_si_hv_adapt(y, filter_size, extra, varargin)
% FILTER_SI_HV_ADAPT
%
%  Filters Y with the NxN gradient (c=2) filters described in SPIE 1999
%  paper.  Like funciton 'filter_si_hv', but can choose other than 13x13.
%
% SYNTAX
%
%  [SI] = filter_si_hv_adapt(Y, N, EXTRA)
%  [SI] = filter_si_hv_adapt(Y, N, EXTRA, rmin, theta)
%  [SI, HV, HVB] = filter_si_hv_adapt(...)
%
% DESCRIPTION
%
%  [SI] = filter_si_hv_adapt(Y, N, EXTRA)  Perceptually fiters liminence image Y using
%  the NxN Horizontal and Vertical gradient filters in a fashion similar
%  to the sobel filter.  Discard the filtered border, of width EXTRA, where
```

```
%  ( EXTRA >= floor(N/2) ).  Filter size, N, must be an odd number.
%
%  If Y is a 3 dimensional matrix, Y will be presumed to contain multiple
%  images as follows:  (row, col, time).  No execution time penalties occur.
%
%  [SI] = filter_si_hv(Y, N, EXTRA, rmin, theta)  allows the user to over ride
%  the default values for rmin and theta.
%
%  [SI, HV, HVB] = filter_si_hv(...)  returns three perceptually fitered
%  versions of image Y:  the SI filtered image, the HV filtered image
%  (containing horiziontal & vertical edges) and the HVB image (containing
%  diagonal edges.)
%
% REMARKS
%
%  rmin defaults to 20, where pixels with a radius (i.e., SI value) less
%  than rmin are set to zero in HV and HVB images.
%
%  Theta defaults to 0.225 radians.  Theta is the maximum angle deviation
%  from the H and V axis for pixels to be considered HV pixels.
%
%  Returned images (SI, HV, and HVB) are the same size as Y; except that a
%  border of EXTRA pixels around the edge of each image is invalid.
%

if mod(filter_size,2) == 0,
    error('SI filter size must be an odd number');
end

% if pass in a time-slice of 2+ images, reshape into 2-D.
if ndims(y) == 3,
    must_reshape = 1;
    [row_size, col_size, time_size] = size(y);
    y = reshape(y, row_size, col_size * time_size);
elseif ndims(y) == 2,
    must_reshape = 0;
    [row_size, col_size, time_size] = size(y);
else
    error('Function ''filter_si_hv'' requires Y to be a 2-D or 3-D image');
```

```
end

%  Assign defaults
[row_size, col_size] = size(y);
rmin = 20;
theta = .225;

if (length(varargin) == 2);
    rmin = varargin{1};
    theta = varargin{2};
end

%
if row_size < filter_size | col_size < filter_size,
    error(sprintf('Function ''filter_si_hv'' requires images to be at least %dx%d', filter_size, filter_size));
end

% compute angle as a ratio of HV and HVbar.
ratio_threshold = tan(theta);

%  The weights for a single row of the H filter
%  is given by: w(x) = k*(x/c)*exp{-(1/2)*(x/c)^2}, where x = {-6, -5, ..., 5, 6},
%  and k is a normalization constant chosen such that this filter produces the same
%  amplitude response on an H V edge as the Sobel filter.

%  Generate the filter_size long filter mask, in one dimension.
filter_mask = filter_h(filter_size);
filter_mask = filter_mask(1,:);

%  Convolve mask with y in horizontal & vertical direction.
%  do two 1xfilter_size convolutions instead of one filter_sizexfilter_size, for speed.
horiz = conv2(y,filter_mask,'same');
horiz = conv2(horiz,ones(filter_size,1),'same');

vert = conv2(y,filter_mask','same');
vert = conv2(vert,ones(1,filter_size),'same');

% for debugging, comment in the below lines.
```

```
% h_in = horiz;
% v_in = vert;

% Construct SI image
si = sqrt(horiz.^2 + vert.^2);

% If use only wants to compute SI, skip HV & HVB.  If need be, reshape back
% into 3-D
if nargout == 1
    if must_reshape == 1,
        si = reshape(si,row_size,col_size/time_size,time_size);
    end
    si = si((extra+1):row_size-extra,(extra+1):col_size/time_size-extra,:);
    return;
end

% Start calculation of HV.
% We don't want to use atan2 (because that is slow) so we are going to
% compute the ratio between h & v, putting the smaller value on top and the
% larger value on the bottom.  Ignore divide by zero, because later code
% checking against rmin will catch that.  Essentially, fold angle into pi/4.
horiz = abs(horiz);
vert = abs(vert);
warning off MATLAB:divideByZero;
ratio = min(horiz,vert) ./ max(horiz,vert);
warning on MATLAB:divideByZero;

clear horiz vert;

% Split image into small values (set to 0) and HV versus HVbar areas.
find_below = find(ratio < ratio_threshold);
find_zeros = find(si <= rmin);

% Start generating HVbar image.  Zero out areas where SI is too small.
hvb = si;
hvb(find_zeros) = 0;

% Generate HV image.  Use HVbar image, so don't have to repeat the zeroing
```

```
% out of small SI values.  Then, zero out HV area.
hv = zeros(row_size,col_size);
hv(find_below) = hvb(find_below);

% Finnish generating HVbar image.  Zero out HVbar area.
hvb(find_below) = 0;

% if needed, reshape back into 3-D
if must_reshape == 1,
    si = reshape(si,row_size,col_size/time_size,time_size);
    hv = reshape(hv,row_size,col_size/time_size,time_size);
    hvb = reshape(hvb,row_size,col_size/time_size,time_size);
end

% take off invalid border around the edge.
    si = si((extra+1):row_size-extra,(extra+1):col_size/time_size-extra,:);
    hv = hv((extra+1):row_size-extra,(extra+1):col_size/time_size-extra,:);
    hvb = hvb((extra+1):row_size-extra,(extra+1):col_size/time_size-extra,:);



function h = filter_h(l)
%  H = FILTER_H(L)
%  Returns a horizontal bandpass filter H (like Figure 27 left, in NTIA
%  Report 02-392) given the filter width L as input.  L must be an odd
%  positive integer greater than or equal to 3.  The vertical bandpass
%  filter is the transpose of this H filter.
%
%  The weights for a single row of the H filter are given by:
%  w(x) = k*(x/c)*exp{-(1/2)*(x/c)^2}, where x = {-m, -(m-1), ..., 0, ..., m-1, m},
%  m = (L-1)/2, and k is a normalization constant chosen such that the filter produces
%  the same amplitude response on a vertical edge as the Sobel filter.
%
%  For the reference 13 x 13 filter, L = 13 and C = 2.  For other filter sizes, C
%  is scaled appropriately, e.g.,
%  for the scaled 9 x 9 filter, L = 9, C = 4/3.
%  for the corresponding 5 x 5 filter, L = 5, C = 2/3.
%
```

```
%  Generate the H filter masks
m = (l-1)/2;  % half filter width
c = 2.0*m/6;  % c = 2 for m = 6, the reference filter
x = -m:m;
h1 = (x/c).*exp(-(1/2)*(x/c).^2);
h = repmat(h1,l,1);

%  Normalize for Sobel filter energy
h = h./(sum(sum(abs(h)))/8);
```

5.28 Function “join_into_frames.m”

```
function [y] = join_into_frames(one, two)
% JOIN_INTO_FRAMES
%  Join two fields into one frame.
% SYNTAX
%  [y] = join_into_frames(one, two);
% DESCRIPTION
%  Create frame 'y' from field one ('one') and field two ('two').  Field two
%  contains the top line of the image; field one contains the second line
%  of the image.  For NTSC, field one occurs before field two in time.  For
%  PAL, the reverse is the case.  Y can be a time-slice of frames.
%
%  If image 'one' and 'two' are not identically sized, then an error will occur.
%  See also function 'split_into_fields'

[row, col, time] = size(one);
[row2, col2, time2] = size(two);

if row ~= row2 || col ~= col2 || time ~= time2,
    error('two fields must be identically sized to be joined into a frame');
end

y(1,:,:,:) = two;
y(2,:,:,:) = one;
y = reshape(y, row*2, col, time);
```

5.29 Function “max_filterw.m”

```
function [result] = max_filterw(feat,w)
% MAX_FILTERW
%   Perform a max filter of width 'w' on one or more 1-D arrays.
% SYNTAX
%   [result] = max_filterw(feat,w)
% DESCRIPTION
%  'feat' is either a 1-dimension array, where the first dimension contains
%         the data (i.e., a column vector), or a 2-dimension array, in which
%         case the function operates independently on each column.  Put
%         another way, 'feat' is either a column vector or a matrix of column
%         vectors.
%  'w' is the filter width, which must be odd.
%
%  Each point is replaced by the maximum of itself and the adjacent points
%  (in that column) +/- ((w-1)/2) neighboring points on either side.  This is
%  aplied to each column vector independently.  At the end points, zero
%  buffering is used (e.g., the beginning point presumes that zeros occured
%  previously).

if mod(w,2) == 0 | w <= 1,
    error('Argument ''w'' must be an odd number greater than one');
end

[nr,nc] = size(feat);

% mat0 holds the zero padded, centered array
mat0 = zeros(nr+(w-1),nc);
mat0(1+floor(w/2):nr+floor(w/2),:) = feat;

% do negative shifts
result = mat0;
for i = -1:-1:-1*floor(w/2)
    result = max(result,circshift(mat0,i));
end
```

```
% do positive shifts
for i = 1:floor(w/2)
    result = max(result,circshift(mat0,i));
end

result = result(1+floor(w/2):nr+floor(w/2),:);
```

5.30 Function “model_fastlowbw_features.m”

```
function [si_std hv_ratio y_mean cb_mean cr_mean ati_rms] = ...
    model_fastlowbw_features (mode, one, two, three, four, five, six);
% MODEL_fastlowbw_features
%   Compute the original features for the ITS fast low bandwidth model.
% SYNTAX
%   model_fastlowbw_features ('memory',  y, cb, cr, fps, filter_size, extra);
%
%   [si_std, hv_ratio, y_mean, cb_mean, cr_mean, ati_rms] = ...
%       model_fastlowbw_features ('eof');
%
%   model_fastlowbw_features ('clear');
% DESCRIPTION
%   When the first argument is the string 'memory', this function takes one
%   second (EXACTLY) of video already in memory, in the YCbCr colorspace
%   (y, cb, and cr respectively).  All three variables are formatted
%   (row,col,time); and (fps) is the number of frames per second (e.g.,
%   29.97, 30, 25, 15, etc).  'filter_size' and 'extra' are from function
%   'adaptive_filter' when given the current image size.
%   Size of y, cb, & cr spatially should be exactly ROI returned by
%   function model_lowbw_sroi.  See 'eof.
%
%   When the first argument is the string 'eof', this function completes
%   the feature calculations, empties internal buffers, and returns the
%   feature stream.  See 'memory'.
%
%   When the first argument is the string 'clear', all internall bufferes
%   are emptied.  This is done when 'eof' is called, also.
```

```
%
%   Warning:  uses 1 set of internal buffers, so don't mix sequences.
%
%   Return variables when called with 'eof' are the six feature streams:
%   - si_std containing standard deviation of the spatial information (SI)
%   - hv_ratio containing the ratio of HV to HVbar energy
%   - cb_mean containing the average Cb value
%   - cr_mean containing the averge Cr value
%   - ati_rms containing the root mean squared of absolute value of
%     temporal information (TI)

persistent hold_si_std;
persistent hold_hv_ratio;
persistent hold_y_mean;
persistent hold_cb_mean;
persistent hold_cr_mean;
persistent hold_ati_rms;
persistent buffer_y;


if strcmpi(mode,'memory') && nargin == 7,
    tis_sec = 0.2;
    [tis_frames] = tslice_conversion(tis_sec, four);

    [si_std hv_ratio y_mean cb_mean cr_mean ati_rms] = ...
        model_fastlowbw_features_memory (one, two, three, four, buffer_y, five, six);

    if length(hold_si_std) == 0,
        hold_si_std = si_std;
        hold_hv_ratio = hv_ratio;
        hold_y_mean = y_mean;
        hold_cb_mean = cb_mean;
        hold_cr_mean = cr_mean;
        hold_ati_rms = ati_rms;
    else
        [row,col,time1] = size(hold_si_std);
        hold_si_std(:,:,time1+1) = si_std;
        hold_hv_ratio(:,:,time1+1) = hv_ratio;
```

```
        hold_y_mean(:,:,time1+1) = y_mean;
        hold_cb_mean(:,:,time1+1) = cb_mean;
        hold_cr_mean(:,:,time1+1) = cr_mean;

        [row,col,time1] = size(hold_ati_rms);
        [row,col,time2] = size(ati_rms);
        hold_ati_rms(:,:,time1+1:time1+time2) = ati_rms;
    end
    [row,col,time] = size(one);
    buffer_y = one(:,:,(time-tis_frames+1):time);
elseif strcmpi(mode,'eof') && nargin == 1,

    si_std = hold_si_std;
    hv_ratio = hold_hv_ratio;
    y_mean = hold_y_mean;
    cb_mean = hold_cb_mean;
    cr_mean = hold_cr_mean;
    ati_rms = hold_ati_rms;
    model_fastlowbw_features('clear');

elseif strcmpi(mode,'clear'),
    hold_si_std = [];
    hold_hv_ratio = [];
    hold_y_mean = [];
    hold_cb_mean = [];
    hold_cr_mean = [];
    hold_ati_rms = [];
    buffer_y = [];
else
    error('argument list not recognized');
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [si_std hv_ratio y_mean cb_mean cr_mean ati_rms] = ...
    model_fastlowbw_features_memory (y, cb, cr, fps, buffer_y, filter_size, extra);
```

```
HV_THRESHOLD = 4.0;

tis_sec = 0.2;
[tis_frames] = tslice_conversion(tis_sec, fps);

% loop through video
tslice_frames = round(fps);
ati_curr = 1;
[row,col,time] = size(y);
rng1 = (extra+1):(row-extra);
rng2 = (extra+1):(col-extra);

% Calculate features.
ym = mean(y,3);
[si, hv, hvb] = filter_si_hv_adapt(ym, filter_size, extra);
[si_std] = block_statistic(si, 30, 30, 'std');
[hv_mean] = block_statistic(hv, 30, 30, 'mean');
[hvb_mean] = block_statistic(hvb, 30, 30, 'mean');
[r,c,t] = size(y);
[y_mean] = block_statistic(ym(rng1, rng2, :), 30, 30, 'mean');
clear si hv hvb;

% Compute Cb, Cr
[cb_mean] = block_statistic(cb(rng1,rng2,:), 30, 30, 'mean');
[cr_mean] = block_statistic(cr(rng1,rng2,:), 30, 30, 'mean');

% compute YCbCr 0.2s ATI on frames
t1 = 0;
if length(buffer_y) > 0,
    [r,c,t1] = size(buffer_y);
    [r,c,t2] = size(y);
    ati_y(:,:,1:t1) = buffer_y(rng1,rng2,:);
    ati_y(:,:,t1+1:t1+t2) = y(rng1,rng2,:);
else
    ati_y = y;
end

% use 5% of pixels, randomly chosen
```

```
ati_y = filter_ati_random(ati_y, tis_frames, 0.05);

% have time-differences; now compute ATI
[rowcol,time] = size(ati_y);
for cnt = 1:time,
    ati_rms(1,1,ati_curr) = block_statistic( ati_y(:,cnt), rowcol, 1, 'rms');
    ati_curr = ati_curr + 1;
end

hv_ratio = max(HV_THRESHOLD, hv_mean) ./ max(HV_THRESHOLD, hvb_mean);
```

5.31 Function “model_fastlowbw_features_shift.m”

```
function [data] = model_fastlowbw_features_shift (mode, one, two, three, four, five, six);
% model_fastlowbw_features_shift
%   Compute the processed features for the ITS fast low bandwidth model.
% SYNTAX
%   model_fastlowbw_features_shift ('memory',  y, cb, cr, fps, filter_size, extra)
%   [data] = model_fastlowbw_features_shift ('eof')
%   model_fastlowbw_features_shift ('clear')
% DESCRIPTION
%   When the first argument is the string 'memory', this function takes one
%   second (EXACTLY) of video already in memory, in the YCbCr colorspace
%   (y, cb, and cr respectively).  All three variables are formatted
%   (row,col,time); and (fps) is the number of frames per second (e.g.,
%   29.97, 30, 25, 15, etc).  'filter_size' and 'extra' are from function
%   'adaptive_filter' when given the current image size.
%   Size of y, cb, & cr spatially should be exactly ROI returned by
%   function model_lowbw_sroi.  See 'eof.
%
%   When the first argument is the string 'eof', this function completes
%   the feature calculations, empties internal buffers, and returns the
%   feature stream.  See 'memory'.
%
%   When the first argument is the string 'clear', the internal buffers are
%   emptied.  This is done when called with 'eof', also.
%
```

```
%   Warning:  uses 1 set of internal buffers, so don't mix sequences.
%
%   Return value 'data' is contains for each of the 9 pixel shifts --
%   data(1) .. data(9) -- the following features as structure elements:
%           si_std hv_ratio y_mean cb_mean cr_mean ati_rms
%   See function model_fastlowbw_features for a brief description of each
%   element in this structure.



persistent buffer;
persistent buffer_y;

BSIZE = 30;


if strcmpi(mode,'memory') && nargin == 7,
    tis_sec = 0.2;
    [tis_frames] = tslice_conversion(tis_sec, four);

    [curr] = ...
        model_fastlowbw_features_memory (one, two, three, four, buffer_y, five, six, BSIZE);

    if length(buffer) == 0,
        buffer = curr;
    else
        [row,col,time1] = size(buffer(1).si_std);
        [row,col,time2] = size(buffer(1).ati_rms);
        [row,col,time3] = size(curr(1).ati_rms);
        for loop = 1:9,
            buffer(loop).si_std(:,:,time1+1) = curr(loop).si_std;
            buffer(loop).hv_ratio(:,:,time1+1) = curr(loop).hv_ratio;
            buffer(loop).y_mean(:,:,time1+1) = curr(loop).y_mean;
            buffer(loop).cb_mean(:,:,time1+1) = curr(loop).cb_mean;
            buffer(loop).cr_mean(:,:,time1+1) = curr(loop).cr_mean;

            buffer(loop).ati_rms(:,:,time2+1:time2+time3) = curr(loop).ati_rms;
        end
```

```
    end
    [row,col,time] = size(one);
    buffer_y = one(:,:,(time-tis_frames+1):time);
elseif strcmpi(mode,'eof') && nargin == 1,

    data = buffer;
    model_fastlowbw_features_shift('clear');

elseif strcmpi(mode,'clear'),
    buffer = [];
    buffer_y = [];
else
    error('argument list not recognized');
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [curr] = ...
    model_fastlowbw_features_memory (y, cb, cr, fps, buffer_y, filter_size, extra, BSIZE);

HV_THRESHOLD = 4.0;

tis_sec = 0.2;
[tis_frames] = tslice_conversion(tis_sec, fps);

% loop through video
tslice_frames = round(fps);
ati_curr = 1;
[row,col,time] = size(y);
rng1 = (extra+1):(row-extra);
rng2 = (extra+1):(col-extra);

% Calculate features.
ym = mean(y,3);
[si, hv, hvb] = filter_si_hv_adapt(ym, filter_size, extra);
```

```
si_std = block_statistic_shift(si, BSIZE, BSIZE, 'std');
hv_mean = block_statistic_shift(hv, BSIZE, BSIZE, 'mean');
hvb_mean = block_statistic_shift(hvb, BSIZE, BSIZE, 'mean');
y_mean = block_statistic_shift(ym(rng1,rng2,:), BSIZE, BSIZE, 'mean');
cb_mean = block_statistic_shift(cb(rng1,rng2,:), BSIZE, BSIZE, 'mean');
cr_mean = block_statistic_shift(cr(rng1,rng2,:), BSIZE, BSIZE, 'mean');

clear si hv hvb;
for loop = 1:9,
    curr(loop).si_std = si_std(loop).std;
    curr(loop).hv_mean = hv_mean(loop).mean;
    curr(loop).hvb_mean = hvb_mean(loop).mean;
    curr(loop).y_mean = y_mean(loop).mean;
    curr(loop).cb_mean = cb_mean(loop).mean;
    curr(loop).cr_mean = cr_mean(loop).mean;
end

% compute Y 0.2s ATI on frames
t1 = 0;
if length(buffer_y) > 0,
    [r,c,t1] = size(buffer_y);
    [r,c,t2] = size(y);
    ati_y(:,:,1:t1) = buffer_y(rng1,rng2,:);
    ati_y(:,:,t1+1:t1+t2) = y(rng1,rng2,:);
else
    ati_y = y;
end

% use 5% of pixels, randomly chosen
ati_y = filter_ati_random(ati_y, tis_frames, 0.05);

% have time-differences; now compute ATI
[rowcol,time] = size(ati_y);
for cnt = 1:time,
    hold = block_statistic( ati_y(:,cnt), rowcol, 1, 'rms');
    for loop = 1:length(curr),
        curr(loop).ati_rms(1,1,ati_curr) = hold;
    end
    ati_curr = ati_curr + 1;
```

```
end

for loop = 1:length(curr),
    curr(loop).hv_ratio = max(HV_THRESHOLD, curr(loop).hv_mean) ...
        ./ max(HV_THRESHOLD, curr(loop).hvb_mean);
end
```

5.32 Function “model_fastlowbw_parameters.m”

```
function [vqm, hv_loss_par, hv_gain_par,si_loss_par, si_gain_par, ...
    color_comb_par, noise_par, error_par] = model_fastlowbw_parameters (...
    si_proc, hv_feat_proc, y_proc, cb_proc, cr_proc, ati_proc, ...
    si_orig, hv_feat_orig, y_orig, cb_orig, cr_orig, ati_orig, ...
    fps, part_si_min, part_si_max, part_hv_min, part_hv_max, ...
    part_c_min, part_c_max, part_c, part_ati_min, part_ati_max, ...
    part_ati, code_ati, do_test_print, TIME_DELTA);
% MODEL_FASTLOWBW_PARAMETERS
%     Compute Fast Low Bandwidth model from original and processed
%     features.
%
%     Each Video Quality Metric (VQM) prediction of the FastLowBandwidth model
%     uses the previous 'time_delta' seconds of video.  The model slides
%     along the time-history of features (i.e., overlapping in time) to yield
%     a continuous time-history of VQM, each based on the previous
%     'time_delta' seconds of video.
% SYNTAX
% [vqm, hv_loss_par, hv_gain_par,si_loss_par, si_gain_par, ...
%     color_comb_par, noise_par, error_par] = model_fastlowbw_parameters (...
%     si_proc, hv_feat_proc, y_proc, cb_proc, cr_proc, ati_proc, ...
%     si_orig, hv_feat_orig, y_orig, cb_orig, cr_orig, ati_orig, ...
%     fps, part_si_min, part_si_max, part_hv_min, part_hv_max, ...
%     part_c_min, part_c_max, part_c, part_ati_min, part_ati_max, ...
%     part_ati, code_ati, do_test_print);
% [vqm, hv_loss_par, hv_gain_par,si_loss_par, si_gain_par, ...
%     color_comb_par, noise_par, error_par] = model_fastlowbw_parameters (...
%     si_proc, hv_feat_proc, y_proc, cb_proc, cr_proc, ati_proc, ...
%     si_orig, hv_feat_orig, y_orig, cb_orig, cr_orig, ati_orig, ...
%     fps, part_si_min, part_si_max, part_hv_min, part_hv_max, ...
```

```
%     part_c_min, part_c_max, part_c, part_ati_min, part_ati_max, ...
%     part_ati, code_ati, do_test_print, TIME_DELTA);
% DESCRIPTION
%  See model_fastlowbw_features.m for description of original features
%  variables ( si_orig, hv_feat_orig, y_orig, cb_orig, cr_orig, ati_orig )
%
%  See model_fastlowbw_features_shift.m for description of processed
%  feature variables ( si_proc, hv_feat_proc, y_proc, cb_proc, cr_proc, ati_proc )
%
%  See model_lowbw_compression.m for description of input variables
%  relating to compression (part_si_min through code_ati)
%
%  'fps' is the frame rate of the video
%  'do_test_print' is 1 to print information and 0 typically.
%  'time_delta' is the number of seconds of video used to compute the
%               FastLowBandwidth model.  If you want a single prediction
%               instead of a continuous history of values, then
%               'time_delta' should be the length of the video sequence in
%               seconds, rounded down (e.g., the 3rd dimension of si_orig).
%               'time_delta' defaults to ten seconds (10).  This is the
%               recommended value.  'time_delta' should never be greater
%               than 30 seconds, because different perceptual decisions
%               apply to longer sequences (e.g., recency effects) and the
%               FastLowBandwidth model does not take those factors into
%               account.
%
%   Return values are as follows.  Each is a continuous time-history of
%   model predictions, one per second, starting at 1-second.  The values
%   from seconds 1 through 4 aren't worth much, since it is questionable
%   (perceptually speaking) to rate the quality of a very short video
%   sequence.  If you want a single prediction for an entire short video
%   sequence (e.g., 5 to 30 seconds), then use the last value in each of
%   these arrays and discard the prior values.
%   'vqm' is the FastLowBandwidth model's video quality prediction.
%   The other return values ( 'hv_loss_par', 'hv_gain_par', 'si_loss_par'
%   'si_gain_par', 'color_comb_par', 'noise_par', and 'error_par')
%   are the seven parameters.  The model weights are applied to the
%   parameters already, so that you can sum up these values to get 'vqm'.
%   Thus, the weighted parameter values indicate the perceptual impact of
```

```
%   each parameter on the viewer (e.g., how much of the quality drop was
%   due to that factor).

if ~exist('TIME_DELTA'),
    TIME_DELTA = 10;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[hv_loss_par, hv_gain_par,si_loss_par, si_gain_par, ...
    color_comb_par, noise_par, error_par] = calc_parameters (...
    si_proc, hv_feat_proc, y_proc, cb_proc, cr_proc, ati_proc, ...
    si_orig, hv_feat_orig, y_orig, cb_orig, cr_orig, ati_orig, ...
    fps, part_si_min, part_si_max, part_hv_min, part_hv_max, ...
    part_c_min, part_c_max, part_c, part_ati_min, part_ati_max, ...
    part_ati, code_ati, TIME_DELTA);

% clear extra variables
clear ati_orig ati_proc cb_orig cb_proc col cr_orig cr_proc hv_feat_orig;
clear hv_feat_proc part_ati part_ati_max part_ati_min part_c part_c_max;
clear part_c_min part_hv_max part_hv_min part_si_max part_si_min;
clear range row si_orig si_proc y_orig y_orig2 y_proc y_proc2;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Fix results
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
hv_loss_par = [hv_loss_par(1) hv_loss_par'];
hv_gain_par = [hv_gain_par(1) hv_gain_par'];
si_loss_par = [si_loss_par(1) si_loss_par'];
color_comb_par = [color_comb_par(1) color_comb_par'];

% si_gain, noise_par, & error_par don't need a fix -- no macro blocks!

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate VQM
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if do_test_print,
```

```
    % for testing purposes only!
    fprintf('print out parameters & VQM for testing\n');
    save raw_pars.mat hv_loss_par hv_gain_par si_loss_par si_gain_par color_comb_par noise_par error_par;
end

% ***Weights updated
DC_WEIGHT = 0.0;
HV_LOSS_WEIGHT = 0.38317338378290;
HV_GAIN_WEIGHT = 0.37313218013131;
SI_LOSS_WEIGHT = 0.58033514546526;
SI_GAIN_WEIGHT = 0.95845512360511;
COLOR_WEIGHT = 1.07581708014998;
NOISE_WEIGHT = 0.17693274495002;
ERROR_WEIGHT = 0.02535903906351;

% upsample (1 sample per sec --> 2 samples per sec) and  weight each parameter.
hv_loss_par = my_interp(2, hv_loss_par) * HV_LOSS_WEIGHT;
hv_gain_par = my_interp(2, hv_gain_par) * HV_GAIN_WEIGHT;
si_loss_par = my_interp(2, si_loss_par) * SI_LOSS_WEIGHT;
si_gain_par = my_interp(2, si_gain_par) * SI_GAIN_WEIGHT;
color_comb_par = my_interp(2, color_comb_par) * COLOR_WEIGHT;
noise_par = noise_par * NOISE_WEIGHT;
error_par = error_par * ERROR_WEIGHT;

% clear a few variables
clear *par1 *par2 *orig *proc;
clear col video_dir ans code_ati oclip otest part* range row time;

% Even out lengths, if needed.  This should never happen if the video
% sequences are 1-minute long each.
hold = [length(hv_loss_par) length(hv_gain_par) length(si_loss_par) length(si_gain_par) ...
    length(color_comb_par) length(noise_par) length(error_par)];
if min(hold) ~= max(hold),
%    fprintf('WARNING:  time-length of features being rectified\n');
    hold = min(hold);
    hv_loss_par = hv_loss_par(1:hold);
    hv_gain_par = hv_gain_par(1:hold);
    si_loss_par = si_loss_par(1:hold);
    si_gain_par = si_gain_par(1:hold);
```

```
    color_comb_par = color_comb_par(1:hold);
    noise_par = noise_par(1:hold);
    error_par = error_par(1:hold);
end

%  Final VQM has clipping at low end and crushing at high end
vqm = DC_WEIGHT + hv_loss_par + hv_gain_par + si_loss_par + si_gain_par + ...
    color_comb_par + noise_par + error_par;

% Clip VQM for values less than zero
vqm(find(vqm < 0)) = 0.0;  % No quality improvements allowed

% Compress VQM values greater than 1 using standard crushing function
c = 0.5;
crush = find(vqm > 1);
vqm(crush) = (1 + c)*vqm(crush) ./ (c + vqm(crush));



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [data2] = my_interp(times, data1);
% perform upsampling (1 sample per second to 2 samples per second)
% put extra sample at the beginning (a repeat of the first sample) rather
% than at the end.

if times ~= 2,
    error('Only implemented for 2');
end

hold = length(data1);
data2(1) = data1(1);
data2(2:2:hold*2,1) = data1;
data2(3:2:hold*2,1) = ( data1(2:hold) + data1(1:hold-1) ) / 2;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [hv_loss_par, hv_gain_par,si_loss_par, si_gain_par, ...
    color_comb_par, noise_par, error_par] = calc_parameters (...
    si_proc, hv_feat_proc, y_proc, cb_proc, cr_proc, ati_proc, ...
    si_orig, hv_feat_orig, y_orig, cb_orig, cr_orig, ati_orig, ...
    fps, part_si_min, part_si_max, part_hv_min, part_hv_max, ...
    part_c_min, part_c_max, part_c, part_ati_min, part_ati_max, ...
    part_ati, code_ati, TIME_DELTA);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Model calculation.  Change from collapse all frames, to running-collapse
% of TIME_DELTA seconds at once.  Remove multiple-clip functionality.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ATI_SEC = 0.2; % width of ATI time-difference
ATI_WIDTH  = tslice_conversion (ATI_SEC, fps);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% HV loss and gain parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Zero parameter values whose hv_feat_orig < part_hv(1)
hvgain_low = find(hv_feat_orig < part_hv_min);
% 0.435 > part_hv(1), better quantizing threshold for hv_loss
hvloss_low = find(hv_feat_orig < 0.435);
% Zero parameter values whose hv_feat_orig > part_hv(code_hv_size-1)
hvloss_high = find(hv_feat_orig > part_hv_max);
% 1.90 < hv_part(code_hv_size-1), better quantizing threshold for hv_gain
hvgain_high = find(hv_feat_orig > 1.90);

% Implement the hv_loss block parameter
hv_loss = (hv_feat_proc-hv_feat_orig)./hv_feat_orig;  % ratio loss calculation
hv_loss = min(hv_loss,0);  % negative part calculation
hv_loss(hvloss_low) = 0.0;
hv_loss(hvloss_high) = 0.0;

% Implement the hv_gain block parameter
```

```
hv_gain = log10(hv_feat_proc./hv_feat_orig);  % log10 gain calculation
hv_gain = max(hv_gain,0);  % positive part calculation
hv_gain(hvgain_low) = 0.0;
hv_gain(hvgain_high) = 0.0;

% Calculate the si_weight function.  If low spatial detail, reduce weight:
% Linear weight reduction, weight goes from weight_low to 1 as spatial goes
% from a to b
[frows,fcols,ftime] = size(si_orig);
si_weight = ones(frows, fcols, ftime);
weight_low = 0;
a = 5;
b = 25;
weight_low_slope = (1-weight_low)/(b-a);
si_weight(find(si_orig < a)) = weight_low;
low = find(si_orig >= a & si_orig < b);
si_weight(low) = weight_low_slope*(si_orig(low)-a) + weight_low;
hv_loss = hv_loss.*si_weight;
clear si_weight;

    % Pick off the valid region for this clip, using same rules as
    % read_feature.

    hv_loss_par_clip = squeeze(hv_loss(:,:,:));
    hv_gain_par_clip = squeeze(hv_gain(:,:,:));

    % y weighting function by macroblocks for less bandwidth
    weight_high = 0;
    c = 175;
    d = 255;
    weight_high_slope = (weight_high-1)/(d-c);
    y_orig_clip = squeeze(y_orig(:,:,:));
    [pr, pc, pt] = size(y_orig_clip);
    y_weight = ones(pr, pc, pt);

    % Quantize y_orig average macroblock values which will be RR info
    high = find(y_orig_clip > c & y_orig_clip <= d);
    y_weight(high) = weight_high_slope*(y_orig_clip(high)-c) + 1;
```

```
    y_weight(find(y_orig_clip > d)) = weight_high;

    % OMB(3,3,2)below1%_minkowski(1,1.5) for hv_loss
    hv_loss_par_clip = hv_loss_par_clip.*y_weight;
    hv_loss_par_clip = st_collapse('below1%', hv_loss_par_clip, 'OverlapMacroBlock', 3, 3, 2);
    hv_loss_par_clip = running_collapse ('minkowski(1,1.5)', hv_loss_par_clip, TIME_DELTA-1, '3D');
    hv_loss_par = hv_loss_par_clip;

    % OMB(3,3,2)above99%_minkowski(1.5,3) for hv_gain
    hv_gain_par_clip = hv_gain_par_clip.*y_weight;
    % Clip the low end and subtract this clipping level
    hv_gain_par_clip = max(hv_gain_par_clip, 0.06) - 0.06;
    hv_gain_par_clip = st_collapse('above99%', hv_gain_par_clip, 'OverlapMacroBlock', 3, 3, 2);
    hv_gain_par_clip = running_collapse ('minkowski(1.5,3)', hv_gain_par_clip, TIME_DELTA-1, '3D');
    hv_gain_par = hv_gain_par_clip;

clear hv_loss hv_gain y_orig;

% Clip the low end of hv_loss and subtract this clipping level
hv_loss_clip = 0.08;
hv_loss_par = max(hv_loss_par, hv_loss_clip) - hv_loss_clip;

% Compress hv_gain values greater than training using crushing function.
% High clipping level of 0.75 is approximately the maximum from the
% training data.
hv_gain_max = 0.75;  % from training data
crush_hv = find(hv_gain_par > hv_gain_max);  % these values will be crushed
hv_gain_par(crush_hv) = (hv_gain_max + 0.25)*hv_gain_par(crush_hv) ./ (0.25 + hv_gain_par(crush_hv));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  SI loss and gain parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
si_t = part_si_min;  % si clipping threshold

% Parameters outside of the si_orig quantizer range will be zeroed.
si_high = find(si_orig > part_si_max);

%  Clip si at low end
```

```
si_proc = max(si_proc,si_t);
si_orig = max(si_orig,si_t);

% Implement the si_loss parameter
si_loss = (si_proc-si_orig)./si_orig;  % ratio loss calculation
si_loss = min(si_loss,0);  % negative part calculation
si_loss(si_high) = 0.0;

% Implement the si_gain parameter
si_gain = log10(si_proc./si_orig);  % log gain calculation
si_gain = max(si_gain,0);  % positive part calculation
si_gain(si_high) = 0.0;

clear si_orig si_proc si_high;

    % Pick off the valid region for this clip, using same rules as
    % read_feature.
    si_loss_par_clip = squeeze(si_loss(:,:,:));
    si_gain_par_clip = squeeze(si_gain(:,:,:));

    % OMB(3,3,2)minkowski(1,2)_minkowski(1.5,2.5) for si_loss
    si_loss_par_clip = si_loss_par_clip.*y_weight;
    si_loss_par_clip = st_collapse('minkowski(1,2)', si_loss_par_clip, 'OverlapMacroBlock', 3, 3, 2);
    si_loss_par_clip = running_collapse ('minkowski(1.5,2.5)', si_loss_par_clip, TIME_DELTA-1, '3D');
    si_loss_par = si_loss_par_clip;

    % above95%tail_minkowski(1.5,2) for si_gain
    [pr, pc, pt] = size(si_gain_par_clip);  % Find size after picking off valid
    % Clip the low end and subtract this clipping level
    si_gain_par_clip = max(si_gain_par_clip, 0.1) - 0.1;
    si_gain_par_clip = st_collapse('above95%tail', reshape(si_gain_par_clip, pr*pc,pt));
    si_gain_par_clip = running_collapse ('minkowski(1.5,2)', si_gain_par_clip, TIME_DELTA, '3D');
    si_gain_par = si_gain_par_clip;

clear si_gain si_loss;

% Clip the low end of si_loss and subtract this clipping level
si_loss_clip = 0.12;
```

```
si_loss_par = max(si_loss_par, si_loss_clip) - si_loss_clip;

% Compress si_gain values greater than training using crushing function.
% High clipping level of 0.48 is approximately the maximum from the
% training data.
si_gain_max = 0.48;  % from training data
crush_si = find(si_gain_par > si_gain_max);  % these values will be crushed
si_gain_par(crush_si) = (si_gain_max + 0.25)*si_gain_par(crush_si) ./ (0.25 + si_gain_par(crush_si));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% color (Cb & Cr) parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Parameters outside of the quantizer ranges will be zeroed, including the
% bin that quantizes to zero.
cb_zero_high = find(cb_orig <= part_c_min | cb_orig >= part_c_max | cb_orig == 0);
cb_orig(cb_zero_high) = 0.0;
cb_proc(cb_zero_high) = 0.0;
clear cb_zero__high;
cr_zero_high = find(cr_orig <= part_c_min | cr_orig >= part_c_max | cr_orig == 0);
cr_orig(cr_zero_high) = 0.0;
cr_proc(cr_zero_high) = 0.0;
clear cb_zero_high cr_zero_high;

% Color extreme parameter with modifications to Euclidean distance.
% Better distance measure for color extreme parameter.
p = 1.0;  % Normal Euclidean uses p = 2.0, this absolute diff works better
r = 0.5;
w = 1.5;
color_euclid = (abs(cb_proc-cb_orig).^p + (w*abs(cr_proc-cr_orig)).^p).^r;
clear cb_proc cr_proc cb_orig cr_orig;

    % Pick off the valid region for this clip, using same rules as
    % read_feature.
    color_extreme_par_clip = squeeze(color_euclid(:,:,:));

    color_spread_par_clip = color_extreme_par_clip;
```

```
    % OMB(3,3,2)above99%_minkowski(0.5,1) for color_extreme
    color_extreme_par_clip = st_collapse('above99%', color_extreme_par_clip, 'OverlapMacroBlock',3,3,2);
    color_extreme_par_clip = running_collapse ('minkowski(0.5,1)', ...
        color_extreme_par_clip, TIME_DELTA-1, '3D');
    color_extreme_par = color_extreme_par_clip;

    % OMB(3,3,2)minkowski(2,4)_90% for color_spread
    color_spread_par_clip = st_collapse('minkowski(2,4)', color_spread_par_clip,...
        'OverlapMacroBlock',3,3,2);
    color_spread_par_clip = running_collapse ('90%', color_spread_par_clip,...
        TIME_DELTA-1, '3D');
    color_spread_par = color_spread_par_clip;

clear color_euclid;

% compute the color combination parameter
color_comb_par = -0.617958*color_spread_par + 0.691686*color_extreme_par;

%  Clip the color combination parameter
color_comb_clip = 0.114;
color_comb_par = max(color_comb_par, color_comb_clip) - color_comb_clip;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ATI parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ati_nt = part_ati(24);  % ati clipping threshold for noise, approximately 5
ati_et = part_ati(57);  % ati clipping threshold for error, approximately 12

% Clip processed features if they exceed the maximum quantizer value
code_ati_size = size(code_ati,2);
ati_proc(find(ati_proc > part_ati(code_ati_size-1))) = code_ati(code_ati_size);

% Do the ATI parameters at each temporal alignment from plus to minus 0.4 seconds.
try_num = 0;
ati_search = floor(ceil(fps) * 0.4);
whole_length = min(length(ati_proc), length(ati_orig));
ati_proc = ati_proc((ati_search+1):(whole_length-ati_search));
```

```
whole_ati_orig = ati_orig;

for try_time = -ati_search:ati_search,
    try_num = try_num + 1;
    clear ati_nproc ati_norig;
    ati_orig = whole_ati_orig((ati_search+1+try_time):(whole_length-ati_search+try_time));

    %  Clip ati at low end for noise calculation
    ati_nproc = max(ati_proc,ati_nt);
    ati_norig = max(ati_orig,ati_nt);

    % Implement the ati gain parameter that will be used for noise
    ati_ngain = (ati_nproc-ati_norig)./ati_norig;  % ratio gain calculation
    ati_ngain = max(ati_ngain,0);  % positive part calculation

    % between25%50% for noise_par
    noise_par_clip = running_collapse ('between25%50%', ati_ngain, ...
        TIME_DELTA * ceil(fps) - ATI_WIDTH, '3D');

    % pick off samples, one every half second
    pattern = ceil( length(noise_par_clip):-ceil(fps)/2:1 );
    % reverse order & extend each end by one.
    pattern = [ pattern(length(pattern)) pattern(length(pattern):-1:1) pattern(1)];
    % error check the length of pattern.  This code will only be triggered
    % if some really strange frame rate is chosen.  Then, instead of adding
    % up to exactly 1sec, the discards might round to a different number.
    if ftime * 2 ~= length(pattern),
        while length(pattern) < ftime * 2,
            pattern = [ pattern(1) pattern ];
        end
        while length(pattern) > ftime * 2,
            pattern = pattern(2:length(pattern));
        end
    end

    noise_par(:,try_num) = noise_par_clip(pattern);

    % Set-up to calculate the error parameter
```

```
    this_proc = ati_proc;
    this_orig = ati_orig;

    % 7-point Max filter
    this_proc = max_filterw(squeeze(this_proc),7);
    this_orig = max_filterw(squeeze(this_orig),7);

    %  Clip ati at low end for error calculation
    this_proc = max(this_proc,ati_et);
    this_orig = max(this_orig,ati_et);

    % Implement the ati gain parameter that will be used for errors
    ati_egain = (this_proc-this_orig)./this_orig;  % ratio gain calculation
    ati_egain = max(ati_egain,0);  % positive part calculation

    % above90% for error_par
    error_par_clip = running_collapse ('above90%', ati_egain, TIME_DELTA * ceil(fps) - ATI_WIDTH, '3D');
    error_par(:,try_num) = error_par_clip(pattern);

end

% Take minimum at each point in time of the temporal shifts
noise_par = min(noise_par')';
error_par = min(error_par')';

clear ati_nproc ati_norig;
clear ati_ngain ati_proc ati_orig;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [dataout] = macro_interp(datain);
%  Takes an input 3d array dimensioned as (r,c,t) and adds interpolated
%  time samples halfway between.  So the output array will have dimension
%  (r,c,2t-1).

[r,c,t] = size(datain);
```

```
dataout = zeros(r,c,2*t-1);

for i = 1:2*t-1
    if (floor(i/2) == i/2) % interpolate
        dataout(:,:,i) = (datain(:,:,i/2)+datain(:,:,1+i/2))/2;
    else % don't interpolate
        dataout(:,:,i) = datain(:,:,(i+1)/2);
    end
end
```

5.33 Function “model_lowbw_compression.m”

```
function [si_orig, part_si_min, part_si_max, ...
        hv_feat_orig, part_hv_min, part_hv_max, y_orig, ...
        cb_orig, cr_orig, part_c_min, part_c_max, part_c, ...
        ati_orig, part_ati_min, part_ati_max, part_ati, code_ati ] ...
 = model_lowbw_compression (mode, transmit_file, si_orig, hv_feat_orig, luma_orig, cb_orig, cr_orig, ati_orig);
% MODEL_LOWBW_COMPRESSION
%   Compress and write original features from the LowBW or FastLowBW model.
%   Alternatively reverse that process: read and uncompress features.
% SYNTAX
%   model_lowbw_compression ('compress', transmit_file, si_orig, hv_feat_orig, ...
%       luma_orig, cb_orig, cr_orig, ati_orig );
%   [si_orig, part_si_min, part_si_max, ...
%         hv_feat_orig, part_hv_min, part_hv_max, y_orig, ...
%         cb_orig, cr_orig, part_c_min, part_c_max, part_c, ...
%         ati_orig, part_ati_min, part_ati_max, part_ati, code_ati ] = ...
%       model_lowbw_compression('uncompress', transmit_file);
% DESCRIPTION
%   When the first argument is 'compress', this function compresses the
%   features associated with the lowbw model or fastlowbw model, and write
%   them to a file for transmission.  Operates on 255 seconds or less.  Longer
%   sequencesmust be split and written to two different files.  To
%   understand the input arguments, see function model_lowbw_features or
%   model_fastlowbw_features.
%
```

```
%   When the first argument is 'uncompress', this function reverses the
%   process: reads compressed original features from file 'transmit_file'
%   and returns those values on the command line.  This function also
%   returns information about the feature compression, which will be needed
%   by function model_lowbw_parameters.m and model_fastlowbw_parameters.m
%   To properly understand those parameters (i.e., everything returned that
%   isn't mentioned in the 'compress' argument list), examine the code
%   below.

if strcmpi(mode,'compress'),
    [row_si,col_si,time_si] = size(si_orig);
    [row_cb,col_cb,time_cb] = size(cb_orig);
    [si_orig]      = model_lowbw_compression_internal (si_orig,      'si', 'compress');
    [hv_feat_orig] = model_lowbw_compression_internal (hv_feat_orig, 'hv', 'compress');
    [luma_orig]    = model_lowbw_compression_internal (luma_orig,    'y', 'compress');
    [cb_orig]      = model_lowbw_compression_internal (cb_orig,      'cb', 'compress');
    [cr_orig]      = model_lowbw_compression_internal (cr_orig,      'cr', 'compress');
    [ati_orig]     = model_lowbw_compression_internal (ati_orig,     'ati', 'compress');

    if time_si > 255 | time_cb > 255,
        error('Time sequence too long.  Split into shorter segments');
    end

    % save data to file
    if exist(transmit_file,'file'),
        delete(transmit_file);
    end
    fid = fopen(transmit_file, 'w');

    % write overhead for SI & HV & Luma
    row_si = uint8(row_si); col_si = uint8(col_si); time_si= uint8(time_si);
    fwrite(fid,row_si,'uint8');
    fwrite(fid,col_si,'uint8');
    fwrite(fid,time_si,'uint8');

    % write out SI & HV & Luma
    fwrite(fid,si_orig, 'ubit9');
    fwrite(fid,hv_feat_orig,'ubit9');
```

```
    fwrite(fid,luma_orig, 'ubit8');

    % write Cb & Cr
    fwrite(fid,cb_orig, 'ubit9');
    fwrite(fid,cr_orig, 'ubit9');

    % write overhead for ATI
    frames = length(ati_orig);
    fwrite(fid,frames,'ushort');

    % write ATI
    fwrite(fid,ati_orig, 'ubit10');

%     fprintf('Total bytes written:  %d  Sequence Size:  (%d,%d,%d) ==> %f kbits/s\n', ftell(fid), ...
%         row_si,col_si,time_si, (ftell(fid) * 8.0) / (double(time_si) * 1000.0));

    fclose(fid);
elseif strcmpi(mode,'uncompress'),
    % load original features
    if ~exist(transmit_file,'file'),
        error(sprintf('Cannot open file ''%s'', file does not exist', transmit_file));
    end
    fid = fopen(transmit_file, 'r');

    % read overhead for SI & HV & Luma
    row = fread(fid,1, 'uint8');
    col = fread(fid,1,'uint8');
    time = fread(fid,1,'uint8');

    % read Si & HV & Luma
    si_orig = fread(fid,row*col*time, 'ubit9');
    hv_feat_orig = fread(fid,row*col*time,'ubit9');
    y_orig = fread(fid,row*col*time, 'ubit8');

    % decompress original features
    [si_orig, part_si_min, part_si_max]      = ...
            model_lowbw_compression_internal (si_orig, 'si', 'decompress', row,col,time);
    [hv_feat_orig, part_hv_min, part_hv_max] = ...
```

```
            model_lowbw_compression_internal (hv_feat_orig, 'hv', 'decompress',row,col,time);
    [y_orig]    = model_lowbw_compression_internal (y_orig, 'y', 'decompress',row,col,time);

    % read Cb & Cr
    cb_orig = fread(fid,row*col*time, 'ubit9');
    cr_orig = fread(fid,row*col*time, 'ubit9');

    % decompress original features
    [cb_orig]    = model_lowbw_compression_internal (cb_orig, 'cb', 'decompress',row,col,time);
    [cr_orig, part_c_min, part_c_max, part_c]    = ...
            model_lowbw_compression_internal (cr_orig, 'cr', 'decompress',row,col,time);

    % read overhead for ATI & read ATI
    frames = fread(fid,1,'ushort');
    ati_orig = fread(fid,frames, 'ubit10');

    % decompress original features
    [ati_orig, part_ati_min, part_ati_max, part_ati, code_ati] = model_lowbw_compression_internal (ati_orig, 'ati',
'decompress',1,1,frames);

    fclose(fid);
else
    error('mode not recognized');
end
```

```
function [data, part_min, part_max, part, code] = ...
    model_lowbw_compression_internal (feature, type, flag, row, col, time);
% model_lowbw_compression_internal
%   Compress or decompress the lowbw/sec model features.
% SYNTAX
%   [data] = model_lowbw_compression_internal (feature, type, 'compress');
%   [data, part_min, part_max, part, code] = ...
%       model_lowbw_compression_internal (feature, type, 'decompress', row, col, time);
% DESCRIPTION
%   Compress or decompress the features in matrix 'feature'.  Variable
```

```
%   'type' controls the compression (e.g., 'hv', 'si','y','cb', 'cr', or 'ati).
%   'flag' should be set to either 'compress' or 'decompress' or 'none'.
%       ('none' is no compression; return data as-is for saving.)
%
%   Return the compressed or decompressed data in [data].  On
%   decompression, also return the minimum & maximum partition boundaries
%   in 'part_min' and 'part_max'.  On compression, [data] will be a one
%   dimensional array.  On decompression, [data] will be of size
%   (row,col,time).
%
%   return value 'part' is the partitions; 'code' is the code values

% get quantizer codebook
if strcmpi(type,'hv'),
    [part,code] = model_lowbw_compression_internal_hv;
elseif strcmpi(type,'si'),
    [part,code] = model_lowbw_compression_internal_si;
elseif strcmpi(type,'y'),
    [part,code] = model_lowbw_compression_internal_y;
elseif strcmpi(type,'cb') | strcmp(type,'cr'),
    [part,code] = model_lowbw_compression_internal_cb_cr;
elseif strcmpi(type,'ati'),
    [part,code] = model_lowbw_compression_internal_ati;
end

if strcmpi(type,'none'),
    data = feature;
    return;
elseif strcmpi(flag,'compress'),

    %  convert from 3D to 1D
    [frows,fcols,ftime] = size(feature);
    feature = reshape(feature,1,frows*fcols*ftime);  % row vector for quantizer

    %  Quantize feature
    [index] = quantiz_fast(feature,part);
    data = index;
    return;
```

```
elseif strcmpi(flag,'decompress'),
    part_min = part(1);
    part_max = part( length(part) );

    % convert back from indicies to quantized values
    feature = code(feature+1);

    data = reshape(feature,row,col,time);
    return;
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [part_hv,code_hv] = model_lowbw_compression_internal_hv;

    %  hv_feat_orig quantizer, 9 bit design
    %  Use a quantizer whose distance between bins (dx) increases as a constant
    %  (err) times the code value.  Only preserve this relationship until the
    %  code value decreases to some low value, then uniformly quantize the
    %  remainder.  Start at 1.0 so the code is exact there.
    %  First bin will be values less than part_hv(1) and last bin will be
    %  values greater than part_hv(code_hv_size-1).  hv_loss and hv_gain parameter values
    %  whose hv_feat_orig values fall outside of the quantizer range will be
    %  zeroed.
    start = 1.0;
    err = 0.00709;  % fraction of error in the quantized value
    high_codes = 228;  % number of nonlinear codes >= 1.0
    low_codes = 202;  % number of nonlinear codes < 1.0
    vlow_codes = 82;  % number of uniform codes for very low values
    code_hv = [start];
    %  Generate high codes
    for i = 1:high_codes-1
        code_hv = [code_hv code_hv(i)*(1+err)];
    end
    %  Generate low codes
    for i = 1:low_codes
```

```
        code_hv = [code_hv(1)*(1-err) code_hv];
    end
    %  Generate very low codes, uniformly distributed to lowest_code
    lowest_code = 0.0991;
    temp = code_hv(1):(lowest_code-1*code_hv(1))/vlow_codes:lowest_code;
    code_hv = [fliplr(temp(2:vlow_codes+1)) code_hv];
    %  Generate the partitions, halfway between codes
    code_hv_size = size(code_hv,2);
    part_hv = (code_hv(2:code_hv_size)+code_hv(1:code_hv_size-1))/2;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [part_si,code_si] = model_lowbw_compression_internal_si;

    %  si_orig quantizer, 9 bit design
    %  Use a quantizer whose distance between bins (dx) increases as a constant
    %  (err) times the code value.
    %  First bin will be values less than part_si(1) and last bin will be
    %  values greater than part_si(code_si_size-1).  si_loss and si_gain parameter values
    %  whose si_orig values fall outside of the quantizer range will be
    %  zeroed.
    start = 2.99;
    err = 0.00728;  % fraction of error in the quantized value
    high_codes = 512;  % number of nonlinear codes
    code_si = [start];
    %  Generate high codes
    for i = 1:high_codes-1
        code_si = [code_si code_si(i)*(1+err)];
    end
    %  Generate the partitions, halfway between codes
    code_si_size = size(code_si,2);
    part_si = (code_si(2:code_si_size)+code_si(1:code_si_size-1))/2;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [part_y, code_y] = model_lowbw_compression_internal_y;
```

```
    %  For y quantizer, just use uniform 8 bit quantizer where the value
    %  represents the y level (0 to 255).  Only one byte per
    %  macroblock(3,3,2).
    start = 0.0;
    err = 1.00;  % distance between codes
    high_codes = 256;  % number of codes
    code_y = [start];
    %  Generate high codes
    for i = 1:high_codes-1
        code_y = [code_y code_y(i)+err];
    end
    %  Generate the partitions, halfway between codes
    code_y_size = size(code_y,2);
    part_y = (code_y(2:code_y_size)+code_y(1:code_y_size-1))/2;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [part_c, code_c] = model_lowbw_compression_internal_cb_cr;

    %  Non-linear quantizer for cb_orig and cr_orig, similar to hv but bipolar (two sided),
    %  since the cb and cr features are bipolar.
    start = 1.0;
    err = 0.0216;  % fraction of error in the quantized value
    high_codes = 217;  % number of nonlinear codes >= start
    low_codes = 40;  % number of uniform codes for very low values (< start)
    code_c = [start];
    %  Generate high codes
    for i = 1:high_codes-1
        code_c = [code_c code_c(i)*(1+err)];
    end
    %  Generate low codes, uniformly distributed from lowest_code to start
    lowest_code = 0.136;
    temp = code_c(1):(lowest_code-1*code_c(1))/low_codes:lowest_code;
    code_c = [fliplr(temp(2:low_codes+1)) code_c];

    %  zero first code and generate sym neg codes
    code_c(1) = 0.0;
```

```
    code_c = [-1.0*fliplr(code_c(2:(high_codes+low_codes-1))) code_c];

    %  Generate the partitions, halfway between codes
    code_c_size = size(code_c,2);
    part_c = (code_c(2:code_c_size)+code_c(1:code_c_size-1))/2;

    %  Correct partition around zero
    low_part_spacing = (start-lowest_code)/low_codes;
    part_c(code_c_size/2) = part_c(code_c_size/2 + 1) - low_part_spacing;
    part_c(code_c_size/2 - 1) = part_c(code_c_size/2 - 2) + low_part_spacing;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [part_ati, code_ati] = model_lowbw_compression_internal_ati;
%  ati_orig quantizer, 10 bit design, uniform quantizer design
%  First bin will be values less than part_ati(1) and last bin will be
%  values greater than part_ati(code_ati_size-1).
    start = 0.0;  % first code
    last = 220.0;  % 210 is the maximum observed in the training data
    high_codes = 1024;  % number of codes for 10-bit quantizer
    code_ati = start:(last-start)/(high_codes-1):last;

    %  Generate the partitions, halfway between codes
    code_ati_size = size(code_ati,2);
    part_ati = (code_ati(2:code_ati_size)+code_ati(1:code_ati_size-1))/2;
```

5.34 Function “model_lowbw_sroi.m”

```
function [valid, cvr, sroi] = model_lowbw_sroi(extra, top, left, bottom, right);
% MODEL_LOWBW_SROI
%   Check image-size validity for lowbw or fastlowbw model; and return spatial region
%   of interest (SROI)
% SYNTAX
%   [valid, cvr, sroi] = model_lowbw_sroi(extra, top, left, bottom, right);
% DESCRIPTION
%   'Extra' is the number of pixels needed on all sides for filtering.
```

```
%   ONE EXTRA pixel will be needed for shifting.
%   (top, left, bottom, right) are the  valid region coordinates.
%   Within above coordinates, find where the low bandwidth model should be run.
%
%   Return whether the model can validly be used (valid == 1) or not
%   (valid == 0); the common valid region (cvr) including just
%   the extra +1 pixel border; and the spatial region of interest without
%   the extra pixels & lines (sroi).   Return variables 'cvr' and 'sroi'
%   are structures, whose elements are top, left, bottom, and right.
%
%   Presume a block-size of 30x30, macro-block size of 3x3.
%   One extra pixel required in all directions, for spatial shift search.

% extra pixel for shifting +- 1 in all directions.
extra = extra + 1;

if nargin ~= 5,
    error('number of input arguments to model_lowbw_sroi invalid.');
end

% if left or top are even, add one
top = top + (1 - mod(top,2));
left = left + (1 - mod(left,2));

% if bottom or right are odd, subtract one
right = right - mod(right,2);
bottom = bottom - mod(bottom,2);

% compute number of rows available, after discard border
num_rows = 30 * floor( ((bottom - top + 1) - extra*2)/30 );
extra_rows = floor(((bottom - top + 1) - num_rows)/2);

num_cols = 30 * floor( ((right - left + 1) - extra*2)/30 );
extra_cols = floor(((right - left + 1) - num_cols)/2);

% figure if valid
if num_rows / 30 < 3 | num_cols / 30 < 3,
    sroi = [];
```

```
    cvr = [];
    valid = 0;
    return;
end

% figure out SROI & CVR
sroi.top = top + extra_rows;
sroi.left = left + extra_cols;
sroi.bottom = sroi.top + num_rows - 1;
sroi.right = sroi.left + num_cols - 1;

valid = 1;

cvr.top = sroi.top-extra;
cvr.left = sroi.left-extra;
cvr.bottom = sroi.bottom+extra;
cvr.right = sroi.right+extra;
```

5.35 Function “quantiz_fast.m”

```
function y = quantiz_fast(x,t)
% QUANTIZ_FAST
%   This function quantizes data.  It is used as an alternative to the
%   MATLAB routine, becase at the time of writing it ran significantly
%   faster.
% SYNTAX
%   y = quantiz_fast(x,t)
% DESCRIPTION
%   'x' is the data to be quantized into discreet bins.
%   't' is the array of quantization thresholds.
%   Return varaible 'y' contains the quantized data.
%
%  This code performs n=2^b level quantization of the signal samples in the
%  row vector x to produce y.  y will have the same size as x.  t is a
%  length n-1 row or column vector of  quantization thresholds.
%  This code works for all b=1,2,3,...
%  Quantization rules are:
```

```
%           x <= t(1)   gives y = 0
%    t(1) < x <= t(2)   gives y = 1
%                ...
%    t(i) < x <= t(i+1) gives y = i
%              ...
%    t(n-1) < x         gives y = n-1


n = length(x);
y = zeros(1,n);           %output variable starts with all zeros
b = log2(length(t)+1);    %number of bits
for i = 1:b               %loop over bits, each pass determines one bit
    temp = find(t(y+(2^(b-i))) < x);
    %compare each sample of x with corresponding current threshold
    y(temp) = y(temp)+(2^(b-i));
    %where x is above current threshold, update y to refect this
    %this update is equivalent to setting the current bit under
    %consideration (b-i) to 1.
end
```

5.36 Function “read_bigyuv.m”

```
function [y,cb,cr] = read_bigyuv(file_name, varargin);
% READ_BIGYUV
%   Read images from bigyuv-file.
% SYNTAX
%   [y] = read_bigyuv(file_name);
%   [y,cb,cr] = read_bigyuv(...);
%   [...] = read_bigyuv(...,'PropertyName',PropertyValue,...);
% DESCRIPTION
%   Read in images from bigyuv file named 'file_name'.
%
%   The luminance plane is returned in 'Y'; the color planes are
%   returned in 'cb' and 'cr' upon request.  The Cb and Cr color planes
%   will be upsampled by 2 horizontally.
%
%   The following optional properties may be requested:
```

```
%
%   'sroi',top,left,bottom,right,
%                       Spatial region of intrest to be returned.  By default,
%                       the entirity of each image is returned.
%                       Inclusive coordinates (top,left),(bottom,right) start
%                       numbering with row/line number 1.
%   'size',row,col,    Size of images (row,col).  By default, row=486,
%                           col=720.
%   'frames',start,stop,    Specify the first and last frames, inclusive,
%                                   to be read ('start' and 'stop').  By
%                                   default, the first frame is read.
%   '128'        Subtract 128 from all Cb and Cr values.  By default, Cb
%                       and Cr values are left in the [0..255] range.
%   'interp'           Interpolate Cb and Cr values.  By default, color
%                       planes are pixel replicated.  Note:  Interpolation is slow.
%
%   Color image pixels will be pixel replicated, so that Cb and Cr images
%   are not subsampled by 2 horizontally.

% read values from clip_struct that can be over written by variable argument
% list.
is_whole_image = 1;
is_sub128 = 0;
is_interp = 0;

num_rows = 486;
num_cols = 720;

start = 1;
stop = 1;

% parse variable arhelp gument list (property values)
cnt = 1;
while cnt <= nargin - 1,
    if ~isstr(varargin{cnt}),
        error('Property value passed into bigyuv_read not recognized');
    end
    if strcmpi((varargin(cnt)),'sroi') == 1,
        sroi.top = varargin{cnt+1};
```

```
        sroi.left = varargin{cnt+2};
        sroi.bottom = varargin{cnt+3};
        sroi.right = varargin{cnt+4};
        is_whole_image = 0;
        cnt = cnt + 5;
    elseif strcmpi((varargin(cnt)),'size') == 1,
        num_rows = varargin{cnt+1};
        num_cols = varargin{cnt+2};
        cnt = cnt + 3;
    elseif strcmpi((varargin(cnt)),'frames') == 1,
        start = varargin{cnt+1};
        stop = varargin{cnt+2};
        cnt = cnt + 3;
    elseif strcmpi((varargin(cnt)),'128') == 1,
        is_sub128 = 1;
        cnt = cnt + 1;
    elseif strcmpi((varargin(cnt)),'interp') == 1,
        is_interp = 1;
        cnt = cnt + 1;
    else
        error('Property value passed into bigyuv_read not recognized');
    end
end

if mod(num_cols,2) ~= 0,
    fprintf('Error: number of columns must be an even number.\n');
    fprintf('This 4:2:2 format sores 4 bytes for each 2 pixels\n');
    error('Invalid specification for argument "num_cols" in read_bigyuv');
end

% Open image file
% [test_struct.path{1} clip_struct.file_name{1}]
[fid, message] = fopen(file_name, 'r');
if fid == -1
    fprintf(message);
    error('bigyuv_read cannot open this clip''s bigyuv file, %s', file_name);
end

% Find last frame.
```

```
fseek(fid,0, 'eof');
total = ftell(fid) / (2 * num_rows * num_cols);
if stop > total,
    error('Requested a frame past the end of the file.  Only %d frames available', total);
end
if stop < 0,
    error('Range of frames invalid');
end
if start > stop | stop < 1,
    error('Range of frames invalid, or no images exist in this bigyuv file');
end

% find range of frames requested.
prev_tslice_frames = start - 1;
tslice_frames = stop - start + 1;
number = start;

% go to requested location
if isnan(start),
    error('first frame of this clip is undefined (NaN).');
end
offset = prev_tslice_frames * num_rows * num_cols * 2; %pixels each image
status = fseek(fid, offset, 'bof');

if status == -1,
    fclose(fid);
    error('bigyuv_read cannot seek requested image location');
end

% initialize memory to hold return images.
y = zeros(num_rows,num_cols,tslice_frames, 'single');

if (nargout == 3),
    cb = y;
    cr = y;
end

% loop through & read in the time-slice of images
```

```
this_try = 1;
for cnt = 1:tslice_frames,
    where = ftell(fid);
    [hold_fread,count] = fread(fid, [2*num_cols,num_rows], 'uint8=>uint8');
    if count ~= 2*num_cols*num_rows,
        % try one more time.
        fprintf('Warning: bigyuv_read could not read entirity of requested image');
        fprintf(' time-slice; re-trying\n');
        %pause(5);
        if where == -1,
            fprintf('Could not determine current location.  Re-try failed.\n');
            error('bigyuv_read could not read entirity of requested image time-slice');
            fclose(fid);
        end
        fseek(fid, where, 'bof');
        [hold_fread,count] = fread(fid, [2*num_cols,num_rows], 'uint8=>uint8');
        if count ~= 2*num_cols*num_rows,
            fclose(fid);
            hold = sprintf('%s%s\n%s', ...
                'time-slice read failed for time-slice in ', file_name, ...
                'bigyuv_read could not read entire requested time-slice');
            error(hold);
        end
    end

    % pick off the Y plane (luminance)
    temp = reshape(hold_fread', num_rows, 2, num_cols);
    uncalib = squeeze(temp(:,2,:));
    y(:,:,cnt) = single(uncalib);

    % If color image planes are requested, pick those off and perform
    % pixel replication to upsample horizontally by 2.
    if nargout == 3,
        temp = reshape(hold_fread,4,num_rows*num_cols/2);

        color = reshape(temp(1,:),num_cols/2,num_rows)';
        color2 = [color ; color];
        uncalib = reshape(color2,num_rows,num_cols);
        cb(:,:,cnt) = single(uncalib);
```

```
        if is_sub128,
            cb(:,:,cnt) = cb(:,:,cnt) - 128;
        end

        color = reshape(temp(3,:),num_cols/2,num_rows)';
        color2 = [color ; color];
        uncalib = reshape(color2,num_rows,num_cols);
        cr(:,:,cnt) = single(uncalib);
        if is_sub128,
            cr(:,:,cnt) = cr(:,:,cnt) - 128;
        end

        % Interpolate, if requested
        if is_interp == 1,
            for i=2:2:num_cols-2,
                cb(:,i,cnt) = (cb(:,i-1,cnt) + cb(:,i+1,cnt))/2;
                cr(:,i,cnt) = (cr(:,i-1,cnt) + cr(:,i+1,cnt))/2;
            end
        end
    end
end

fclose(fid);

if ~is_whole_image,
    y = y(sroi.top:sroi.bottom, sroi.left:sroi.right, :);
    if nargout == 3,
        cb = cb(sroi.top:sroi.bottom, sroi.left:sroi.right, :);
        cr = cr(sroi.top:sroi.bottom, sroi.left:sroi.right, :);
    end
end
```

5.37 Function “resample_image.m”

```
function [image] = resample_image(image, v, h, varargin);
% RESAMPLE_IMAGE
%  stretch or shrink an image
```

```
% SYNTAX
%  [scaled_image] = resample_image(image, v, h);
%  [...] = resample_image(...'PropertyName',...);
% DESCRIPTION
%  This function applies horizontal scaling factor (h / 1000)
%  and vertical scaling factor (v / 1000).  The returned image,
%  'scaled_image', will be of the same size as the input image.
%
%  The following optional properties may be requested.  Fast, Linear, and
%  Quadratic are mutually exclusive (i.e., only one of these may
%  be selected):
%
%   'Fast'  When this option is selected, the function will use a very fast
%           but significantly less accurate resampling algorithm.  The
%           nearest neighbor pixel value will be used (1-point).  This
%           approach appears to be sufficient for color planes (Cb and Cr)
%           but not luminance (Y).
%
%   'Linear' When this option is selected, the function will use linear
%           interpolation (2-point).
%
%   'Quadratic' When this option is selected, the function will use
%           quadratic interpolation (3-point).  This is the default.
%
%   'Interlace' The image is interlaced, so vertical scaling needs to be
%           performed on each field separately.


do_fast = 0;
do_linear = 0;
do_quadratic = 1;
do_interlace = 0;
is_type = 'quadratic';

cnt = 1;
while cnt <= nargin - 3,
    if strcmpi(varargin(cnt),'fast') == 1,
        do_fast = 1;
        do_linear = 0;
```

```
        do_quadratic = 0;
        is_type = 'fast';
        cnt = cnt + 1;
    elseif strcmpi(varargin(cnt),'linear') == 1,
        do_fast = 0;
        do_linear = 1;
        do_quadratic = 0;
        is_type = 'linear';
        cnt = cnt + 1;
    elseif strcmpi(varargin(cnt),'quadratic') == 1,
        do_fast = 0;
        do_linear = 0;
        do_quadratic = 1;
        is_type = 'quadratic';
        cnt = cnt + 1;
    elseif strcmpi(varargin(cnt),'interlace') == 1,
        do_interlace = 1;
        cnt = cnt + 1;
    else
        error('optional argument not recognized.');

    end
end

% vertical scaling on interlaced images must be done on fields.
% handle this here, by recursing (calling this function on each field).
if do_interlace && v ~= 1000,
    % split into fields
    [image1, image2] = split_into_fields(image);

    % call this routine for each field
    [image1] = resample_image(image1, v, h, is_type);
    [image2] = resample_image(image2, v, h, is_type);

    % join into frames
    image = join_into_frames(image1, image2);
    return;
end
```

```
% split into frames, if needed, and run each separately
if ndims(image) == 3,
    [rows,cols,time] = size(image);
    if time > 1,
        for i=1:time,
            image(:,:,i) = resample_image(image(:,:,i), v, h, is_type);
        end
    end
    return;
elseif ndims(image) > 3,
    error('Function resample_image cannot work on arrays of images with 4 or more dimensions');
end


if do_linear,

    [rows,cols] = size(image);

    if v ~= 1000,
        % scale coordinates by vertical factor
        offset = (1:rows)' * 1000 / v + (rows/2 - (1000/v) * (rows/2));

        % find pixels invalidated by this operation
        invalid = find(offset < 1 | offset > rows);

        % if exceed boundary of image, clip at edge.
        offset = max(min(offset, rows), 1);

        % find closest pixels and distance (alpha)
        offset_before = floor(offset);
        offset_after = ceil(offset);
        alpha = offset - offset_before;

        % change from vectors to matrixes, to apply this to the entire image
        alpha = repmat(alpha, 1, cols);
```

```
        % apply linear scaling
        image = (1 - alpha) .* image(offset_before, :) + alpha .* image(offset_after, :);

        % fill invalid portion with zeros
        image(invalid, :) = 0;
    end

    if h ~= 1000,
        % scale coordinates by vertical factor
        offset = (1:cols) * 1000 / h + (cols/2 - (1000/h) * (cols/2));

        % find pixels invalidated by this operation
        invalid = find(offset < 1 | offset > cols);

        % if exceed boundary of image, clip at edge.
        offset = max(min(offset, cols), 1);

        % find closest pixels and distance (alpha)
        offset_before = floor(offset);
        offset_after = ceil(offset);
        alpha = offset - offset_before;

        % change from vectors to matrixes, to apply this to the entire image
        alpha = repmat(alpha, rows, 1);

        % apply linear scaling
        image = (1 - alpha) .* image(:, offset_before) + alpha .* image(:, offset_after);

        % fill invalid portion with zeros
        image(:, invalid) = 0;
    end


elseif do_quadratic,

    [rows,cols] = size(image);
```

```
if v ~= 1000,
    % scale coordinates by vertical factor
    offset = (1:rows)' * 1000 / v + (rows/2 - (1000/v) * (rows/2));

    % find pixels invalidated by this operation
    invalid = find(offset < 1 | offset > rows);

    % find closest pixels and distance (alpha)
    y1 = round(offset);
    y0 = y1 - 1;
    y2 = y1 + 1;

    % find distance
    d = offset - y0;

    % change d from vector to matrix, to apply this to the entire image
    d = repmat(d, 1, cols);
    d2 = d.^2;

    % if exceed boundary of image, clip at edge (i.e., take last pixel value).
    y0 = max(min(y0, rows), 1);
    y1 = max(min(y1, rows), 1);
    y2 = max(min(y2, rows), 1);

    % apply to image
    image_y0 = image(y0,:);
    image_y1 = image(y1,:);
    image_y2 = image(y2,:) ./ 2;

    % calculate weights a, b & c
    % where c = image_y0;
    % where a = y(2) / 2 - y(1) + y(0) / 2
    % where b = -y(2) / 2 + 2 * y(1) - 3/2 * y(0)
    a = image_y2 - image_y1 + image_y0 ./ 2;
    b = -image_y2 + 2 .* image_y1 - (3/2) .* image_y0;

    % apply quadratic scaling: a .* d2 + b .* d + c
    image = a .*d2 + b .* d + image_y0;
```

```
    % fill invalid portion with zeros
    image(invalid, :) = 0;
end

if h ~= 1000,
    % scale coordinates by vertical factor
    offset = (1:cols)' * 1000 / h + (cols/2 - (1000/h) * (cols/2));

    % find pixels invalidated by this operation
    invalid = find(offset < 1 | offset > cols);

    % find closest pixels and distance (alpha)
    y1 = round(offset);
    y0 = y1 - 1;
    y2 = y1 + 1;

    % find distance
    d = offset - y0;

    % change d from vector to matrix, to apply this to the entire image
    d = repmat(d', rows,1);
    d2 = d.^2;

    % if exceed boundary of image, clip at edge (i.e., take last pixel value).
    y0 = max(min(y0, cols), 1);
    y1 = max(min(y1, cols), 1);
    y2 = max(min(y2, cols), 1);

    % apply to image
    image_y0 = image(:,y0);
    image_y1 = image(:,y1);
    image_y2 = image(:,y2) ./ 2;

    % calculate weights a, b & c
    % where c = image_y0;
    % where a = y(2) / 2 - y(1) + y(0) / 2
    % where b = -y(2) / 2 + 2 * y(1) - 3/2 * y(0)
```

```
        a = image_y2 - image_y1 + image_y0 ./ 2;
        b = -image_y2 + 2 .* image_y1 - (3/2) .* image_y0;

        % apply quadratic scaling: a .* d2 + b .* d + c
        image = a .*d2 + b .* d + image_y0;

        % fill invalid portion with zeros
        image(:,invalid) = 0;
    end


elseif do_fast,

    [rows,cols] = size(image);

    if v ~= 1000,
        % scale coordinates by vertical factor
        offset = (1:rows)' * 1000 / v + (rows/2 - (1000/v) * (rows/2));

        % find pixels invalidated by this operation
        invalid = find(offset < 1 | offset > rows);

        % if exceed boundary of image, clip at edge.
        offset = max(min(offset, rows), 1);

        % find closest pixels
        offset = round(offset);

        % apply linear scaling
        image = image(offset, :);

        % fill invalid portion with zeros
        image(invalid, :) = 0;
    end

    if h ~= 1000,
        % scale coordinates by vertical factor
```

```
        offset = (1:cols) * 1000 / h + (cols/2 - (1000/h) * (cols/2));

        % find pixels invalidated by this operation
        invalid = find(offset < 1 | offset > cols);

        % if exceed boundary of image, clip at edge.
        offset = max(min(offset, cols), 1);

        % find closest pixels
        offset = round(offset);

        % apply linear scaling
        image = image(:, offset);

        % fill invalid portion with zeros
        image(:, invalid) = 0;
    end


else
    error('argument not recognized. "resample" no longer available');
end
```

5.38 Function “running_collapse.m”

```
function [data] = running_collapse (request, raw_data, delta, property);
% RUNNING_COLLAPSE
%   Perform a "running" spatial-temporal collapse over a long sequence.
% SYNTAX
%   [data] = running_collapse (request, raw_data, delta, property);
% DESCRIPTION
%   Take a request & raw_data, as defined by function st_collapse.
%   'property' is one of st_collapse's optional properties.  Instead of
%   calling st_collapse directly, divide it into shorter arrays, where the
%   last non-unit dimension is of length 'delta'.  Return the "running"
%   collapse in 'data'.  The length of data will be identical to the last
```

```
%   dimension of 'raw_data'.  The leading values (1 to time-1) will use
%   shorter time lengths.
%
%   'raw_data' must be either 1D or 3D.  See also function st_collapse.m

[row,col,time] = size(raw_data);
if time == 1,
    % 1D
    if col ~= 1,
        error('raw_data cannot be 2D');
    end
    time = row;
    data = zeros(time,1);
    for cnt = 1:time,
        start = max(1, cnt - delta + 1);
        stop = cnt;
        data(cnt) = st_collapse(request, raw_data(start:stop), property);
    end
else
    % 3D
    data = zeros(time,1);
    for cnt = 1:time,
        start = max(1, cnt - delta + 1);
        stop = cnt;
        data(cnt) = st_collapse(request, raw_data(:,:,start:stop), property);
    end
end
```

5.39 Function “split_into_fields.m”

```
function [one, two] = split_into_fields(y);
% SPLIT_INTO_FIELDS
%  Splits one frame into two fields.
% SYNTAX
%  [one two] = split_into_fields(y);
% DESCRIPTION
%  Split frame 'y' into field one ('one') and field two ('two').  Field two
%  contains the top line of the image; field one contains the second line
```

```
%  of the image.  For NTSC, field one occurs before field two in time.  For
%  PAL, the reverse is the case.  Y can be a time-slice of frames.
%
%  If image 'y' contains an odd number of rows, then the last (bottom) row
%  will be eliminated.  See also function 'join_into_frames'

[row, col, time] = size(y);
if mod(row,2),
    y = y(1:row-1,1:col,1:time);
    [row, col, time] = size(y);
end
y = reshape(y,2,row/2,col,time);
two = squeeze(y(1,:,:,:));
one = squeeze(y(2,:,:,:));
```

5.40 Function “st_collapse.m”

```
function [data] = st_collapse(request, raw_data, varargin)
% ST_COLLAPSE
%  Compute the requested spatial or temporal collapsing function.
% SYNTAX
%  [data] = st_collapse(request, raw_data)
%  [data] = st_collapse(...,'PropertyName',...);
% DESCRIPTION
%  Compute the requested spatial or temporal collapsing function on the
%  FIRST dimension of the array or matrix, 'raw_data', and return the
%  results in 'data'.  The available precentile functions are:
%   'mean', 'std', 'rms', 'min', 'max', '10%', '25%', '50%', '90%',
%   'above90%', 'above95%', 'above99%', 'above90%tail', 'above95%tail', 'above98%tail', 'above99%tail',
%   'below5%', 'below10%', 'below1%', 'below1%tail, 'below2%tail', 'below5%tail', 'below10%tail'
%   'below25%', 'above75%', 'below2%', 'above98%', 'below50%tail'
%   'between25%50%'
%           [ The meanings of the above are as defined in "Video Quality
%             Measurement Techniques" NTIA Technical Report 02-392. ]
%  and 'minkowski(P,R)'.
%           [ minkowski = mean(abs(raw_data).^P).^(1/R) ]
%           Where 'P' and 'R' are replaced with the actual values to be
%           used.  For example, 'minkowski(1.8,2.8)' or 'minkowski(6,7)'
```

```
%
% The following optional parameters are also available.  All of these
% options are mutually exclusive.
%
%   'MacroBlock', row, col, time,
%           Apply the requested function to macro blocks, of size (row,col,time).
%           If the region does not evenly divide, center spatially, and
%           abut with the end of the raw data temporally.
%   'OverlapMacroBlock', row, col, time,
%           Apply the requested function to macro blocks, of size (row,col,time).
%           Unlike option 'MacroBlock', blocks overlap rather than
%           abutting.  Thus, returned data will be smaller only by
%           (row-1) rows,(col-1) columns and (time-1) in time.
%   'SlideMacroBlock', row, col, time,
%           Apply the requested function to macro blocks, of size (row,col,time).
%           Blocks will overlap in time and abut spatially.
%   '3D',   Apply the requested function simultaneously to all
%           dimensions.  Thus, convert all of the data into a 1D array and
%           apply the colllapsing function to that 1D array.
%   '1D',   Apply the requested function to only the first
%           dimensions.  For example, the variable 'raw_data' should be 2D
%           (spatial,temporal) or 1D (temporal).  This is the default
%           behavior.
%
%
% WARNING: Unless 'MacroBlock' or '3D' arguments are selected, the
% requested function will be applied to the first dimension only!
%

collapse_3d = 0;
collapse_macroblock = 0;
collapse_overlap_macroblock = 0;
collapse_slide_macroblock = 0;
mb_row = 1;
mb_col = 1;
mb_time = 1;

cnt = 1;
while cnt <= nargin-2,
```

```
    switch lower(varargin{cnt}),
        case { 'macroblock' },
            collapse_macroblock = 1;
            mb_row = varargin{cnt+1};
            mb_col = varargin{cnt+2};
            mb_time = varargin{cnt+3};
            cnt = cnt + 4;
        case { 'overlapmacroblock' },
            collapse_overlap_macroblock = 1;
            mb_row = varargin{cnt+1};
            mb_col = varargin{cnt+2};
            mb_time = varargin{cnt+3};
            cnt = cnt + 4;
        case { 'slidemacroblock' },
            collapse_slide_macroblock = 1;
            mb_row = varargin{cnt+1};
            mb_col = varargin{cnt+2};
            mb_time = varargin{cnt+3};
            cnt = cnt + 4;
        case { '3d' },
            collapse_3d = 1;
            cnt = cnt + 1;
        case { '1d' },
            collapse_3d = 0;
            cnt = cnt + 1;
        otherwise
            error('Function st_collapse, optional argument "%s" not recognized', varargin{cnt});
    end

    if cnt <= nargin-2,
        error('Optional arguments are mutually exclusive.  Choose one only.');
    end
end

% Overlapping Macroblock Case.  Recurse to calculate.
if collapse_overlap_macroblock,
    [row_raw,col_raw,time_raw] = size(raw_data);
    data = zeros(row_raw-mb_row+1, col_raw-mb_col+1, time_raw-mb_time+1);
    [row,col,time] = size(data);
```

```
    for cnt1 = 1:mb_row,
        temp = floor((row_raw - cnt1 + 1) / mb_row) * mb_row;
        rng1 = cnt1:(cnt1-1+temp);
        if length(rng1) < mb_row,
            continue;
        end
        for cnt2 = 1:mb_col,
            temp = floor((col_raw - cnt2 + 1) / mb_col) * mb_col;
            rng2 = cnt2:(cnt2-1+temp);
            if length(rng2) < mb_col,
                continue;
            end
            for cnt3 = 1:mb_time,
                temp = floor((time_raw - cnt3 + 1) / mb_time) * mb_time;
                rng3 = cnt3:(cnt3-1+temp);
                if length(rng3) < mb_time,
                    continue;
                end
                data(cnt1:mb_row:row, cnt2:mb_col:col, cnt3:mb_time:time) = ...
                    st_collapse(request, raw_data(rng1,rng2,rng3), 'macroblock', ...
                    mb_row,mb_col,mb_time);
            end
        end
    end
    return;
end

% Slide Macroblock Case.  Recurse to calculate.
if collapse_slide_macroblock,
    [row_raw,col_raw,time_raw] = size(raw_data);
    row = floor(row_raw / mb_row);
    col = floor(col_raw / mb_col);
    time = time_raw - mb_time + 1;
    data = zeros(row,col,time);

    for cnt = 1:mb_time,
        temp = floor((time_raw - cnt + 1) / mb_time) * mb_time;
        rng = cnt:(temp+cnt-1);
```

```
        if length(rng) < mb_time,
            continue;
        end
        data(:,:, cnt:mb_time:time) = ...
            st_collapse(request, raw_data(:,:,rng), 'macroblock', ...
            mb_row,mb_col,mb_time);
    end
    return;
end

% if wanting to collapse a 3D structure all at once, reshape into an array.
if collapse_3d,
    [r,c,t] = size(raw_data);
    raw_data = reshape(raw_data, r*c*t,1);
end

% reshape for macroblocks.
if collapse_macroblock,
    [r,c,t] = size(raw_data);

    % error check.
    if floor(r / mb_row) == 0 | floor(c / mb_col) == 0 | floor(t / mb_time) == 0,
        error('st_collapse, macroblock too large.');
    elseif mb_row == 1 & mb_col == 1 & mb_time == 1,
        error('st_collapse, macro-block size must be greater than one.');
    end

    % figure out how to center the macroblocks
    [r,c,t] = size(raw_data);
    r_start = floor( mod(r,mb_row) / 2 ) + 1;
    r_stop = r_start + floor(r / mb_row)*mb_row - 1;

    c_start = floor( mod(c,mb_col) / 2 ) + 1;
    c_stop = c_start + floor(c / mb_col)*mb_col - 1;

    t_start = mod(t,mb_time) + 1;
    t_stop = t;
```

```
    % copy over macro-block data.
    raw_data = raw_data(r_start:r_stop,c_start:c_stop,t_start:t_stop);

    % reorganize first dimension
    [r,c,t] = size(raw_data);
    raw_data = reshape(raw_data, mb_row, r / mb_row, c, t);
    raw_data = permute(raw_data, [1 3 4 2]);
    raw_data = reshape(raw_data, mb_row * mb_col, c / mb_col, t, r / mb_row);
    raw_data = permute(raw_data, [1 3 4 2]);
    raw_data = reshape(raw_data, mb_row * mb_col * mb_time, t / mb_time, r / mb_row, c / mb_col);
    raw_data = permute(raw_data, [1 3 4 2]);
end

% error checking.
[r,c,t] = size(raw_data);
if t > 1 & ~collapse_macroblock,
    error('Function st_collapse requires 2D or 1D arrays, only.  See help information warning.');
end

% Apply requested function.
above = 0;
below = 0;
tail = 0;

if r == 1,
    % Special case.  ST-collapse over a singleton dimension.  This is
    % always either the same as the input or zero.
    if strcmp(request,'std') | length(findstr(request,'tail')) > 0,
        data = 0 * raw_data;
    else
        data = raw_data;
    end

% handle the usual cases.
elseif strcmp(request,'mean'),
    data = mean(raw_data);
elseif strcmp(request,'std'),
    data = std(raw_data);
```

```
elseif strcmp(request,'rms'),
    data = sqrt(mean(raw_data.^2));
elseif strcmp(request,'min'),
    data = min(raw_data);
elseif strcmp(request,'max'),
    data = max(raw_data);
elseif strncmp(request, 'minkowski(', 10),
    [mink n] = sscanf(request(11:length(request)), '%f,%f');
    if n ~= 2,
        error('Cannot parse minkowski P and R values in string "%s"', request(10:length(request)));
    end
    data = mean(abs(raw_data).^mink(1)).^(1.0/mink(2));
elseif strcmp(request,'between25%50%'),
    percentile1 = 0.25;
    percentile2 = 0.50;

    % if 1D but wrong direction vector, transpose it.
    if ndims(raw_data) == 2 & r == 1,
        raw_data = raw_data';
    end

    % compute percentile functions
    [rows,cols] = size(raw_data);

    want1 = 1 + round((rows-1) * percentile1);
    want2 = 1 + round((rows-1) * percentile2);

    temp = sort(raw_data, 1);
    data = mean(temp(want1:want2,:,:,:),1);
else
    if strcmp(request,'10%'),
        percentile = 0.10;
    elseif strcmp(request,'25%'),
        percentile = 0.25;
    elseif strcmp(request,'50%'),
        percentile = 0.50;
    elseif strcmp(request,'75%'),
        percentile = 0.75;
    elseif strcmp(request,'90%'),
```

```
    percentile = 0.90;
elseif strcmp(request,'above95%'),
    percentile = 0.95;
    above = 1;
elseif strcmp(request,'below5%'),
    percentile = 0.05;
    below = 1;
elseif strcmp(request,'above99%'),
    percentile = 0.99;
    above = 1;
elseif strcmp(request,'below1%'),
    percentile = 0.01;
    below = 1;
elseif strcmp(request,'above98%'),
    percentile = 0.98;
    above = 1;
elseif strcmp(request,'below2%'),
    percentile = 0.02;
    below = 1;
elseif strcmp(request,'above90%'),
    percentile = 0.90;
    above = 1;
elseif strcmp(request,'below10%'),
    percentile = 0.10;
    below = 1;
elseif strcmp(request,'above75%'),
    percentile = 0.75;
    above = 1;
elseif strcmp(request,'below25%'),
    percentile = 0.25;
    below = 1;
elseif strcmp(request,'above95%tail'),
    percentile = 0.95;
    above = 1;
    tail = 1;
elseif strcmp(request,'below5%tail'),
    percentile = 0.05;
    below = 1;
    tail = 1;
```

```
elseif strcmp(request,'below50%tail'),
    percentile = 0.50;
    below = 1;
    tail = 1;
elseif strcmp(request,'below2%tail'),
    percentile = 0.02;
    below = 1;
    tail = 1;
elseif strcmp(request,'above98%tail'),
    percentile = 0.98;
    above = 1;
    tail = 1;
elseif strcmp(request,'above99%tail'),
    percentile = 0.99;
    above = 1;
    tail = 1;
elseif strcmp(request,'below1%tail'),
    percentile = 0.01;
    below = 1;
    tail = 1;
elseif strcmp(request,'above90%tail'),
    percentile = 0.90;
    above = 1;
    tail = 1;
elseif strcmp(request,'below10%tail'),
    percentile = 0.10;
    below = 1;
    tail = 1;
else
    error('ERROR: percentile function "%s" not recognized by function compute_percentile', request);
end

% if 1D but wrong direction vector, transpose it.
if ndims(raw_data) == 2 & r == 1,
    raw_data = raw_data';
end

% compute percentile functions
[rows,cols] = size(raw_data);
```

```
    want = 1 + round((rows-1) * percentile);
    %fprintf('r=%d, c=%d, percentile %f, want=%d\n', rows, cols, percentile, want);

    temp = sort(raw_data, 1);
    if ~below & ~above & ~tail,
        data = temp(want,:,:,:);
    elseif above & ~tail,
        data = mean(temp(want:rows,:,:,:),1);
    elseif below & ~tail,
        data = mean(temp(1:want,:,:,:),1);
    elseif above & tail,
        if want == rows,
            % special case, can't do tail.
            data = temp(want,:,:,:) * 0;
        else
            data = mean(temp(want:rows,:,:,:),1) - temp(want,:,:,:);
        end
    elseif below & tail,
        if want == 1,
            % special case, can't do tail.
            data = temp(want,:,:,:) * 0;
        else
            data = temp(want,:,:,:) - mean(temp(1:want,:,:,:),1);
        end
    end;
end

% get rid of extra dimension.
[a,b,c,d] = size(data);
data = reshape(data,b,c,d);
```

5.41 Function “tslice_conversion.m”

```
function [tslice_frames, over_sec] = tslice_conversion (tslice_sec, fps)
% TSLICE_CONVERSION
%   Convert from time-slice length in seconds, to time-slice length in
```

```
%   frames at a given frame rate.
% SYNTAX
%  [tslice_frames, over_sec] = tslice_conversion (tslice_sec, fps)
% DESCRIPTION
%
%  [tslice_frames, over_sec] = tslice_conversion (tslice_sec, fps)
%  Given the current frame rate in frames per second ('fps') and the length
%  of the current time-slice in seconds ('tslice_sec'), compute the
%  length of the current time-slice in frames ('tslice_frames').  Also return how
%  much the chosen block length exceeds the requested block length, in
%  seconds ('over_sec').
%
%  NOTE:  if the specified tslice_sec is within (plus or minus) one
%  thousandth of one millisecond of being exactly tslice_frames
% (e.g., over_sec <= 0.000001), then this function will assume the
% user intended that exact number of tslice_frames.
%
%  NOTE:  Any time-slice smaller than one frame will be implemented as
%  one frame per time-slice.

% compute length of time-slice, in frames
tslice_frames = ceil(tslice_sec * fps);

% compute amount of seconds that above measurement exceeds that requested.
over_sec = tslice_frames - tslice_sec * fps;

% if within 1 ms of an integer tslice_frames with over_sec = 0, use that.
if tslice_frames == 1,
    over_sec = 0;
elseif over_sec <= 0.000001,
    over_sec = 0;
elseif over_sec >= 0.999999,
    tslice_frames = tslice_frames - 1;
    over_sec = 0;
end
```