NTIA Technical Memorandum TM-10-463

A Full Reference (FR) Method Using Causality Processing for Estimating Variable Video Delays

Stephen Wolf

technical memorandum

U.S. DEPARTMENT OF COMMERCE • National Telecommunications and Information Administration

NTIA Technical Memorandum TM-10-463

A Full Reference (FR) Method Using Causality Processing for Estimating Variable Video Delays

Stephen Wolf

U.S. DEPARTMENT OF COMMERCE
Gary Locke, Secretary

Lawrence E. Strickling, Assistant Secretary
for Communications and Information

October 2009

DISCLAIMER

CONTENTS

A FULL REFERENCE (FR) METHOD USING CAUSALITY PROCESSING FOR ESTIMATING VARIABLE VIDEO DELAYS

Stephen Wolf [1]

Digital video transmission systems consisting of a video encoder, a digital transmission method (e.g., Internet Protocol – IP), and a video decoder can produce pauses in the video presentation, after which the video may continue with or without skipping video frames. This time varying delay of the output (or processed) video frames can present a challenge for some video quality measurement systems. The reason is that time alignment errors between the output video sequence and the input (or reference) video sequence may produce measurement errors for full reference measurements like Peak Signal to Noise Ratio (PSNR) that greatly exceed the perceptual impact of the time varying video delays. This document presents a Full Reference (FR) method for estimating variable video delays. The algorithm can optionally execute a sophisticated causality processing algorithm to improve the robustness of the delay estimates. The delay estimates produced by this algorithm can be utilized by a FR quality measurement system to remove variable video delay as a calibration step before computing the quality measurement.

Key words: calibration; causality; delay; dropped frames; Full Reference (FR); pausing; skipping; video quality

1. INTRODUCTION

Digital video transmission systems consisting of a video encoder, a digital transmission method (e.g., Internet Protocol – IP), and a video decoder can produce pauses in the video presentation, after which the video may continue with or without skipping video frames. There are several possible reasons for this behavior. One reason is that the video encoder may decide to reduce the video frame transmission rate momentarily in order to save bits. For example, an original video stream with a frame rate of 30 frames per second (fps) may be reduced to 15 fps by dropping every other video frame. Another reason is that the video decoder may decide to freeze the last good video frame when the digital transmission is interrupted or when errors such as IP packet loss are detected. This is a simple error concealment algorithm that is used by many video decoders. After the interruption, output video frames may be displayed without loss (e.g., pausing without skipping) or with some loss (e.g., pausing with skipping). Another option might be to display the output video frames in fast forward motion to make up for the lost time.

[1] The author is with the Institute for Telecommunication Sciences, National Telecommunications and Information Administration, U.S. Department of Commerce, 325 Broadway, Boulder, CO 80305.

Whatever the case, there are many reasons why output video frames from modern video compression systems have time varying video delays.

This time varying delay of the output (or processed) video frames can present a challenge for some video quality measurement systems. The reason is that time alignment errors between the output video sequence and the input (or reference) video sequence may produce measurement errors for full reference measurements like Peak Signal to Noise Ratio (PSNR) that greatly exceed the perceptual impact of the time varying video delays. For example, a one-frame video freeze without skipping will result in either the prior or later output segment being shifted by one video frame with respect to the original reference segment. While this is barely noticeable to viewers, the PSNR measurement will detect a large impairment for the output segment that is off by one video frame with respect to the original.

This document presents a Full Reference (FR) method for estimating variable video delays. The best matching original (or input) video frame is determined for each processed (or output) video frame. For some video sequences, estimating the best matching original video frames is difficult. This is true for still or nearly still video, video with repetitive motion, highly distorted processed video, or processed video that has partial frame updates (e.g., the foreground updates while the background does not update). For these difficult sequences, sophisticated causality processing (i.e., the processed video cannot go backwards in time) that assembles the time aligned sequence using a set of heuristic rules can improve the video delay estimates. This algorithm can also produce a list of rank sorted matching original video frames for each processed video frame (sorted from most likely match to least likely match).

One possible application for the results from this algorithm is to remove variable frame delays from the processed sequence as a calibration step before computing a full reference quality measurement. A second application is to compute a video quality parameter that only measures the perceptual effects of variable frame delays. Combining this video quality parameter with the full reference quality measurement (with variable video delay removed) should produce a more accurate overall estimate of video quality.

2. ALGORITHM DESCRIPTION

The FR algorithm for determining the best matching original video frame[2] for each processed video frame was developed using sequences of captured video frames (often called video clips) that ranged from 8 to 15 seconds in length. The behavior of the algorithm for shorter or longer video sequences should be analyzed before being applied. This section provides a step by step description of the algorithm as applied to one video clip. The algorithm only utilizes the luminance images of the video clip (e.g., the Y channel in an ITU-R Recommendation BT.601 sampled video stream), which will be denoted in this document as $Y(i, j, t)$, where $t = 0, 1, 2, ..., N\text{-}1$ (where N is the total number of frames in the video clip) and i and j are the row and column indices of the individual pixels, respectively. A subscript of "p" on the luminance image will denote a processed video frame while a subscript of "o" will denote an original video frame.

The full algorithm that includes the causality processing can be partitioned into 6 stages. Stage 1 (described in Section 2.1) involves computation of the Mean Squared Errors (MSEs) between each processed frame and the set of original frames that are within a user-specified temporal search window. This computationally intensive step produces the information utilized by the rest of the algorithm (Stages 2 to 6). Stage 2 (described in Section 2.2) performs a threshold procedure on the original MSEs for each processed frame to produce a set of rank-sorted fuzzy time alignments for each processed frame (rank sorted from most likely to least likely). Stage 3 (described in Section 2.3) examines the list of minimum MSE alignments to determine time segments that exhibit "normal causality," where "normal causality" imposes a user-specified limit on the magnitude of the frame-by-frame forward jumps in time (backward jumps in time are not allowed for causal processing). Stage 4 (described in Section 2.4) begins the generation of the causal time alignment for the processed video clip by starting with the longest normal causal segment (from Stage 3) and filling in the shorter normal causal segments. Stage 5 (described in Section 2.5) fills in the remaining holes using either interpolation or the rank-sorted MSEs generated from Stage 2 (which includes an expanded list of likely time alignments for each processed frame in addition to the most likely minimum MSE time alignment). At the end of Stage 5, a best guess of the causal time alignment is obtained. Stage 6 (described in Section 2.6) expands the list of rank-sorted fuzzy alignments generated from Stage 2 to include the points in the causal time alignment found at the end of Stage 5.

2.1. Stage 1: Compute MSEs Between Processed Frames and Original Frames

This stage of the algorithm computes the MSE between each processed video frame and all original video frames that are within plus or minus a temporal uncertainty search window. The processed video clip is assumed to have been spatially aligned to match the original video clip. References [1] and [3] present methods for performing this spatial shift estimation and

[2] For progressive scan video systems, the algorithm presented here uses frames. For interlaced video systems, the algorithm uses fields. Some interlaced video systems might reframe the output video and this complicates the time alignment algorithm considerably. For a definition and explanation of reframing, please see Section 3.1.2 of [1]. For simplicity, this document will generally use the term "frame" or "frames" to describe the algorithm, but processing details that are specific to fields in interlaced systems will be described where relevant.

correction. The processed video clip might also have gain and level offset errors with respect to the original clip. Gain and level offset errors can be accommodated by pre-normalizing the original and processed video clips to have zero mean and unit variance. Here is the procedure to compute the MSEs[3]:

<u>Step 1)</u> Normalize the processed video clip to have zero mean and unit variance.

$$Y_p(i,j,t_p) = \left(Y_p(i,j,t_p) - m_p\right)/\sigma_p\,, \quad (i,j) \in SROI, \quad t_p = 0,1,2,\ldots,N-1. \tag{1}$$

Here *SROI* is the Spatial Region of Interest, which may be selected to be the central portion of the image to eliminate image border pixels that do not contain valid picture (e.g., some cameras may not fill the entire ITU-R Recommendation BT.601 frame, encoders may not transmit the entire frame). The mean (m_p) and standard deviation (σ_p) of the processed video clip are calculated using all the image pixels within the *SROI* and the time duration of the video clip (t_p = 0, 1, 2, ..., *N-1*).

<u>Step 2)</u> Similar to Step 1 above, normalize the original video clip to have zero mean and unit variance.

$$\begin{aligned}&Y_o(i,j,t_o) = \left(Y_o(i,j,t_o) - m_o\right)/\sigma_o\,,\\ &(i,j) \in SROI,\\ &t_o = first\_align, first\_align+1, first\_align+2,\ldots, first\_align+N-1.\end{aligned} \tag{2}$$

Here, *first_align*[4] is the best guess for the original video frame that aligns with the first processed video frame (i.e., t_p = 0 in the equation for Step 1 above). The mean (m_o) and standard deviation (σ_o) of the original video clip are calculated using all the image pixels within the *SROI* and the time duration of the video clip (t_o = *first_align*, *first_align*+1, *first_align*+2, ..., *first_align*+ *N-1*).

<u>Step 3)</u> For each processed frame t_p, compute the MSE between this frame and all original frames within plus or minus the temporal uncertainty (*t_uncert*), which is the maximum expected time alignment offset error between processed frames and original frames (when using the single time alignment point given by *first_align*).

$$\begin{aligned}&MSE(t_o,t_p) = \underset{over\ i,j}{mean}\left\{\left[Y_o(i,j,t_o) - Y_p(i,i,t_p)\right]^2\right\},\\ &t_p = 0,1,2,\ldots,N-1,\\ &t_o = first\_align - t\_uncert + t_p,\ldots, first\_align + t_p,\ldots, first\_align + t\_uncert + t_p\,.\end{aligned} \tag{3}$$

[3] In order to make this algorithm available for researchers, the Appendix provides MATLAB® code that fully implements the ideas presented in this document. The variable names used in this description are the same as those used in the MATLAB code. The MATLAB code for Stage 1 begins on page 33.

[4] There are a number of parameters (e.g., *first_align*) that control the behavior of the algorithm presented herein. The user can specify values for these parameters, or accept the default values given either here or in the MATLAB® code that implements the algorithm (see the Appendix).

This formulation assumes an extra *t_uncert* original frames before frame *first_align* and after frame *first_align*+*N*-1. If some of these frames are not available, then the maximum number of original frames that are available are searched.

Step 4) Rank sort (from low to high) each column of $MSE(t_o, t_p)$. This sorting is performed over the row index t_o. Thus for each value of t_p, all the t_o MSE values are sorted from minimum to maximum.

$$MSE\_Sort(t_{os}, t_p) = \underset{over\ t_o}{sort}\{MSE(t_o, t_p)\}. \tag{4}$$

Keep track of the original frame numbers that correspond to the *MSE_Sort* values in a separate variable *Orig_Index_Sort* (t_{os}, t_p). The first row of *Orig_Index_Sort* (i.e., $t_{os} = 1$) contains the original frame numbers that minimize MSE for the processed frames ($t_p = 0, 1, 2, ..., N\text{-}1$), with their corresponding MSEs given by the first row of *MSE_Sort*.

For interlaced video systems, the above MSE computations are performed on fields rather than frames. If the processed video has the possibility of being reframed with respect to the original video, then extra MSE computations are required where the processed field is shifted vertically by up to one video line with respect to the original field.

Step 5) Define a Boolean row vector *Edge* that is 1 when the best matching original frame (for a given processed frame) has hit the edge of the search range (otherwise *Edge* is set to 0).[5]

$$Edge(t_p) = \begin{cases} 1 & if \quad Orig\_Index\_Sort(1, t_p) = first\_align - t\_uncert + t_p \\ 1 & if \quad Orig\_Index\_Sort(1, t_p) = first\_align + t\_uncert + t_p \\ 0 & otherwise \end{cases} \tag{5}$$

The *Edge* vector will be used in the selection of the causal segments in Stage 3 of the algorithm (Section 2.3). Frame alignments that are “on the edge” of the search range will not be included in a causal segment on the presumption that a better alignment might exist that is outside of the temporal search window. In addition, when the search goes awry, it commonly hits the edge of the search window. Thus, the robustness of the algorithm is improved by not including the edge points.

Figure 1 gives an example plot of the *MSE* function (3) for one processed frame. The original frame with the smallest *MSE* (i.e., closest match to the processed frame) is shown as a red circle on the plot. Figure 1 gives one slice of the three dimensional *MSE* function shown in Figure 2 (at processed frame 50). In Figure 2, higher *MSE* values are hotter colors (with red being the highest) while lower *MSE* values are cooler colors (with blue being the lowest), except for the best matching original frame, which is shown as a red circle like Figure 1.

[5] As mentioned in Step 3, this mathematical formulation assumes an extra *t_uncert* original frames before frame *first_align* and after frame *first_align*+*N*-1. If these extra original frames are not available, and the search hits the beginning or end of the original frames that are available, then an *Edge* (set equal to 1) is declared in these cases.

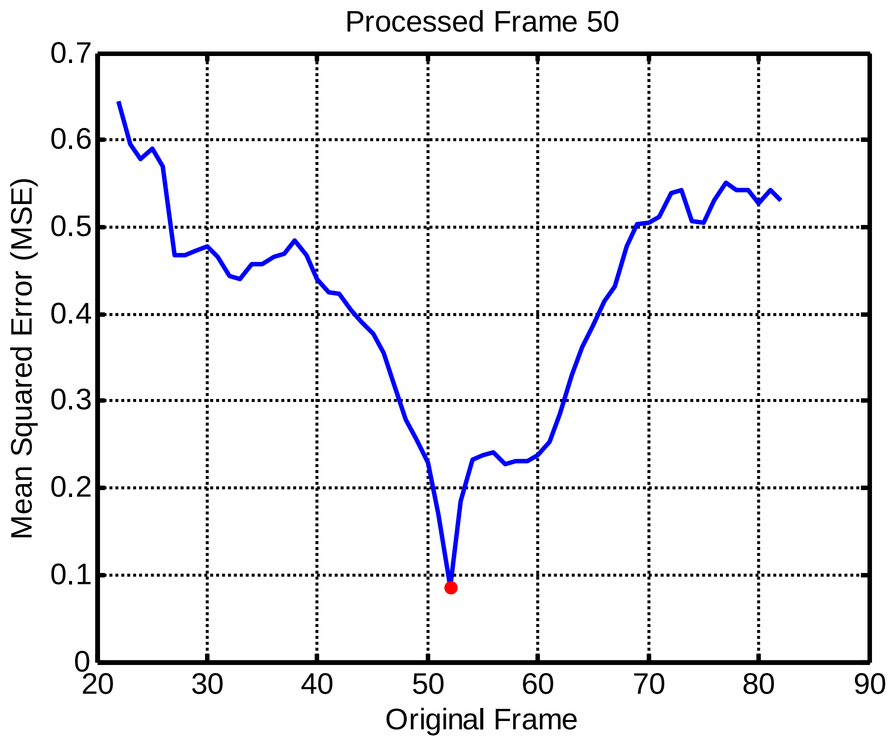

Figure 1. Example *MSE* plot for one processed frame.

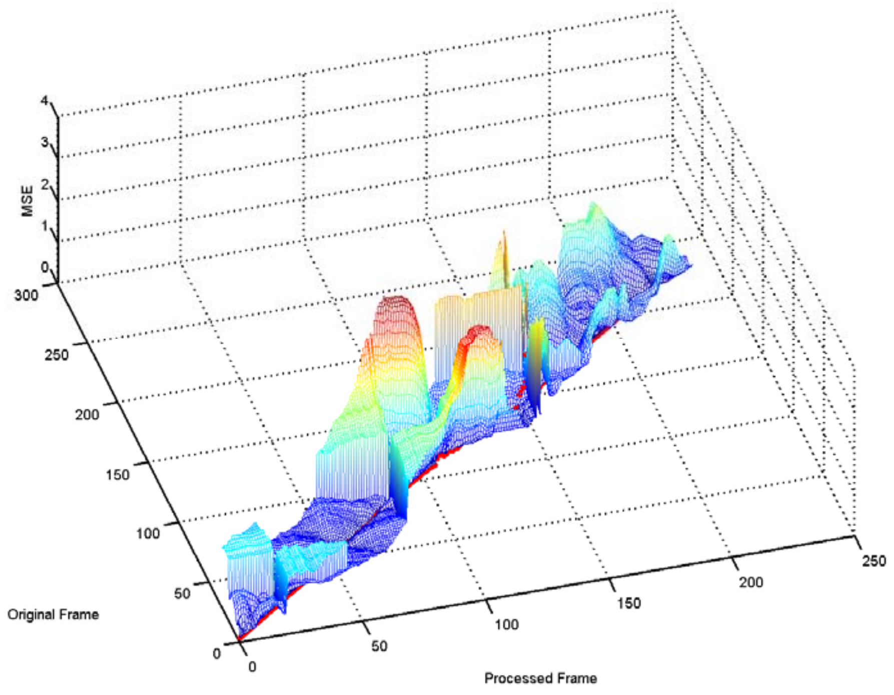

Figure 2. Example three dimensional view of the *MSE* function.

2.2. Stage 2: Compute Initial Set of Rank-Sorted Fuzzy Time Alignments

For a given processed video frame, the *MSE* function shown in Figure 1 will be a minimum at the best matching original video frame. However, this minimum might not be the correct time aligned original video frame. For instance, if the processed video frame is distorted, the wrong original video frame might be chosen. This stage of the algorithm uses a dynamic threshold scheme to select a set of closely matched original video frames for each processed video frame. In Figure 2, the amplitude of the *MSE* waveform for a given processed frame (Figure 1) changes significantly as you move along the processed frame axis. By using a dynamic threshold that is based upon the *MSE* values at each processed frame, one can obtain a fuzzy set of original time alignments that depend upon the shape of the Figure 1 *MSE* function. Shapes with a well defined minimum dip produce a few (or just one) time alignments while shapes with a broad minimum or multiple dips produce more time alignments.

We continue with the step by step numbering used in Section 2.1 to specify the algorithm description (the MATLAB code for Stage 2 begins on page 34).

<u>Step 6)</u> For each processed frame t_p, compute a fuzzy set of matching original frame indices as:

$$\begin{aligned}&Fuzzy(.,t_p) = \{Orig\_Index\_Sort(t_{os},t_p)\} \quad such\ that \\ &MSE\_Sort(t_{os},t_p) \le MSE\_Sort(1,t_p) + 0.005 * MSE\_Sort(floor(0.95*L),t_p).\end{aligned} \qquad (6)$$

Here, L is the total number of t_{os} samples present for each processed frame t_p, and *floor* performs an integer round down. Since L varies for each processed frame t_p, the first index of the array *Fuzzy* is given a '.' in (6). This equation basically sets a threshold on the acceptable original *MSE* values such that all those points less than or equal to the minimum *MSE* plus 0.005 (or 0.5%) of the 95% rank sorted *MSE* value are included in the set of likely time alignments. The 95% rank sorted value is used rather than the maximum *MSE* for greater robustness. Equation (6) provides the desired dynamic thresholding scheme.

<u>Step 7)</u> Create a bigger fuzzy array that expands the set of frame indices found in (6) to include the full range of original frames that were found (from the earliest to the latest in time). This bigger fuzzy array will include additional original frames if the Figure 1 *MSE* function has multiple dips and these dips were captured by (6) but some points in between the dips were not captured. Sort each column of this array according to *MSE* (from lowest to highest) to create a final fuzzy array of original frame indices $Final\_Fuzzy\_Index(t_{os},t_p)$, with a corresponding *MSE* array $Final\_Fuzzy\_MSE(t_{os},t_p)$. After this step, the variable used to access the original frames (t_{os}) can have a different number of elements for each processed frame t_p. The first row of $Final\_Fuzzy\_Index(t_{os},t_p)$, $t_{os} = 1$, gives the original frames that have the minimum *MSE* with respect to the processed frames. Original frames given by higher t_{os} values have higher *MSE*s.

Figure 3 gives the result from Step 7 for processed frame 117 of Figure 2. The original frames shown with a red circle met the dynamic threshold scheme given by (6). The original frame shown with a green square was included in the bigger set from Step 7 since it fell between the red circles.

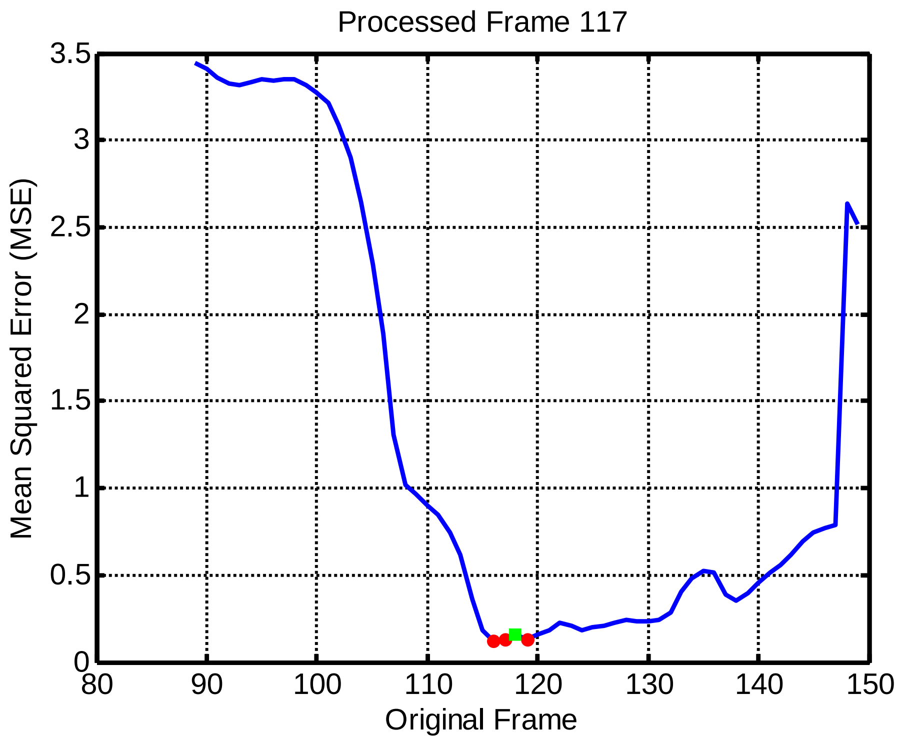

Figure 3. Example *MSE* plot demonstrating the expanded fuzzy set from Step 7.

2.3. Stage 3: Determine Time Segments that Exhibit "Normal Causality"

For progressive video, a normal causal time segment is defined as a contiguous time segment where frame *n* jumps forward in time with respect to frame *n*-1 by 0 to *cjump* (causality jump) frames. Here *cjump* provides a mechanism to limit forward jumps in time. The idea is that normal causal time segments are perceived as natural-looking video (which also includes still video). Stage 4 (Section 2.4) uses the longest such time segment in the processed video clip to anchor the causality estimation of the entire clip. This is based on the observation that the longest normal causal segment normally has fewer time alignment errors (e.g., forward jumps by more than *cjump* can be indicative of noisy and unreliable *MSE* information for various reasons). However, for low frame rate systems that repeat many frames and then jump forward by more than *cjump* frames, this type of categorization will declare each period of frame repetition as a new normal causal segment (i.e., in this case, one normal causal segment will not include both the frame repetitions and the frame update that follows the frame repetitions).

For interlaced video, a normal causal time segment is defined as a contiguous time segment where field *n* jumps forward in time with respect to field *n*-1 by 0 to 2**cjump* fields. In addition, a backward jump of 1 field is allowed for the case of an interlaced video system that repeats frames.

We continue with the step by step numbering used in Section 2.2 to specify the algorithm description (the MATLAB code for Stage 3 begins on page 36).

<u>Step 8)</u> Using the closest matching original frame for each processed frame (i.e., *Final_Fuzzy_Index*$(1,t_p)$ from Step 7 of Section 2.2), search through the processed frames (from earliest to latest) using the jumps in the matching original frame indices to determine the beginning of each normal causal segment i in the processed video clip, *Run_Beg*(i), and its length, *Run_Length*(i). For this search, use *cjump* = 2 with the added constraint that processed frames that aligned with original frame indices on the edge of the search window (as given by the *Edge*(t_p) vector from Step 5 of Section 2.1) are not included.[6] *Edge* points are excluded from the normal causal segments because a better alignment with lower *MSE* might exist beyond the edge.

<u>Step 9)</u> Sort the *Run_Length* vector from Step 8 (from longest to shortest) to produce a new vector *Run_Length_Sort*. Apply the same sort order to the *Run_Beg* vector to produce *Run_Beg_Sort*. After this sort, the normal causal time segment with the longest length is given by *Run_Beg_Sort*(1) and *Run_Length_Sort*(1).

This stage of the algorithm only categorizes normal causal time segments in the processed video clip. There might be some processed frames that do not belong to a normal causal time segment. In addition, consecutive normal causal time segments might not be causal when taken together, as the end of one normal causal time segment might be later in time than the beginning of the next normal causal time segment.

2.4. Stage 4: Fill in Normal Causal Segments from Longest to Shortest

Stage 4 of the algorithm assembles the information from Step 9 of Section 2.3 (i.e., *Run_Length_Sort* and *Run_Beg_Sort*) in such a manner as to preserve causality for the processed video clip. We continue with the step by step numbering used in Section 2.3 to specify the algorithm description (the MATLAB code for Stage 4 begins on page 38).

<u>Step 10)</u> Start with the normal causal time segment with the longest length as given by *Run_Beg_Sort*(1) and *Run_Length_Sort*(1). This time segment will be used as the initial anchor point for assembling all the other normal causal time segments. The longest time segment contains the most consistent time alignments and is less prone to containing time alignment errors, so it forms a natural anchor point for the entire processed video clip. Initialize a vector *Causal* of length N (the total number of processed frames) with zeros. This vector will contain the causal time alignments of the processed video frames. For the longest normal causal segment, simply assign their *Causal* vector elements to the corresponding original frame alignments taken from the array *Final_Fuzzy_Index*$(1,t_p)$ (from Step 7 of Section 2.2). Now, step through the successively shorter normal causal time segments, testing each one to see if causality would be preserved if that segment was inserted into the *Causal* vector.[7] If so, fill in the *Causal* vector elements for that time segment. If not, proceed to the next shorter normal

[6] For example, if when examining the time sequence of matching original frame indices they are observed to jump forward in time by 3 frames, then that would end one normal causal segment and begin a new one.

[7] For progressive video, jumps backward in time are not permitted. For interlaced video, a one-field jump back in time is permitted to allow for the frame repetition case.

causal segment. Continue until all the normal causal time segments have been considered that are longer than *min_length*[8] frames.

Figure 4 demonstrates the Step 10 procedure for the processed clip whose *MSE* function is shown in Figure 2. The thin blue lines plot *Final_Fuzzy_Index*(1,t_p) for all processed frames t_p, which gives the original frames that minimize *MSE* for each processed frame (plotted on the x-axis). As can be seen from the dips in the thin blue lines around processed frame 150, the minimum *MSE* alignment does not produce a causal alignment for this processed clip. The thick red lines plot *Causal*(t_p) for all processed frames t_p after each iteration for the first four iterations (top-left, top-right, bottom-left, bottom-right, in that order). The first iteration (top-left) is after the longest normal causal time segment has been inserted into the *Causal* vector and this serves as an anchor point for the successive iterations. *Causal* values that are zero in these plots have not yet been set (i.e., the *Causal* vector was initialized with zeros). After four iterations (bottom-right), some obvious non-causal time segments remain (e.g., around processed frame 150).

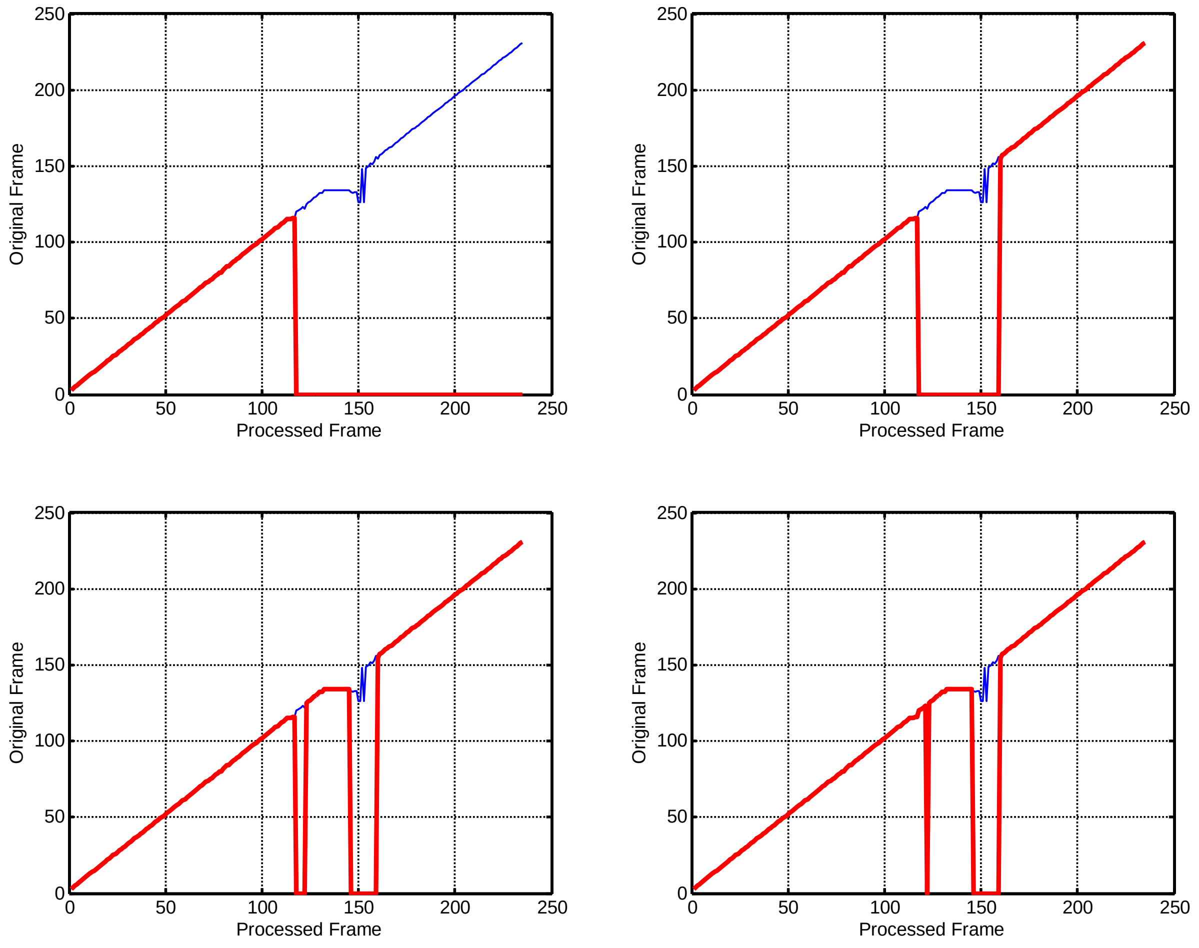

Figure 4. Assembling the causal alignment from normal causal segments (four iterations).

[8] The recommended value for *min_length* is 2, which eliminates causal time segments that are a single point. This eliminates a single mis-aligned frame from consideration, a somewhat common occurrence.

2.5. Stage 5: Fill in Remaining Holes

After filling in all the normal causal time segments, the *Causal* vector may still contain holes (or zero values) for some processed video frames. Hole segments (i.e., segments preceded and followed by valid causal alignments) are filled as they are encountered from early time to late time in the following manner (the MATLAB code for Stage 5 begins on page 40):

Step 11) The earliest point in the hole is filled by the value in the *Final_Fuzzy_Index*$(1,t_p)$ vector, if this is allowed by the causality rules. Then the latest point in the hole is filled by the value in the *Final_Fuzzy_Index*$(1,t_p)$ vector, if this is allowed by the causality rules. This process alternates back and forth until no other extensions are possible for the hole segment being filled. After this, if unfilled points remain in the hole segment, and the *Final_Fuzzy_Index* array has valid next best alignments for these unfilled points (e.g., *Final_Fuzzy_Index*$(2,t_p)$ would be the next best alignment for processed frame t_p if it exists, then *Final_Fuzzy_Index*$(3,t_p)$, etc.), these are examined in turn to see if causality can be achieved using them. Before these secondary insertions are made, they are compared to what linear interpolation (using the closest valid *Causal* points before and after the hole) would have produced to fill in the hole, and the choice that produces the minimum Root Mean Squared Error (RMSE) with respect to the *Final_Fuzzy_Index*$(1,t_p)$ vector is chosen.

In the above manner, holes are filled in as they are encountered from early to late time. When the hole segment occurs at the beginning or end of the time sequence, then the holes are filled by the *Final_Fuzzy_Index*$(1,t_p)$ alignments (if they are causal), or the optimum choice between the next best *Final_Fuzzy_Index*$(*,t_p)$ alignments (if available and if causal) and the last good causal alignment which is replicated/extended over the hole segment. The last good causal alignment is replicated/extended over the hole segment if no other options are available.

Figure 5 demonstrates the output of the hole filling algorithm for the processed video clip in Figure 4. The left plot is before and the right plot is after filling of the hole segments. The final *Causal* time alignment (thick red line) is obtained after this stage of the algorithm.

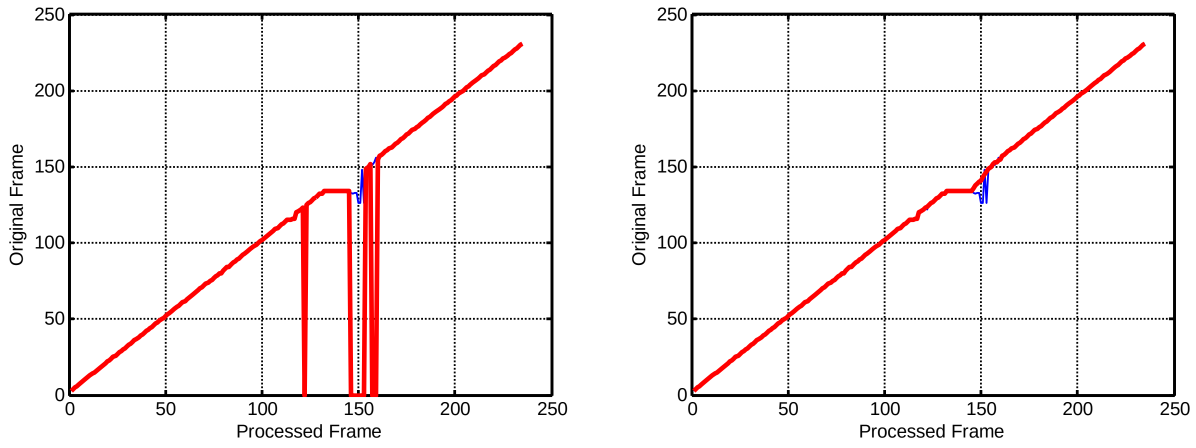

Figure 5. Hole filling example, before (left) and after (right).

2.6. Stage 6: Expand Set of Fuzzy Time Alignments from Stage 2

This optional stage of the algorithm can be used to expand the set of fuzzy alignments given by Stage 2 (Section 2.2) to include the causal time alignments at the end on Stage 5 (Section 2.5). The MATLAB code for Stage 6 begins on page 52.

Step 12) The expanded *Final_Fuzzy_Index* array will be given by *Final_Fuzzy_Causal_Index*, and the expanded *Final_Fuzzy_MSE* array will be given by *Final_Fuzzy_Causal_MSE*. The *Final_Fuzzy_Causal_Index* array is generated by stepping through the processed frames t_p and expanding the set of fuzzy alignments for t_p to include the *Causal*(t_p) alignment point and all points in between this point and the points in *Final_Fuzzy_Index*(*,t_p). The array *Final_Fuzzy_Causal_MSE* gives the corresponding *MSE*s of *Final_Fuzzy_Causal_Index*. The *Final_Fuzzy_Causal_MSE* array is then sorted together with *Final_Fuzzy_Causal_Index* such that the most likely alignments (i.e., those with minimum *MSE*) are first and the least likely alignments (i.e., those with maximum *MSE*) are last. Finally, the *Causal* alignments are moved into the most likely positions in these arrays, regardless of their *MSE*s.

Figure 6 gives an example plot for processed frame 147 of the video clip whose *MSE* function is shown in Figure 2. Figure 6 is similar to Figure 1 and Figure 3 except that this figure demonstrates that the solid black diamond points to the right of the solid red circle point were added to the fuzzy set of alignments during Step 12. The actual causal alignment point that was chosen by the algorithm is also plotted as a solid black diamond point but it is overlaid with a larger open black diamond.

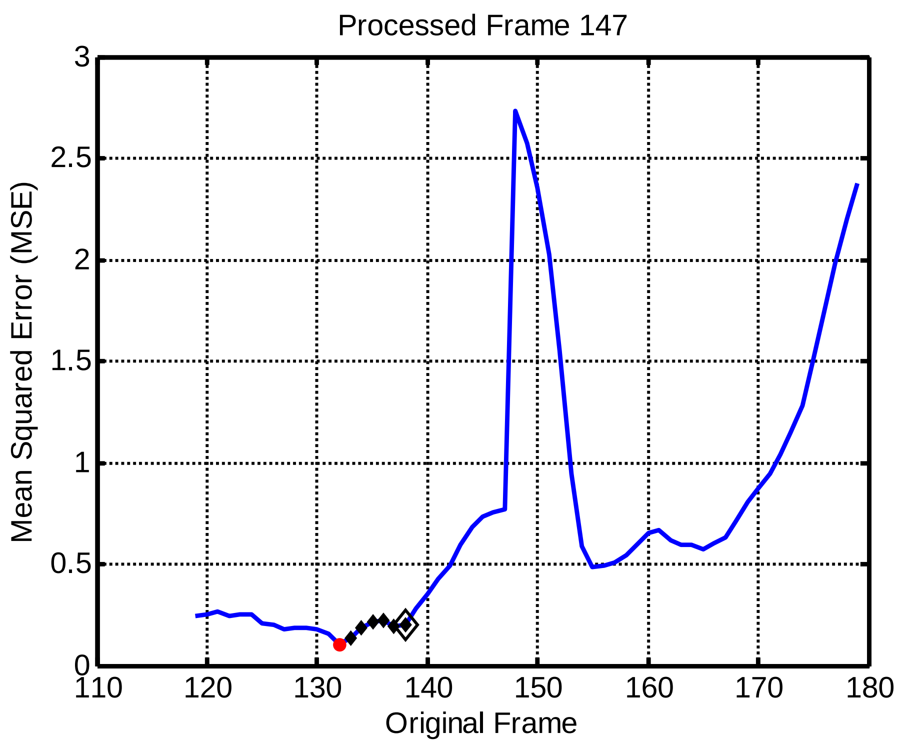

Figure 6. Example *MSE* plot demonstrating the expanded causal fuzzy set from Step 12.

3. SUMMARY

This document has described a full reference method for calculating variable frame delays that might be present in processed video sequences that are output from modern video coding and transmission systems. The full reference method uses the mean squared error (MSE) between normalized processed video frames and their corresponding original video frames (before coding and transmission). The algorithm imposes an additional causality constraint to reduce processed-to-original video frame alignment errors that would result from algorithms which rely solely on minimum MSE values. These alignment errors become more likely for still or nearly still video, video with repetitive motion, highly distorted processed video, or processed video that has partial frame updates (e.g., the foreground updates while the background does not update).

Possible applications for the algorithm presented here include removing variable frame delays from the processed sequence as a calibration step before computing a full reference quality measurement, and computing a video quality parameter that only measures the perceptual effects of variable frame delays.

4. REFERENCES

[1] S. Wolf and M. Pinson, “Video quality measurement techniques,” NTIA Technical Report TR-02-392, Jun. 2002.

[2] ITU-R Recommendation BT.601, “Studio encoding parameters of digital television for standard 4:3 and wide screen 16:9 aspect ratios,” Recommendations of the ITU, Radiocommunication Sector.

[3] M. Pinson and S. Wolf, “Reduced reference video calibration algorithms,” NTIA Technical Report TR-08-433b, Nov. 2007.

APPENDIX: MATLAB Code

The function variable frame delays (vfd) can be compiled (using the MATLAB compiler) and run from a DOS prompt. If compilation is performed, the routine is called as given in the below examples. For instance, the user would type the following at the DOS prompt:

vfd 'proc.yuv' 'orig.yuv' 'interlaced_lff' 'results.csv' 'yuv' 486 720 'reframe' 'causal' 'verbose'

However, if vfd is run from the MATLAB prompt, the user would need to type the following at the MATLAB prompt:

vfd '''proc.yuv''' '''orig.yuv''' '''interlaced_lff''' '''results.csv''' '''yuv''' 486 720 '''reframe''' '''causal''' '''verbose'''

Also please note that the causal option must be specified as a command line argument in order to activate the causal processing described in this document.

```
function vfd(proc_file, orig_file, scan_type, results_file, varargin)
% VFD 'proc_file' 'orig_file' 'scan_type' 'results_file' options
%
%   Estimate the variable frame delays (VFD) of all frames (or fields) in
%   the user specified processed video file ('proc_file') that is in either
%   uncompressed UYVY AVI file format (default) or raw big-YUV file format
%   (optional).  The original video file is given by 'orig_file'.  The user
%   must specify the 'scan_type' of the video files as either
%   'progressive', 'interlaced_uff' (interlaced upper field first), or
%   'interlaced_lff' (interlaced lower field first), since this information
%   is not available in the AVI file format.  VFD results are appended to
%   the file 'results_file' (see the RESULTS section below for a complete
%   description of the results output file).
%
%   This routine does not perform any spatial registration so the original
%   and processed clips are assumed to have been spatially registered
%   beforehand.  The original and processed video files can have a
%   different number of frames but ideally, there should be a matching
%   original frame (or field) for every processed frame (or field).
%
% SYNTAX
%   vfd 'proc_file' 'orig_file' 'scan_type' 'results_file' options
%
```

```
% DESCRIPTION
%    For each frame (or field) in the processed file, this algorithm finds
%    the frame (or field) in the original file that minimizes the mean
%    squared error, subject to the constraints imposed by the optional
%    arguments.  See the RESULTS section below for the results output file
%    format.
%
%   Any or all of the following optional properties may be requested (the
%   'yuv' option is required for yuv files, but not for avi files since
%   this information is read from the avi header).
%
%   'yuv' rows cols        Specifies the number of rows and cols of the
%                          big-YUV files.
%
%   'sroi' top left bottom right    Only use the specified spatial region
%                                   of interest (sroi) for the vfd
%                                   calculation.  By default, all of the
%                                   image is used.  The sroi is inclusive,
%                                   where top/left start at 1.
%
%   'troi' fstart fstop    Only calculate vfd for the specified temporal
%                          region of interest (troi) of the processed video
%                          clip, where fstart and fstop are included and
%                          given in frames.  By default, the vfd of the
%                          entire processed video clip is calculated
%                          (fstart = 1, fstop = number of frames in file).
%
%   'first_align' a           Specifies the best guess for the original
%                             file frame number that corresponds to the
%                             first frame in the processed troi.  By
%                             default, this is set to fstart, which is set
%                             to 1 when troi is not specified.
%
%   't_uncert' t              Specifies the temporal uncertainty (plus or
%                             minus t frames) over which to search.  The
%                             processed remains fixed and the original is
%                             shifted.  The center (zero shift) point for
%                             the first frame (or field) is given by
%                             first_align.  By default, temporal
%                             uncertainty is set to 30 frames.  It can have
%                             a minimum value of 1 frame.  When the
%                             original cannot be shifted by the temporal
%                             uncertainty (e.g., perhaps near the ends of
%                             the sequence), the original will be shifted
%                             up to the maximum extent possible.
```

```
%
%   'reframe'  Allow for the possibility that the processed video clip has
%              been reframing.  This option is only valid for a scan_type
%              of 'interlaced_uff' or 'interlaced_lff'.  Reframing can vary
%              throughout the processed clip, although this should be rare.
%              This option will increase the runtime substantially since
%              extra spatial shifts must be examined, but it should be used
%              if there is any possibility of reframing existing in the
%              processed video clip.  See Section 3.1.2 of NTIA Report
%              TR-02-392 for a definition of reframing.
%
%   'causal'   Impose causality constraint so that later frames (fields) in
%              the processed clip cannot align to original frames (fields)
%              that are earlier in time than found for the proceeding
%              processed frames (fields).  For interlaced video, a
%              one-field jump back in time is allowed since this is
%              indicative of a frozen frame.  By default, causality is
%              turned off (yes, codecs can output non-causal sequences).
%              But specifying the causal option is usually recommended.
%
%   'verbose'   Display output and plots during processing.
%
% RESULTS
%   The output results file is in Comma-Separated Values (CSV format):
%
%   'proc_file', proc_indices
%   'orig_file', orig_indices
%
%   proc_indices and orig_indices are row vectors of length equal to the
%   number of frames in the processed temporal region of interest (for
%   scan_type = 'progressive') or of length equal to 2 * the number of
%   frames in the processed temporal region of interest (for scan_type =
%   'interlaced_uff' or 'interlaced_lff').  For each processed frame (or
%   field) index, the best matching original frame (or field) index is
%   given, where 1 is the first frame (or field) in the file.
%
% EXAMPLES
%   vdf 'proc.avi' 'orig.avi' 'progressive' 'results.csv'
%   vfd 'proc.yuv' 'orig.yuv' 'interlaced_lff' 'results.csv' 'yuv' 486 720 'reframe' 'causal' 'verbose'
%

if nargin == 0,
    fprintf(' vfd ''proc_file'' ''orig_file'' ''scan_type'' ''results_file'' options\n');
    fprintf('\n');
    fprintf('   Estimate the variable frame delays (VFD) of all frames (or fields) in\n');
fprintf('   the user specified processed video file (''proc_file'') that is in either\n');
fprintf('   uncompressed UYVY AVI file format (default) or raw big-YUV file format\n');
fprintf('   (optional).  The original video file is given by ''orig_file''.  The user\n');
fprintf('   must specify the ''scan_type'' of the video files as either\n');
fprintf('   ''progressive'', ''interlaced_uff'' (interlaced upper field first), or\n');
fprintf('   ''interlaced_lff'' (interlaced lower field first), since this information\n');
fprintf('   is not available in the AVI file format.  VFD results are appended to \n');
fprintf('   the file ''results_file'' (see the RESULTS section below for a complete\n');
fprintf('   description of the results output file).  \n');
fprintf('\n');
fprintf('   This routine does not perform any spatial registration so the original\n');
fprintf('   and processed clips are assumed to have been spatially registered\n');
fprintf('   beforehand.  The original and processed video files can have a\n');
fprintf('   different number of frames but ideally, there should be a matching\n');
fprintf('   original frame (or field) for every processed frame (or field).\n');
fprintf('\n');
fprintf(' SYNTAX\n');
fprintf('   vfd ''proc_file'' ''orig_file'' ''scan_type'' ''results_file'' options\n');
fprintf('\n');
fprintf(' DESCRIPTION\n');
fprintf('    For each frame (or field) in the processed file, this algorithm finds\n');
fprintf('    the frame (or field) in the original file that minimizes the mean\n');
fprintf('    squared error, subject to the constraints imposed by the optional\n');
fprintf('    arguments.  See the RESULTS section below for the results output file\n');
fprintf('    format.\n');
fprintf('\n');
fprintf('   Any or all of the following optional properties may be requested (the\n');
fprintf('   ''yuv'' option is required for yuv files, but not for avi files since\n');
fprintf('   this information is read from the avi header).\n');
fprintf('\n');
fprintf('   ''yuv'' rows cols        Specifies the number of rows and cols of the \n');
fprintf('                          big-YUV files. \n');
fprintf('\n');
fprintf('   ''sroi'' top left bottom right    Only use the specified spatial region \n');
fprintf('                                  of interest (sroi) for the vfd\n');
fprintf('                                  calculation.  By default, all of the\n');
fprintf('                                  image is used.  The sroi is inclusive,\n');
fprintf('                                  where top/left start at 1. \n');
fprintf('\n');
fprintf('   ''troi'' fstart fstop    Only calculate vfd for the specified temporal\n');
fprintf('                          region of interest (troi) of the processed video\n');
fprintf('                          clip, where fstart and fstop are included and\n');
fprintf('                          given in frames.  By default, the vfd of the\n');
fprintf('                          entire processed video clip is calculated\n');
fprintf('                          (fstart = 1, fstop = number of frames in file).\n');
fprintf('\n');
fprintf('   ''first_align'' a           Specifies the best guess for the original\n');
fprintf('                             file frame number that corresponds to the\n');
fprintf('                             first frame in the processed troi.  By\n');
fprintf('                             default, this is set to fstart, which is set\n');
fprintf('                             to 1 when troi is not specified.\n');
fprintf('\n');
fprintf('   ''t_uncert'' t              Specifies the temporal uncertainty (plus or\n');
fprintf('                             minus t frames) over which to search.  The\n');
fprintf('                             processed remains fixed and the original is\n');
fprintf('                             shifted.  The center (zero shift) point for\n');
fprintf('                             the first frame (or field) is given by\n');
fprintf('                             first_align.  By default, temporal\n');
fprintf('                             uncertainty is set to 30 frames.  It can have\n');
fprintf('                             a minimum value of 1 frame.  When the\n');
fprintf('                             original cannot be shifted by the temporal \n');
fprintf('                             uncertainty (e.g., perhaps near the ends of \n');
fprintf('                             the sequence), the original will be shifted\n');
fprintf('                             up to the maximum extent possible.\n');
fprintf('\n');
fprintf('   ''reframe''  Allow for the possibility that the processed video clip has\n');
fprintf('              been reframing.  This option is only valid for a scan_type\n');
fprintf('              of ''interlaced_uff'' or ''interlaced_lff''.  Reframing can vary\n');
fprintf('              throughout the processed clip, although this should be rare.\n');
fprintf('              This option will increase the runtime substantially since\n');
fprintf('              extra spatial shifts must be examined, but it should be used\n');
fprintf('              if there is any possibility of reframing existing in the\n');
fprintf('              processed video clip.  See Section 3.1.2 of NTIA Report\n');
fprintf('              TR-02-392 for a definition of reframing.\n');
fprintf('\n');
fprintf('   ''causal''   Impose causality constraint so that later frames (fields) in\n');
fprintf('              the processed clip cannot align to original frames (fields)\n');
fprintf('              that are earlier in time than found for the proceeding\n');
fprintf('              processed frames (fields).  For interlaced video, a\n');
fprintf('              one-field jump back in time is allowed since this is\n');
fprintf('              indicative of a frozen frame.  By default, causality is\n');
fprintf('              turned off (yes, codecs can output non-causal sequences).\n');
fprintf('              But specifying the causal option is usually recommended. \n');
fprintf('\n');
fprintf('   ''verbose''   Display output and plots during processing.\n');
fprintf('\n');
fprintf(' RESULTS\n');
fprintf('   The output results file is in Comma-Separated Values (CSV format):\n');
fprintf('\n');
fprintf('   ''proc_file'', proc_indices\n');
```

```
    fprintf('   ''orig_file'', orig_indices\n');
    fprintf('\n');
    fprintf('   proc_indices and orig_indices are row vectors of length equal to the\n');
    fprintf('   number of frames in the processed temporal region of interest (for\n');
    fprintf('   scan_type = ''progressive'') or of length equal to 2 * the number of \n');
    fprintf('   frames in the processed temporal region of interest (for scan_type =\n');
    fprintf('   ''interlaced_uff'' or ''interlaced_lff'').  For each processed frame (or\n');
    fprintf('   field) index, the best matching original frame (or field) index is\n');
    fprintf('   given, where 1 is the first frame (or field) in the file.\n');
    fprintf('\n');
    fprintf(' EXAMPLES\n');
    fprintf('   vdf ''proc.avi'' ''orig.avi'' ''progressive'' ''results.csv''\n');
    fprintf('   vfd ''proc.yuv'' ''orig.yuv'' ''interlaced_lff'' ''results.csv'' ''yuv'' 486 720 ''reframe'' ''causal'' ''verbose'' \n');
    fprintf('\n');
    return;
end

% strip off the extra single quotes ''
proc_file = eval(proc_file);
orig_file = eval(orig_file);
scan_type = eval(scan_type);
results_file = eval(results_file);

% Validate the scan_type
if (~strcmpi(scan_type,'progressive') && ~strcmpi(scan_type,'interlaced_lff') && ~strcmpi(scan_type,'interlaced_uff'))
    error('Invalid scan_type');
end

% Validate input arguments and set their defaults
is_yuv = 0;  % default file type, uncompressed UYVY AVI
is_whole_image = 1;
is_whole_time = 1;
first_align = 0;
t_uncert = 30;
reframe = 0;
causal = 0;
verbose = 0;
cnt=1;
while cnt <= length(varargin),
    if ~ischar(varargin{cnt}),
        error('Property value passed into vfd is not recognized');
    end
    if strcmpi(eval(char(varargin(cnt))),'yuv') == 1
        rows = str2double(varargin{cnt+1});
        cols = str2double(varargin{cnt+2});
        is_yuv = 1;
        cnt = cnt + 3;
    elseif strcmpi(eval(char(varargin(cnt))),'sroi') == 1
        top = str2double(varargin{cnt+1});
        left = str2double(varargin{cnt+2});
        bottom = str2double(varargin{cnt+3});
        right = str2double(varargin{cnt+4});
        is_whole_image = 0;
        cnt = cnt + 5;
    elseif strcmpi(eval(char(varargin(cnt))),'troi') == 1
        fstart = str2double(varargin{cnt+1});
        fstop = str2double(varargin{cnt+2});
        is_whole_time = 0;
        cnt = cnt + 3;
    elseif strcmpi(eval(char(varargin(cnt))),'first_align') == 1
        first_align = str2double(varargin{cnt+1});
        cnt = cnt + 2;
    elseif strcmpi(eval(char(varargin(cnt))),'t_uncert') == 1
        t_uncert = str2double(varargin{cnt+1});
        cnt = cnt + 2;
    elseif strcmpi(eval(char(varargin(cnt))),'reframe') == 1
        reframe = 1;
        cnt = cnt + 1;
    elseif strcmpi(eval(char(varargin(cnt))),'causal') == 1
        causal = 1;
        cnt = cnt + 1;
    elseif strcmpi(eval(char(varargin(cnt))),'verbose') == 1
        verbose = 1;
        cnt = cnt + 1;
    else
        error('Property value passed into vfd not recognized');
    end
end

% Validate reframing option
if (reframe && strcmpi(scan_type,'progressive'))
    error('Reframe option not allowed for progressive video');
end

% Get the processed and original file information
if (~is_yuv)  % AVI file

    % Get processed file information
    [avi_info] = read_avi('Info',proc_file);
    rows = avi_info.Height;
    cols = avi_info.Width;
    tframes = avi_info.NumFrames;  % total frames in processed file


    % Get original file information
    [avi_info_orig] = read_avi('Info',orig_file);
    rows_orig = avi_info_orig.Height;
    cols_orig = avi_info_orig.Width;
    tframes_orig = avi_info_orig.NumFrames;

else  % big-YUV file

    % Get the processed file information
    [fid, message] = fopen(proc_file, 'r');
    if fid == -1
        fprintf(message);
        error('Cannot open processed big-YUV file %s', proc_file);
    end
    % Find last frame.
    fseek(fid,0, 'eof');
    tframes = ftell(fid) / (2 * rows * cols);
    fclose(fid);

    % Get the original file information
    rows_orig = rows;
    cols_orig = cols;
    [fid, message] = fopen(orig_file, 'r');
    if fid == -1
        fprintf(message);
        error('Cannot open original big-YUV file %s', orig_file);
    end
    % Find last frame.
    fseek(fid,0, 'eof');
    tframes_orig = ftell(fid) / (2 * rows * cols);
    fclose(fid);

end

% Verify that the processed and original are the same resolution
if (rows ~= rows_orig || cols ~= cols_orig)
    error('Processed and original files have different image resolutions.');
end

% Set/Validate the SROI
if (is_whole_image)
```

```
    top = 1;
    left = 1;
    bottom = rows;
    right = cols;
elseif (top<1 || left<1 || bottom>rows || right>cols || top>bottom || left>right)
    error('Invalid spatial region of interest (SROI) for the processed file.');
end

% Set/Validate the TROI of the processed file
if (is_whole_time)
    fstart= 1;
    fstop = tframes;
elseif (fstart<1 || fstop>tframes || fstart>fstop)
    error('Invalid temporal region of interest (TROI) for the processed file.');
end

%  Assign the original first alignment point and validate
if (~first_align)  % a value for first_align was not input by the user so assign default
    first_align = fstart;
end
if (first_align<1 || first_align>tframes_orig)
    error('Invalid first_align for the original file.');
end

%  Validate t_uncert
if (t_uncert<1 || t_uncert>tframes_orig)
    error('Invalid search uncertainty t_uncert.');
end

%  Find the original time segment to read
offset_orig = first_align-fstart;  % search offset of the orig file with respect to the proc file
fstop_orig = min(tframes_orig, fstop+offset_orig+t_uncert);  % the last original frame to read

% Read extra orig frames at the beginning to allow for search range.
% Calculate a new first_align point that is referenced to the first
% original frame that is read rather than to the first frame in the original file.
fstart_orig = max(1, fstart+offset_orig-t_uncert);
new_first_align = first_align-fstart_orig+1;  % no change if I start reading the first frame of orig

% Read in the original and processed S-T segments
if (~is_yuv)  % AVI file

    % Read in video and clear color planes to free up memory
    [yp, cb, cr] = read_avi('YCbCr',proc_file,'frames',fstart,fstop,'sroi',top,left,bottom,right);
    clear cb cr;
    [yo, cb, cr] = read_avi('YCbCr',orig_file,'frames',fstart_orig,fstop_orig,'sroi',top,left,bottom,right);
    clear cb cr;

else  % big-YUV file

    % Read in video and clear color planes to free up memory
    [yp] = read_bigyuv(proc_file,'frames',fstart,fstop,'size',rows,cols,'sroi',top,left,bottom,right);
    [yo] = read_bigyuv(orig_file,'frames',fstart_orig,fstop_orig,'size',rows,cols,'sroi',top,left,bottom,right);

end

% Generate the function call with all the desired options
func_call = 'est_var_frame_delays(yp,yo,''normalize'',';  % always use the normalize option as it seems to work the best

if (reframe)
    func_call = strcat(func_call,'''reframe'',');
end

if (causal)
    func_call = strcat(func_call,'''causal'',');
end

if (verbose)
    func_call = strcat(func_call,'''verbose'',');
end

if (strcmpi(scan_type,'interlaced_lff'))
    func_call = strcat(func_call,'''interlaced'',1,');
end

if (strcmpi(scan_type,'interlaced_uff'))
    func_call = strcat(func_call,'''interlaced'',2,');
end

func_call = strcat(func_call,'''first_align'',',num2str(new_first_align),',','''t_uncert'',',num2str(t_uncert),')');

% Call the est_var_frame_delays function to get results
[results results_rmse results_fuzzy results_fuzzy_mse] = eval(func_call);

% Translate results to use the orig and proc file indexing
if (strcmpi(scan_type,'progressive'))
    proc_indices = (fstart-1) + (1:length(results));
    orig_indices = (fstart_orig-1) + results;
else % interlaced
    proc_indices = 2*(fstart-1) + (1:length(results));
    orig_indices = 2*(fstart_orig-1) + results;
end

% Save results
fid_results = fopen(results_file,'a');  % open results file for appending

if (strcmpi(scan_type,'progressive'))
    fprintf(fid_results,'File Name, Matching Frame Indices\n');
else % interlaced
    fprintf(fid_results,'File Name, Matching Field Indices\n');
end

npts = length(proc_indices);

fprintf(fid_results,'%s, ',proc_file);
for i = 1:npts-1
    fprintf(fid_results,'%f, ',proc_indices(i));
end
fprintf(fid_results,'%f\n',proc_indices(i+1));

fprintf(fid_results,'%s, ',orig_file);
for i = 1:npts-1
    fprintf(fid_results,'%f, ',orig_indices(i));
end
fprintf(fid_results,'%f\n',orig_indices(i+1));
fclose(fid_results);

close all;

end


function [results, results_rmse, results_fuzzy, results_fuzzy_mse] = est_var_frame_delays(proc, orig, varargin)
% EST_VAR_FRAME_DELAYS
%   Estimate the variable delays of each frame in a processed video clip
%   (i.e., proc) given an original video clip (i.e., orig).  The processed
%   and original clips are three dimensional (rows x cols x frames) Y (luma)
%   matrices that may have a different number of frames.  The results row
%   vector gives the best matching frame number in the original clip for each
%   frame in the processed clip, where original frame indices start from 1.
%   The user may optionally specify interlaced format, and then the results
%   vector gives the best matching field number (i.e., the results vector
%   will be twice the frame-length of the processed clip).  Note that this
%   routine does not perform spatial registration and/or gain and level
%   offset so the orig and proc clips are assumed to have been fully
%   calibrated.
%
% SYNTAX
%   [results, results_rmse, results_fuzzy, results_fuzzy_mse] = est_var_frame_delays (proc, orig, options)
%
% DESCRIPTION
%   For each frame in the processed clip, this algorithm finds the frame in
%   the original clip that minimizes the mean squared error, subject to the
%   constraints imposed by the optional inputs.  The following output
%   arguments can be requested:
%
%   results       A row vector of the same length as the processed clip in
%                 frames (or fields) that gives the best matching original
%                 frame (or field) for each processed frame (or field).
%                 The original frame (or field) indices are assumed to
%                 start at one.  If the 'causal' option is NOT specified, this
%                 is merely the original frame (or field) with the smallest
%                 Mean Squared Error (MSE) when compared to the processed
%                 frame.  If the 'causal' option is specified, then the
%                 results row vector contains the results from the causal
%                 filtering algorithm (see below).
%
%   results_rmse  If two output arguments are requested, the second one
%                 will contain the Root Mean Squared Error (RMSE) between
%                 the causal alignment and the unfiltered and possibly
%                 non-causal alignment (in frames or fields).  The higher
%                 the value, the more difference there is between the
%                 the two algorithms, which probably indicates that the
%                 scene is difficult to align (e.g., a scene with a small
%                 amount of motion or repetitive motion), or that not
%                 enough temporal uncertainty (t_uncert) was used.
%
%   results_fuzzy If three output arguments are requested, the third one
%                 will contain the fuzzy alignment results.  This is a
%                 matrix where each column gives a set of rank sorted fuzzy
%                 original frame (or field) alignments, sorted from most
%                 likely to least likely.  The first element of each column
%                 vector gives the most likely alignment for that processed
%                 frame (field) and is the same as the first output
%                 argument (results).  The number of rows in each column is
%                 equal to how many frames (or fields) were searched, which
%                 depends upon t_uncert.  But only likely alignments are
%                 included and the remainder of the rows are filled in by
%                 'NaN' (Not-a-Number).  The fuzzy matrix is influenced by
%                 the 'causal' option since the range of possible
%                  alignments may be expanded to include the causal
%                  alignments.
%
%   results_fuzzy_mse If four output arguments are requested, the fourth one
%                      will contain the Mean Squared Error of each fuzzy
%                      frame (or field) alignment, so this matrix is the
%                      same size as results_fuzzy.
%
% OPTIONS
%   Any or all of the following optional inputs may be requested to control
%   the behavior of the algorithm.
%
%   'interlaced',first_field  Specifies interlaced format, where first_field
%                              specifies which field is first in time,
%                              first_field = 1 if the lower field is first,
%                              first_field = 2 if the upper field is first.
%                              The orig and proc clips must have an even
%                              number of rows for this option.
%
%   'sroi',top,left,bottom,right    Only use the specified spatial region
%                                    of interest (sroi) for the frame delay
%                                    calculation.  sroi is referenced to the
%                                    original frame, where the (top, left)
%                                    corner of the image is (1, 1).  For
%                                    interlaced video, top must be odd and
%                                    bottom must be even. By default, sroi
%                                    is the entire image.
%
%   'first_align',a           Specifies the best guess for the original
%                              frame (or field) number that corresponds to
%                              the first frame (or field) in the processed
%                              clip.  Set to 1 by default.  Note that for
%                              interlaced video, this value must be in
%                              fields, not frames!
%
%   't_uncert',t              Specifies the temporal uncertainty (always
%                              plus or minus t frames, NOT fields) over which
%                              to search.  The processed remains fixed and
%                              the original is shifted.  The center (zero
%                              shift) point for the first frame (or field)
%                              is given by first_align. By default,
%                              temporal uncertainty is set to 30.  When the
%                              original cannot be shifted by the
%                              temporal uncertainty (e.g., near the ends of
%                              the sequence), the original will be shifted
%                             by the maximum possible.
%
%   'normalize'     Perform a normalization on the original and processed
%                   clips such that the processed and original clips have
%                   zero mean, unit variance.  All frames in the processed
%                   clip are used to estimate its mean and variance.  For
%                   the original clip, the first_align point is used to
%                   select an equal number of frames from the original clip
%                   upon which to base its mean and variance (provided
%                   these corresponding frames are available, if not, the
%                   number of frames for the original's mean and variance
%                   estimate will be reduced).  By default, no
%                   normalization is performed.  However, normalization is
%                   a recommended option unless the processed video is
%                   perfectly calibrated wrt gain and level offset.
%
%   'causal'   Impose causality constraint so that later frames (fields) in
%              the processed clip cannot align to original frames (fields)
%              that are earlier in time than found for the proceeding
%              processed frames (fields).  For interlaced video, a
%              one-field jump back in time is allowed since this is
%              indicative of a frozen frame.  By default, causality is
%              turned off (yes, codecs can output non-causal sequences).
%              But specifying the causal option is usually recommended.
%
%   'reframe'  Allow for the possibility that the processed video clip has
%              been reframing.  Must also specify the 'interlaced' option.
%              Reframing can vary throughout the processed clip, although
%              this should be rare.  This option will increase the
%              runtime substantially since extra spatial shifts must be
%              examined.
%
%   'verbose'   Display output during processing.  verbose mode is turned
%               off by default.
%
% EXAMPLES
%

% Return values of function if failure
results = 0;
results_rmse = 0;
results_fuzzy = 0;
results_fuzzy_mse = 0;

% Set input arguments to their defaults and assign optional inputs
```

```
interlaced = 0;
is_whole_image = 1;
first_align = 1;
t_uncert = 30;
normalize = 0;
is_causal = 0;
reframe = 0;
verbose = 0;
% Assign optional inputs
cnt=1;
while cnt <= length(varargin),
    if strcmpi(varargin(cnt),'interlaced') == 1
        interlaced = 1;
        first_field = varargin{cnt+1};
        if (first_field~=1 && first_field~=2)
            error('first_field must be 1 or 2.');
        end
        cnt = cnt + 2;
    elseif strcmpi(varargin(cnt),'sroi') == 1
        is_whole_image = 0;
        top = varargin{cnt+1};
        left = varargin{cnt+2};
        bottom = varargin{cnt+3};
        right = varargin{cnt+4};
        cnt = cnt + 5;
    elseif strcmpi(varargin(cnt),'first_align') == 1
        first_align = varargin{cnt+1};
        cnt = cnt + 2;
    elseif strcmpi(varargin(cnt), 't_uncert') == 1
        t_uncert = varargin{cnt+1};
        cnt = cnt + 2;
    elseif strcmpi(varargin(cnt), 'normalize') == 1
        normalize = 1;
        cnt = cnt + 1;
    elseif strcmpi(varargin(cnt), 'causal') == 1
        is_causal = 1;
        cnt = cnt + 1;
    elseif strcmpi(varargin(cnt), 'reframe') == 1
        reframe = 1;
        cnt = cnt +1;
    elseif strcmpi(varargin(cnt),'verbose') == 1
        verbose = 1;
        cnt = cnt +1;
    else
        error('Property value passed into est_frame_delays not recognized');
    end
end

% Temporal uncertainty search bound check
if(t_uncert < 1)
    error('t_uncert must be at least 1 frame.');
end

%  Find image resolution and number of frames in orig and proc clips
[nr, nc, nf] = size(proc);
[nr_o, nc_o, nf_o] = size(orig);
if (nr ~= nr_o || nc ~= nc_o)
    error('Orig and proc clips must have the same number of rows and cols.');
end

% Validate the SROI
if (is_whole_image) % make ROI whole image
    top = 1;
    left = 1;
    bottom = nr;
    right = nc;
elseif (top<1 || left<1 || bottom>nr || right>nc || top>bottom || left>right)
    error('Requested SROI incompatible with image size.');
end

% Additional checks on SROI for interlaced video
if (interlaced && (mod(nr,2)~=0))
    error('Number of rows must be even.')
end
if (interlaced && (mod(top+1,2)~=0 || (mod(bottom,2)~=0)))
    error('Requested SROI invalid for interlaced video.');
end
if (reframe && ~interlaced)
    error('For the reframe option, you must also specify the interlaced option.')
end

% Split into fields from the get-go if this is interlaced video
if (interlaced)

    % This is the normal split (not reframing)
    if (first_field == 2)  % upper field is first
        origf = reshape(orig(top:bottom,left:right,:), 2, (bottom-top+1)/2, right-left+1, nf_o);
        orig = reshape(permute(origf, [2 3 1 4]), (bottom-top+1)/2, right-left+1, 2*nf_o);
        clear origf;
        procf = reshape(proc(top:bottom,left:right,:), 2, (bottom-top+1)/2, right-left+1, nf);
        proc = reshape(permute(procf, [2 3 1 4]), (bottom-top+1)/2, right-left+1, 2*nf);
        clear procf;
    else  % lower field is first
        origf = flipdim(reshape(orig(top:bottom,left:right,:), 2, (bottom-top+1)/2, right-left+1, nf_o),1);
        orig = reshape(permute(origf, [2 3 1 4]), (bottom-top+1)/2, right-left+1, 2*nf_o);
        clear origf;
        procf = flipdim(reshape(proc(top:bottom,left:right,:), 2, (bottom-top+1)/2, right-left+1, nf),1);
        proc = reshape(permute(procf, [2 3 1 4]), (bottom-top+1)/2, right-left+1, 2*nf);
        clear procf;
    end

    % Adjust t_uncert and number of fields in orig and proc
    t_uncert = 2*t_uncert;
    nf = 2*nf;
    nf_o = 2*nf_o;

    % Number of samples used for mse calculation, field based
    nrows = (bottom-top+1)/2;
    ncols = right-left+1;
    nsamps = nrows*ncols;

else  % Progressive sequence

    % Window out desired SROI
    orig = orig(top:bottom, left:right, :);
    proc = proc(top:bottom, left:right, :);

    % Number of samples used for mse calculation
    nrows = bottom-top+1;
    ncols = right-left+1;
    nsamps = nrows*ncols;

end

% Check on alignment of first frame (or field) before we start searching
if (first_align<1 || first_align>nf_o)
    error('Requested first_align is not valid for orig clip.');
end

%%%%%%%%%%
% Stage 1
% Normalize video sequences and compute the Mean Squared Error (MSE)
% between each processed frame (or field) and the set of original frames
% (or fields) within the temporal search window.
%%%%%%%%%%
```

```
% Perform the normalization on the orig and proc clips (if requested)
if (normalize)

    % Compute mean and stdev of the proc clip and normalize
    proc_mean = sum(reshape(proc, nsamps*nf, 1))/(nsamps*nf);
    proc_std = sqrt(sum(reshape(proc, nsamps*nf, 1).^2)/(nsamps*nf) - proc_mean^2);
    proc = (proc-proc_mean)/proc_std;

    % Compute the mean and stdev of the orig clip and normalize
    nf_o1 = first_align;  % The first frame (or field) in the orig to use
    nf_o2 = min(first_align + nf -1, nf_o);  % The final frame (or field) in the orig to use
    orig_mean = sum(reshape(orig(:,:,nf_o1:nf_o2), nsamps*(nf_o2-nf_o1+1), 1))/(nsamps*(nf_o2-nf_o1+1));
    orig_std = sqrt(sum(reshape(orig(:,:,nf_o1:nf_o2), nsamps*(nf_o2-nf_o1+1), 1).^2)/(nsamps*(nf_o2-nf_o1+1)) - orig_mean^2);
    orig = (orig-orig_mean)/orig_std;

end

%  Find the best matching orig frame (field) for every proc frame (Field)
if (verbose)
    fprintf('proc frame/field   orig frame/field   mse\n');
    fprintf('----------------   ----------------   ---\n');
end
max_size = 2*t_uncert+1;  % the max size of the alignment results
mse = NaN(max_size,nf);  % holds the mse alignment results
offsets = []; % holds the orig index offsets
for t = 1:nf  % Loop over all processed frames

    this_align = first_align+t-1;  % when t = 1, this is the reference alignment
    neg = max(this_align-t_uncert,1);  % the negative most orig index to search
    pos = min(this_align+t_uncert,nf_o);  % the positive most orig index to search
    this_proc = repmat(squeeze(proc(:,:,t)),[1 1 pos-neg+1]);  % Create replicas of this processed frame

    if(~reframe)  % Progressive or interlaced with no reframing

        this_mse = sum(reshape((orig(:,:,neg:pos)-this_proc).^2, nsamps, pos-neg+1), 1) / nsamps;
        [best_mse best_ind] = min(this_mse);
        best_ind = best_ind + neg - 1;

    else  % Must do extra comparisons shifted by 1 line, but all comparisons use the same lines from orig

        origt = orig(2:nrows-1,:,neg:pos);

        %  First comparison, proc is not shifted
        this_mse1 = sum(reshape((origt-this_proc(2:nrows-1,:,:)).^2, nsamps-2*ncols, pos-neg+1), 1) / (nsamps-2*ncols);
        %  Second comparison, proc shifted down by one line wrt orig
        this_mse2 = sum(reshape((origt-this_proc(1:nrows-2,:,:)).^2, nsamps-2*ncols, pos-neg+1), 1) / (nsamps-2*ncols);

        %  Third comparison, proc shifted up by one line wrt orig
        this_mse3 = sum(reshape((origt-this_proc(3:nrows,:,:)).^2, nsamps-2*ncols, pos-neg+1), 1) / (nsamps-2*ncols);

        %  Combine comparisons and take min overall
        this_mse = min([this_mse1;this_mse2;this_mse3]);
        [best_mse best_ind] = min(this_mse);
        best_ind = best_ind + neg - 1;

    end

    % Save the offset indices; the mse row indices plus this offset
    % is the original frame (field) indices
    offsets(t) = neg-1;

    % save mse vector time history for later fuzzy processing
    mse(1:pos-neg+1,t) = this_mse';

    if (verbose)

        % Plot the correlation function for this processed frame
        figure(1)
        plot(offsets(t)+(1:pos-neg+1),this_mse,'LineWidth',2);
        hold on
        set(gca,'LineWidth',2)
        set(gca,'FontName','Ariel')
        set(gca,'fontsize',12)
        xlabel('Original Frame or Field');
        ylabel('Mean Squared Error (MSE)');
        this_title = ['Processed Frame or Field ' int2str(t)];
        title(this_title);
        grid on
        hold off
        pause(0.01);

        % Print out the best matching point
        fprintf('     %4i              %4i          %5.4e\n', t, best_ind, best_mse);

    end

end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Process the correlation results to compute the four output arguments.
%  WARNING:
%  This algorithm is a highly complicated heuristic multi-stage algorithm.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[range num_frames] = size(mse);

% Generate matrix that provides the matching original frame/field for
% every MSE value in the matrix
orig_index = repmat((1:range)',1,num_frames) + repmat(offsets,range,1);

% Sort the MSE values and their orig indices for each proc frame/field
[mse_sort index_sort] = sort(mse);
orig_index_sort = [];
for j = 1:num_frames
    orig_index_sort(:,j) = orig_index(index_sort(:,j),j);
end

% Define a Boolean array that is 1 when the best alignment has hit the
% edge of the search range and zero otherwise (i.e., not enough
% temporal uncertainty or perhaps the correlation function has gone
% berserk).  This array will be used as a filter later to de-weight
% these points in the causal alignment estimation.  When the
% best aligned frame (first element of index_sort) is equal to 1 or
% range, you have hit the edge.
edge = zeros(1, num_frames);
edge(index_sort(1,:) == 1) = 1;  % left search edge
edge(index_sort(1,:) == range) = 1;  % right search edge

%%%%%%%%%%
% Stage 2
% Step thru the frames for this clip and find an initial set of fuzzy alignments
%%%%%%%%%%
final_fuzzy_index = NaN(range, num_frames); % The final fuzzy alignment from most to least likely at end of stage 1
final_fuzzy_mse = NaN(range, num_frames); % Their corresponding MSEs
for j = 1:num_frames

    % Since subtractive correlation is being used, the correlation function
    % will be a minimum at the best match and increase from there.  This
    % code uses a thresholding scheme that sets a threshold at the minimum
    % correlation value + thres * maximum correlation value, but the
    % maximum correlation value used is the top (95%) rank sorted value
    % for robustness.  All alignments within this threshold from the best
    % correlation value are considered possible alignments for later
    % processing by the algorithm.
```

```
thres = 0.005;  % Fraction of the maximum correlation level above the minimum for valid fuzzy
top = 0.95;  % Rank sorted fraction used to determine the max correlation level, for robustness
this_mse = mse(:,j);
this_mse_valid = find(~isnan(this_mse));  % array can contain NaN if orig frames did not exist
range_valid = length(this_mse_valid);
this_mse_sort = sort(this_mse);
this_index = orig_index(:,j);
this_index_sort = orig_index_sort(:,j);

max_corr = this_mse_sort(floor(top*range_valid));
min_corr = this_mse_sort(1);
fuzzy = find(this_mse >= min_corr & this_mse <= min_corr+thres*max_corr);
num_fuzzy = length(fuzzy);  % will always be at least one
fuzzy_sort = [];
for k = 1:num_fuzzy
    fuzzy_sort(k) = find(this_index_sort == this_index(fuzzy(k)));
end

% Expand the fuzzy alignments to include the range of original indices
% that are covered by the minimum MSE points.  This will include extra
% points that were not in the minimum MSE fuzzy set.
bigger_fuzzy = (fuzzy(1):fuzzy(length(fuzzy)))';
bigger_fuzzy_mse = this_mse(bigger_fuzzy);
bigger_fuzzy_index = this_index(bigger_fuzzy);

% Added fuzzy points that were not included in the original set
added_fuzzy = setxor(fuzzy,bigger_fuzzy);
added_fuzzy_mse = this_mse(added_fuzzy);
added_fuzzy_index = this_index(added_fuzzy);
num_added_fuzzy = length(added_fuzzy);
added_fuzzy_sort = [];
for k = 1:num_added_fuzzy
    added_fuzzy_sort(k) = find(this_index_sort == this_index(added_fuzzy(k)));
end

% Sort the bigger fuzzy information from the most likely alignment
% to the least likely alignment, based on MSE.  The array
% final_fuzzy_index gives the best fuzzy alignments at this stage of
% the algorithm without causal processing.  So final_fuzzy_index and
% final_fuzzy_mse will be assigned to results_fuzzy and
% results_fuzzy_mse if causal output was not requested by the user.
[bigger_fuzzy_mse_sort mse_order] = sort(bigger_fuzzy_mse);
bigger_fuzzy_index_sort = bigger_fuzzy_index(mse_order);
final_fuzzy_index(1:length(bigger_fuzzy_index_sort),j) = bigger_fuzzy_index_sort;
final_fuzzy_mse(1:length(bigger_fuzzy_index_sort),j) = bigger_fuzzy_mse_sort;
```

```
end

%  The array best holds the best alignments based on min MSE
best = final_fuzzy_index(1,:);

%%%%%%%%%%
% Stage 3
% Find normal causal segments
%%%%%%%%%%
% Find normal causal segments using only the best alignment point.  A
% normal causal segment is defined as follows:
%
% For interlaced video: A segment where field n jumps forward in
% alignment by 0 to 2*cjump fields (a field repeating system will cause a
% 2 field jump) with respect to field n-1.  A jump back of 1 field during
% the segment is allowed for the frame repeating case.
%
% For progressive video: A segment where frame n jumps forward in
% alignment by 0 to cjump frames with respect to frame n-1.
%
% A frame alignment that is on the edge will not be included in any
% normal causal segment.
run_beg = [];  % Holds the beginning index of a run
run_length = [];  % Holds the length of the run
ri = 1;  % Index counter for the causal runs
cjump = 2;  % Specifies the jump forward (in frames) that is allowed for 'normal' causal

k = 1;  % begin search at the first frame
while(edge(k))
    k = k+1;
    if (k > num_frames)
        break;
    end
end

if(k <= num_frames)  % Found at least one causal alignment
    run_beg(ri) = k;  % Beginning of first causal run
    run_length(ri) = 1;
    k = k+1;

    while(k <= num_frames)  % Complete search for causal runs

        if(~edge(k))
            if(~interlaced)  % Progressive algorithm
            if( (best(k)-best(k-1)>=0) && (best(k)-best(k-1)<=cjump) )
                run_length(ri) = run_length(ri)+1;
                k = k +1;
            else
                % Skip ahead to the next valid causal alignment
                while(edge(k))
                    k = k+1;
                    if (k > num_frames)
                        break;
                    end
                end
                if (k <= num_frames)
                    ri = ri + 1;
                    run_beg(ri) = k;
                    run_length(ri) = 1;
                    k = k + 1;
                end
            end
        else  % Interlaced algorithm
            if( (min(repmat(best(k),1,run_length(ri))-best(k-run_length(ri):k-1))>=-1) && (best(k)-best(k-1)<=2*cjump) )
                run_length(ri) = run_length(ri)+1;
                k = k +1;
            else
                % Skip ahead to the next valid causal alignment
                while(edge(k))
                    k = k+1;
                    if (k > num_frames)
                        break;
                    end
                end
                if (k <= num_frames)
                    ri = ri + 1;
                    run_beg(ri) = k;
                    run_length(ri) = 1;
                    k = k +1;
                end
            end
        end
    else % hit an edge alignment, close off this causal segment
        % Skip ahead to the next valid causal alignment
        while(edge(k))
            k = k+1;
            if (k > num_frames)
                break;
            end
            end
            if (k <= num_frames)
                ri = ri + 1;
                run_beg(ri) = k;
                run_length(ri) = 1;
                k = k +1;
            end
        end

    end

    % Sort the causal runs according to their length
    [run_length_sort length_order] = sort(run_length,'descend');
    run_beg_sort = run_beg(length_order);

end

%%%%%%%%%%
% Stage 4
% Fill normal causal segments from the longest to the shortest
%%%%%%%%%%
causal = zeros(1,num_frames);  % Holds the non-fuzzy causal alignment

%  This code will fill in the normal causal segments from the longest
%  to the shortest.  To qualify for a fill, a normal causal segment
%  must have at least min_length points.
if(~isempty(run_length))

    min_length = 2;
    for k = 1:length(run_length_sort)

        if (run_length_sort(k) >= min_length)

            % See if there are any runs before or after the current run
            if (run_beg_sort(k)+run_length_sort(k) <= num_frames)
                after_index = find(causal(run_beg_sort(k)+run_length_sort(k):num_frames) ~= 0);
            else
                after_index = [];
            end
            if (run_beg_sort(k)-1 >= 1)
                before_index = find(causal(1:run_beg_sort(k)-1) ~= 0);
            else
                before_index = [];
            end
```

```
% Determine the index of the nearest after and before for
% progressive algorithm
if(~isempty(before_index))
    before_orig = before_index(length(before_index));
end
if (~isempty(after_index))
    after_orig = run_beg_sort(k)+run_length_sort(k)+after_index(1)-1;
end

% If causal, fit this run's alignment into the existing
if (~interlaced) % Progressive algorithm
    if ( isempty(before_index) && isempty(after_index) )
        causal(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1) = ...
            best(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1);
    elseif ( isempty(before_index) )
        if (best(after_orig)>=best(run_beg_sort(k)+run_length_sort(k)-1))
            causal(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1) = ...
                best(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1);
        end
    elseif ( isempty(after_index) )
        if (best(before_orig)<=best(run_beg_sort(k)))
            causal(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1) = ...
                best(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1);
        end
    elseif ((best(before_orig)<=best(run_beg_sort(k))) && (best(after_orig)>=best(run_beg_sort(k)+run_length_sort(k)-1)))
        causal(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1) = ...
            best(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1);
    end
else  % Interlaced algorithm allows for up to 1 field jump back in time for the segment you are adding into timeline
    if ( isempty(before_index) && isempty(after_index) )
        causal(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1) = ...
            best(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1);
    elseif ( isempty(before_index) )
        if (min(best(after_index+run_beg_sort(k)+run_length_sort(k)-1)) >= ...
                max(best(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1))-1)
            causal(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1) = ...
                best(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1);
        end
    elseif ( isempty(after_index) )
        if (max(best(before_index))<=min(best(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1))+1)
            causal(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1) = ...
                best(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1);
        end
    elseif ( (min(best(after_index+run_beg_sort(k)+run_length_sort(k)-1)) >= ...
            max(best(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1))-1) && ...
```

```
                        (max(best(before_index))<=min(best(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1))+1) )
                    causal(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1) = ...
                        best(run_beg_sort(k):run_beg_sort(k)+run_length_sort(k)-1);
                end
            end

        end

    end

end

%%%%%%%%%%
% Stage 5
% Fill in the missing holes in the causal array - see algorithm description
%%%%%%%%%%

%  At this point the causal array may still contain holes (zeros).  The
%  algorithm for filling in these segments is as follows: The earlier
%  time point is extended later in time by the value in the best array
%  if this is allowed by the causality rules, then the later time point
%  is extended earlier in time by the value in the best array if this
%  is allowed by the causality rules, alternating back and forth until
%  no other extensions are possible.  If causality cannot be achieved,
%  and the final_fuzzy_index has valid next best alignments (i.e.,
%  something other than 'NaN'), these are examined to see if causality
%  can be achieved using them rather than the best array.  If
%  insertions are made from the final_fuzzy_index array, these are
%  compared to what interpolation would have yielded to fill in the
%  segment hole - and the choice that produces the minimum RMSE with
%  respect to the best array is chosen.
%
%  In this manner, missing segments are filled in as they are
%  encountered from early to late time. When the missing segment occurs
%  at the beginning or end of the time sequence, then the holes are
%  filled by the best array alignments (if they are causal), or the
%  optimum choice between final_fuzzy_index alignments (if available
%  and if causal) and the last good causal alignment which is
%  replicated/extended. The last good causal alignment is
%  replicated/extended if no other options are available.

holes = find(causal==0);
while(~isempty(holes))  % keep filling hole segments from early to late time
    num_holes = length(holes);
```

```
% Find the length of the first hole
hole_start = holes(1);
hole_stop = hole_start;
if (num_holes > 1)
    for k = 2:num_holes
        if (holes(k)-holes(k-1) == 1)
            hole_stop = hole_stop+1;
        else
            break;
        end
    end
end

% Fill the hole segment - there are four cases
% 1. Segment is at beginning of clip but is followed by valid causal
% 2. Segment is at end of clip but is preceded by valid causal
% 3. Segment is preceded and followed by valid causal (most likely)
% 4. Segment is not proceeded or followed by valid causal (function aborts)

% Do progressive filling algorithm first
if(~interlaced)

    % Case 3, probably the most likely case
    if (hole_start > 1 && hole_stop < num_frames)

        % This code toggles from beg to end, starting at beg
        hole_pts = hole_stop-hole_start+1;  % number of points in this hole segment
        valid_early = hole_start-1;  % last valid causal point before time segment
        valid_late = hole_stop+1; % last valid causal point after time segment
        for k = 1:hole_pts
            if (mod(k,2)==1) % try to fill from the beginning first and then from the end if that fails
                if (best(valid_early+1)>=causal(valid_early) && best(valid_early+1)<=causal(valid_late))
                    causal(valid_early+1) = best(valid_early+1);
                    valid_early = valid_early + 1;
                elseif (best(valid_late-1)<=causal(valid_late) && best(valid_late-1)>=causal(valid_early))
                    causal(valid_late-1) = best(valid_late-1);
                    valid_late = valid_late - 1;
                else % see if other possible alignments exist besides the best array
                    early_insert = 0;  % tells if this is an early or late insert
                    succeed = 0;  % set to 1 when a causal substitution is made
                    depth = 2;  % the final_fuzzy_index level, depth = 1 is the best array
                    while (~succeed && (~isnan(final_fuzzy_index(depth,valid_early+1)) || ...
                            ~isnan(final_fuzzy_index(depth,valid_late-1))) )
                        if (~isnan(final_fuzzy_index(depth,valid_early+1)))
                            if (final_fuzzy_index(depth,valid_early+1) >= causal(valid_early) && ...
                            final_fuzzy_index(depth,valid_early+1)<=causal(valid_late))
                        causal(valid_early+1) = final_fuzzy_index(depth,valid_early+1);
                        valid_early = valid_early + 1;
                        succeed = 1;
                        early_insert = 1;
                    end
                else
                    if (final_fuzzy_index(depth,valid_late-1)<=causal(valid_late) && ...
                            final_fuzzy_index(depth,valid_late-1)>=causal(valid_early))
                        causal(valid_late-1) = final_fuzzy_index(depth,valid_late-1);
                        valid_late = valid_late - 1;
                        succeed = 1;
                    end
                end
                depth = depth +1;
                if (depth > range)
                    break;
                end
            end
            if (~succeed) % linear interpolation at beginning as last resort after all possible alignments exhausted
                causal(valid_early+1) = causal(valid_early) + ...
                    round((causal(valid_late)-causal(valid_early))/(valid_late-valid_early));
                valid_early = valid_early + 1;
            else % see if interpolation yields better RMSE than the alignment from the final_fuzzy_index
                if (early_insert)
                    interp_value = causal(valid_early-1) + ...
                        round((causal(valid_late)-causal(valid_early-1))/(valid_late-(valid_early-1)));
                    if (abs(causal(valid_early)-best(valid_early)) > abs(interp_value-best(valid_early)))
                        causal(valid_early) = interp_value;
                    end
                else % late insert
                    interp_value = causal(valid_late+1) - ...
                        round((causal(valid_late+1)-causal(valid_early))/(valid_late+1-valid_early));
                    if (abs(causal(valid_late)-best(valid_late)) > abs(interp_value-best(valid_late)))
                        causal(valid_late) = interp_value;
                    end
                end
            end
        end
    else % try to fill from the end first and then from the beginning if that fails
        if (best(valid_late-1)<=causal(valid_late) && best(valid_late-1)>=causal(valid_early))
            causal(valid_late-1) = best(valid_late-1);
            valid_late = valid_late - 1;
        elseif (best(valid_early+1)>=causal(valid_early) && best(valid_early+1)<=causal(valid_late))
            causal(valid_early+1) = best(valid_early+1);
        valid_early = valid_early + 1;
    else
        early_insert = 0; % tells if this is an early or late insert
        succeed = 0;  % set to 1 when a causal substitution is made
        depth = 2;  % the final_fuzzy_index level, depth = 1 is the best array
        while (~succeed && (~isnan(final_fuzzy_index(depth,valid_early+1)) || ...
                ~isnan(final_fuzzy_index(depth,valid_late-1))) )
            if (~isnan(final_fuzzy_index(depth,valid_late-1)))
                if (final_fuzzy_index(depth,valid_late-1)<=causal(valid_late) && ...
                        final_fuzzy_index(depth,valid_late-1)>=causal(valid_early))
                    causal(valid_late-1) = final_fuzzy_index(depth,valid_late-1);
                    valid_late = valid_late - 1;
                    succeed = 1;
                end
            else
                if (final_fuzzy_index(depth,valid_early+1)>=causal(valid_early) && ...
                        final_fuzzy_index(depth,valid_early+1)<=causal(valid_late))
                    causal(valid_early+1) = final_fuzzy_index(depth,valid_early+1);
                    valid_early = valid_early + 1;
                    succeed = 1;
                    early_insert = 1;
                end
            end
            depth = depth +1;
            if (depth > range)
                break;
            end
        end
        if (~succeed) % perform linear interpolation at end as a last resort after all possible alignments exhausted
            causal(valid_late-1) = causal(valid_late) - ...
                round((causal(valid_late)-causal(valid_early))/(valid_late-valid_early));
            valid_late = valid_late - 1;
        else % see if interpolation yields better RMSE than the alignment from the final_fuzzy_index
            if (early_insert)
                interp_value = causal(valid_early-1) + ...
                    round((causal(valid_late)-causal(valid_early-1))/(valid_late-(valid_early-1)));
                if (abs(causal(valid_early)-best(valid_early)) > abs(interp_value-best(valid_early)))
                    causal(valid_early) = interp_value;
                end
            else % late insert
                interp_value = causal(valid_late+1) - ...
                    round((causal(valid_late+1)-causal(valid_early))/(valid_late+1-valid_early));
                if (abs(causal(valid_late)-best(valid_late)) > abs(interp_value-best(valid_late)))
                    causal(valid_late) = interp_value;
                end
```

```
                        end
                    end
                end
            end
        end

        % Case 1, always fill from the end
    elseif (hole_start == 1 && hole_stop < num_frames)
        hole_pts = hole_stop-hole_start+1;  % number of points in this hole segment
        valid_late = hole_stop+1; % last valid causal point after time segment
        for k = 1:hole_pts
            % try to fill from the end
            if (best(valid_late-1)<=causal(valid_late))
                causal(valid_late-1) = best(valid_late-1);
                valid_late = valid_late - 1;
            else
                succeed = 0;  % set to 1 when a causal substitution is made
                depth = 2;  % the final_fuzzy_index level, depth = 1 is the best array
                while (~succeed && ~isnan(final_fuzzy_index(depth,valid_late-1)))
                    if (final_fuzzy_index(depth,valid_late-1)<=causal(valid_late))
                        causal(valid_late-1) = final_fuzzy_index(depth,valid_late-1);
                        valid_late = valid_late - 1;
                        succeed = 1;
                    end
                    depth = depth +1;
                    if (depth > range)
                        break;
                    end
                end
                if(~succeed) % extend last good alignment
                    causal(valid_late-1) = causal(valid_late);
                    valid_late = valid_late - 1;
                else % see if extension yields better RMSE than the alignment from the final_fuzzy_index
                    interp_value = causal(valid_late+1);
                    if (abs(causal(valid_late)-best(valid_late)) > abs(interp_value-best(valid_late)))
                        causal(valid_late) = interp_value;
                    end
                end
            end
        end

        % Case 2, always fill from the beginning
    elseif (hole_start > 1 && hole_stop == num_frames)
        hole_pts = hole_stop-hole_start+1;  % number of points in this hole segment
        valid_early = hole_start-1;  % last valid causal point before time segment
        for k = 1:hole_pts
            % try to fill from the beginning
            if (best(valid_early+1)>=causal(valid_early))
                causal(valid_early+1) = best(valid_early+1);
                valid_early = valid_early + 1;
            else
                succeed = 0;
                depth = 2;
                while (~succeed && ~isnan(final_fuzzy_index(depth,valid_early+1)))
                    if (final_fuzzy_index(depth,valid_early+1)>=causal(valid_early))
                        causal(valid_early+1) = final_fuzzy_index(depth,valid_early+1);
                        valid_early = valid_early + 1;
                        succeed = 1;
                    end
                    depth = depth +1;
                    if (depth > range)
                        break;
                    end
                end
                if (~succeed) % extend last good alignment
                    causal(valid_early+1) = causal(valid_early);
                    valid_early = valid_early + 1;
                else % see if extension yields better RMSE than the alignment from the final_fuzzy_index
                    interp_value = causal(valid_early-1);
                    if (abs(causal(valid_early)-best(valid_early)) > abs(interp_value-best(valid_early)))
                        causal(valid_early) = interp_value;
                    end
                end
            end
        end

        % Case 4, super rare case might be possible, no valid causal segments anywhere in the entire clip
    else
        error('Warning: No valid 2-pt causal alignment segments found in clip - aborting.\n');

    end

else  % Interlaced algorithm allows for 1 field jump back in time for segment you are adding

    % Case 3, probably the most likely case
    if (hole_start > 1 && hole_stop < num_frames)

        % This code toggles from beg to end, starting at beg
        hole_pts = hole_stop-hole_start+1;  % number of points in this hole segment
        valid_early = hole_start-1;  % last valid causal point before time segment
```

```
valid_late = hole_stop+1; % last valid causal point after time segment
for k = 1:hole_pts
    causal_temp = causal(valid_late:num_frames);
    min_valid_late = min(causal_temp(causal_temp~=0));  % don't include causal=0 points
    max_valid_early = max(causal(1:valid_early)); % causal=0 points don't affect this calculation
    if (mod(k,2)==1) % try to fill from the beginning first and then from the end if that fails
        if (best(valid_early+1)>=max_valid_early-1 && best(valid_early+1)<=min_valid_late+1)
            causal(valid_early+1) = best(valid_early+1);
            valid_early = valid_early + 1;
        elseif (best(valid_late-1)<=min_valid_late+1 && best(valid_late-1)>=max_valid_early-1)
            causal(valid_late-1) = best(valid_late-1);
            valid_late = valid_late - 1;
        else % see if other possible alignments exist besides the best array
            early_insert = 0; % tells if this is an early or late insert
            succeed = 0;
            depth = 2;
            while (~succeed && (~isnan(final_fuzzy_index(depth,valid_early+1)) || ...
                    ~isnan(final_fuzzy_index(depth,valid_late-1))) )
                if (~isnan(final_fuzzy_index(depth,valid_early+1)))
                    if (final_fuzzy_index(depth,valid_early+1) >= ...
                            max_valid_early-1 && final_fuzzy_index(depth,valid_early+1)<=min_valid_late+1)
                        causal(valid_early+1) = final_fuzzy_index(depth,valid_early+1);
                        valid_early = valid_early + 1;
                        succeed = 1;
                        early_insert = 1;
                    end
                else
                    if (final_fuzzy_index(depth,valid_late-1)<=min_valid_late+1 && ...
                            final_fuzzy_index(depth,valid_late-1)>=max_valid_early-1)
                        causal(valid_late-1) = final_fuzzy_index(depth,valid_late-1);
                        valid_late = valid_late - 1;
                        succeed = 1;
                    end
                end
                depth = depth +1;
                if (depth > range)
                    break;
                end
            end
            if (~succeed) % linear interpolation at beginning as last resort after all possible alignments exhausted
                causal(valid_early+1) = causal(valid_early) + ...
                    round((causal(valid_late)-causal(valid_early))/(valid_late-valid_early));
                valid_early = valid_early + 1;
            else % see if interpolation yields better RMSE than the alignment from the final_fuzzy_index
                if (early_insert)
```

```
                    interp_value = causal(valid_early-1) + ...
                        round((causal(valid_late)-causal(valid_early-1))/(valid_late-(valid_early-1)));
                    if (abs(causal(valid_early)-best(valid_early)) > abs(interp_value-best(valid_early)))
                        causal(valid_early) = interp_value;
                    end
                else % late insert
                    interp_value = causal(valid_late+1) - ...
                        round((causal(valid_late+1)-causal(valid_early))/(valid_late+1-valid_early));
                    if (abs(causal(valid_late)-best(valid_late)) > abs(interp_value-best(valid_late)))
                        causal(valid_late) = interp_value;
                    end
                end
            end
        end
    else % try to fill from the end first and then from the beginning if that fails
        if (best(valid_late-1)<=min_valid_late+1 && best(valid_late-1)>=max_valid_early-1)
            causal(valid_late-1) = best(valid_late-1);
            valid_late = valid_late - 1;
        elseif (best(valid_early+1)>=max_valid_early-1 && best(valid_early+1)<=min_valid_late+1)
            causal(valid_early+1) = best(valid_early+1);
            valid_early = valid_early + 1;
        else
            early_insert = 0; % tells if this is an early or late insert
            succeed = 0;  % set to 1 when a causal substitution is made
            depth = 2;  % the final_fuzzy_index level, depth = 1 is the best array
            while (~succeed && (~isnan(final_fuzzy_index(depth,valid_early+1)) || ...
                    ~isnan(final_fuzzy_index(depth,valid_late-1))) )
                if (~isnan(final_fuzzy_index(depth,valid_late-1)))
                    if (final_fuzzy_index(depth,valid_late-1)<=min_valid_late+1 && ...
                            final_fuzzy_index(depth,valid_late-1)>=max_valid_early-1)
                        causal(valid_late-1) = final_fuzzy_index(depth,valid_late-1);
                        valid_late = valid_late - 1;
                        succeed = 1;
                    end
                else
                    if (final_fuzzy_index(depth,valid_early+1)>=max_valid_early-1 && ...
                            final_fuzzy_index(depth,valid_early+1)<=min_valid_late+1)
                        causal(valid_early+1) = final_fuzzy_index(depth,valid_early+1);
                        valid_early = valid_early + 1;
                        succeed = 1;
                        early_insert = 1;
                    end
                end
                depth = depth +1;
                if (depth > range)
```

```
                        break;
                    end
                end
                if (~succeed) % perform linear interpolation at end
                    causal(valid_late-1) = causal(valid_late) - ...
                        round((causal(valid_late)-causal(valid_early))/(valid_late-valid_early));
                    valid_late = valid_late - 1;
                else % see if interpolation yields better RMSE than the alignment from the final_fuzzy_index
                    if (early_insert)
                        interp_value = causal(valid_early-1) + ...
                            round((causal(valid_late)-causal(valid_early-1))/(valid_late-(valid_early-1)));
                        if (abs(causal(valid_early)-best(valid_early)) > abs(interp_value-best(valid_early)))
                            causal(valid_early) = interp_value;
                        end
                    else % late insert
                        interp_value = causal(valid_late+1) - ...
                            round((causal(valid_late+1)-causal(valid_early))/(valid_late+1-valid_early));
                        if (abs(causal(valid_late)-best(valid_late)) > abs(interp_value-best(valid_late)))
                            causal(valid_late) = interp_value;
                        end
                    end
                end
            end
        end
    end

    % Case 1, always fill from the end
elseif (hole_start == 1 && hole_stop < num_frames)
    hole_pts = hole_stop-hole_start+1;  % number of points in this hole segment
    valid_late = hole_stop+1; % last valid causal point after time segment
    for k = 1:hole_pts
        causal_temp = causal(valid_late:num_frames);
        min_valid_late = min(causal_temp(causal_temp~=0));  % don't include causal=0 points
        % try to fill from the end
        if (best(valid_late-1) <= min_valid_late+1)
            causal(valid_late-1) = best(valid_late-1);
            valid_late = valid_late - 1;
        else
            succeed = 0;  % set to 1 when a causal substitution is made
            depth = 2;  % the final_fuzzy_index level, depth = 1 is the best array
            while (~succeed && ~isnan(final_fuzzy_index(depth,valid_late-1)))
                if (final_fuzzy_index(depth,valid_late-1)<=min_valid_late+1)
                    causal(valid_late-1) = final_fuzzy_index(depth,valid_late-1);
                    valid_late = valid_late - 1;
                    succeed = 1;
                end
                depth = depth +1;
                if (depth > range)
                    break;
                end
            end
            if (~succeed) % extend last good alignment
                causal(valid_late-1) = causal(valid_late);
                valid_late = valid_late - 1;
            else % see if extension yields better RMSE than the alignment from the final_fuzzy_index
                interp_value = causal(valid_late+1);
                if (abs(causal(valid_late)-best(valid_late)) > abs(interp_value-best(valid_late)))
                    causal(valid_late) = interp_value;
                end
            end
        end
    end

    % Case 2, always fill from the beginning
elseif (hole_start > 1 && hole_stop == num_frames)
    hole_pts = hole_stop-hole_start+1;  % number of points in this hole segment
    valid_early = hole_start-1;  % last valid causal point before time segment
    for k = 1:hole_pts
        max_valid_early = max(causal(1:valid_early));  % causal=0 points don't affect this calculation
        % try to fill from the beginning
        if (best(valid_early+1) >= max_valid_early-1)
            causal(valid_early+1) = best(valid_early+1);
            valid_early = valid_early + 1;
        else
            succeed = 0;
            depth = 2;
            while (~succeed && ~isnan(final_fuzzy_index(depth,valid_early+1)))
                if (final_fuzzy_index(depth,valid_early+1)>=max_valid_early-1)
                    causal(valid_early+1) = final_fuzzy_index(depth,valid_early+1);
                    valid_early = valid_early + 1;
                    succeed = 1;
                end
                depth = depth +1;
                if (depth > range)
                    break;
                end
            end
            if (~succeed) % extend last good alignment
                causal(valid_early+1) = causal(valid_early);
                valid_early = valid_early + 1;
```

```
                    else % see if extension yields better RMSE than the alignment from the final_fuzzy_index
                        interp_value = causal(valid_early-1);
                        if (abs(causal(valid_early)-best(valid_early)) > abs(interp_value-best(valid_early)))
                            causal(valid_early) = interp_value;
                        end
                    end
                end
            end

            % Case 4, very rare case might be possible, no valid 2-pt causal segments anywhere in the entire clip
        else
            error('Warning: No valid 2-pt causal alignment segments found in clip - aborting.\n');

        end

    end

    % Update the holes that are remaining
    holes = find(causal==0);

end

% Compute the average alignment RMSE between the causal alignment and the
% minimum MSE alignment, this is return argument number 2.
results_rmse = sqrt(mean((causal-best).^2));

if (verbose)

    % Plot final causal alignment
    figure(2)
    plot(best,'LineWidth',1.25)
    hold on
    set(gca,'LineWidth',2)
    set(gca,'FontName','Ariel')
    set(gca,'fontsize',12)
    plot(causal,'r','LineWidth',3)
    xlabel('Processed Frame or Field');
    ylabel('Original Frame or Field');
    title('Processed Clip Alignment Results');
    grid on
    hold off;
    pause(0.01);

    %  Print out results for verbose mode.
    fprintf('results_rmse %f\n',results_rmse);
    pause(5);

end

% Compute the causal index and its MSE and put into an array for later use
causal_index = zeros(1,num_frames);
causal_mse = zeros(1,num_frames);
for j = 1:num_frames
    this_mse = mse(:,j);
    this_index = orig_index(:,j);
    causal_index(j) = find(this_index == causal(j));
    causal_mse(j) = this_mse(causal_index(j));
end

if (verbose)

    % Plot the 3d correlation function, x = proc frame, y = orig frame, z =
    % mse, where x is the second index and the y is the first index
    orig_last = max(max(orig_index));  % last orig frame index that has data
    [x, y] = meshgrid(1:num_frames, 1:orig_last);
    [ysize, xsize] = size(x);
    z = NaN(ysize, xsize);  % missing values will be NaN
    for j = 1:num_frames  % fill column by column
        for i = 1:range  % row element
            z(orig_index(i,j),j) = mse(i,j);
        end
    end

    figure(3)
    mesh(x,y,z, 'EdgeColor', 'black');
    hold on
    hidden on
    set(gca,'LineWidth',2)
    set(gca,'FontName','Ariel')
    set(gca,'fontsize',12)
    xlabel('Processed Frame or Field');
    ylabel('Original Frame or Field');
    zlabel('MSE');
    title('3D MSE Function');
    shading interp

    % Now overlay the causal function MSE using a red line
    x_causal = 1:num_frames;
    y_causal = causal;
    z_causal = causal_mse;
    warning off
    alpha(0.9);  % Sets some face transparency to see points
    scatter3(x_causal,y_causal,z_causal,'r','filled');
    hold off
    pause(5);  % pause 5 sec to display plots properly

end

% Assign results array as requested by the user
if (~is_causal)

    results = best;
    results_fuzzy = final_fuzzy_index;
    results_fuzzy_mse = final_fuzzy_mse;

else % further processing is required for fuzzy alignments

    results = causal;

    %%%%%%%%%%
    % Stage 6
    % Expand the fuzzy alignments to include the causal alignments.
    % Some of this code is replicated from stage 1.
    %%%%%%%%%%

    % Step thru the frames for this clip and find the causal fuzzy
    % alignments.  The causal fuzzy alignments expands the set of final
    % fuzzy alignments found previously to include the causal alignment
    % point and all points in-between this point and the prior set of final
    % fuzzy alignments.
    final_fuzzy_causal_index = NaN(range, num_frames); % The final fuzzy causal alignment from most to least likely
    final_fuzzy_causal_mse = NaN(range, num_frames); % The corresponding MSEs
    for j = 1:num_frames

%         % Plot the correlation function
%         figure(1)
%         plot(orig_index(:,j),mse(:,j),'LineWidth',2);
%         hold on
%         grid on
%         set(gca,'LineWidth',2)
%         set(gca,'FontName','Ariel')
%         set(gca,'fontsize',12)
%         xlabel('Original Frame or Field');
%         ylabel('Mean Squared Error (MSE)');
%         this_title = ['Processed Frame or Field ' int2str(j)];
%         title(this_title);
%         hold off

        % Since subtractive correlation is being used, the correlation function
        % will be a minimum at the best match and increase from there.  This
        % code uses a thresholding scheme that sets a threshold at the minimum
        % correlation value + thres * maximum correlation value, but the
        % maximum correlation value used is the top (95%) rank sorted value
        % for robustness.  All alignments within this threshold from the best
        % correlation value are considered possible alignments for later
        % processing by the algorithm.
        this_mse = mse(:,j);
        this_mse_valid = find(~isnan(this_mse));  % array can contain NaN if orig frames did not exist
        range_valid = length(this_mse_valid);
        this_mse_sort = sort(this_mse);
        this_index = orig_index(:,j);
        this_index_sort = orig_index_sort(:,j);

        max_corr = this_mse_sort(floor(top*range_valid));
        min_corr = this_mse_sort(1);
        fuzzy = find(this_mse >= min_corr & this_mse <= min_corr+thres*max_corr);
        num_fuzzy = length(fuzzy);  % will always be at least one
        fuzzy_sort = [];
        for k = 1:num_fuzzy
            fuzzy_sort(k) = find(this_index_sort == this_index(fuzzy(k)));
        end

%         % Add the fuzzy alignments to the plot, red points meet threshold
%         figure(1)
%         hold on
%         plot(this_index(fuzzy),this_mse(fuzzy),'r.','MarkerSize',15);
%         hold off

        % Expand the fuzzy alignments to include the range of original indices
        % that are covered by the minimum MSE points.  This will include
        % extra points that were not in the minimum MSE fuzzy set.
        bigger_fuzzy = (fuzzy(1):fuzzy(length(fuzzy)))';

        % Added fuzzy points that were not included in the original set
        added_fuzzy = setxor(fuzzy,bigger_fuzzy); % just the added points, for plotting
        added_fuzzy_mse = this_mse(added_fuzzy);
        added_fuzzy_index = this_index(added_fuzzy);
        num_added_fuzzy = length(added_fuzzy);
        added_fuzzy_sort = [];
        for k = 1:num_added_fuzzy
            added_fuzzy_sort(k) = find(this_index_sort == this_index(added_fuzzy(k)));
        end

%         % Add the expanded fuzzy points to the plots in green squares.
%         figure(1)
%         hold on
%         plot(this_index(added_fuzzy),this_mse(added_fuzzy),'gs','MarkerSize',5,'MarkerFaceColor','g','MarkerEdgeColor','g');
%         hold off

%         % Overlay the causal point with a large black diamond
%         figure(1)
%         hold on
%         plot(this_index(causal_index(j)),causal_mse(j),'kd','MarkerSize',10,'LineWidth',1.25);
%         hold off

        % Now expand the fuzzy set to include the causal point
        added_causal = [];
        added_causal_mse = [];
        cplot = 0;
        if (this_index(causal_index(j)) < this_index(fuzzy(1))) % Must expand on left
            added_causal = (causal_index(j):(fuzzy(1)-1))';
            added_causal_mse = this_mse(added_causal);
            bigger_fuzzy = (causal_index(j):fuzzy(length(fuzzy)))';
            cplot = 1;
        elseif (this_index(causal_index(j)) > this_index(fuzzy(length(fuzzy)))) % Must expand on right
            added_causal = ((fuzzy(length(fuzzy))+1):causal_index(j))';
            added_causal_mse = this_mse(added_causal);
            bigger_fuzzy = (fuzzy(1):causal_index(j))';
            cplot = 1;
        end

%         % Overlay the added causal points in solid black diamonds
%         if(cplot)
%             figure(1)
%             hold on
%             plot(this_index(added_causal),added_causal_mse,'kd','MarkerSize',5,'MarkerFaceColor','k','MarkerEdgeColor','k');
%             hold off
%         end

        % Sort the bigger fuzzy information from the most likely alignment
        % to the least likely alignment, based on MSE.  Then move the causal
        % alignment point to the first element if it is not there already.
        % The array final_fuzzy_causal_index gives the fuzzy alignments
        % with causal processing.
        bigger_fuzzy_mse = this_mse(bigger_fuzzy);
        bigger_fuzzy_index = this_index(bigger_fuzzy);
        [bigger_fuzzy_mse_sort mse_order] = sort(bigger_fuzzy_mse);
        bigger_fuzzy_index_sort = bigger_fuzzy_index(mse_order);
        tlen = length(bigger_fuzzy_index_sort);
        tind = find(bigger_fuzzy_index_sort == this_index(causal_index(j)));
        if (tind ~= 1)  % reorder so causal alignment is first
            if (tind ~= tlen)
                bigger_fuzzy_index_sort = cat(1, bigger_fuzzy_index_sort(tind), bigger_fuzzy_index_sort(1:tind-1), ...
                    bigger_fuzzy_index_sort(tind+1:tlen));
                bigger_fuzzy_mse_sort = cat(1, bigger_fuzzy_mse_sort(tind), bigger_fuzzy_mse_sort(1:tind-1), ...
                    bigger_fuzzy_mse_sort(tind+1:tlen));
            else
                bigger_fuzzy_index_sort = cat(1, bigger_fuzzy_index_sort(tind), bigger_fuzzy_index_sort(1:tind-1));
                bigger_fuzzy_mse_sort = cat(1, bigger_fuzzy_mse_sort(tind), bigger_fuzzy_mse_sort(1:tind-1));
            end
        end
        final_fuzzy_causal_index(1:tlen,j) = bigger_fuzzy_index_sort;
        final_fuzzy_causal_mse(1:tlen,j) = bigger_fuzzy_mse_sort;

    end

    % Assign the fuzzy results for a causal user request
    results_fuzzy = final_fuzzy_causal_index;
    results_fuzzy_mse = final_fuzzy_causal_mse;

end

return;

end
```

FORM **NTIA-29** U.S. DEPARTMENT OF COMMERCE
(4-80) NAT'L. TELECOMMUNICATIONS AND INFORMATION ADMINISTRATION

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION NO. TM-10-463	2. Government Accession No.	3. Recipient's Accession No.
4. TITLE AND SUBTITLE A Full Reference (FR) Method using Causality Processing for Estimating Variable Video Delays		5. Publication Date October 2009 6. Performing Organization NTIA/ITS.T
7. AUTHOR(S) Stephen Wolf		9. Project/Task/Work Unit No. 3141011
8. PERFORMING ORGANIZATION NAME AND ADDRESS Institute for Telecommunication Sciences National Telecommunications & Information Administration U.S. Department of Commerce 325 Broadway Boulder, CO 80305		10. Contract/Grant No.
11. Sponsoring Organization Name and Address National Telecommunications & Information Administration Herbert C. Hoover Building 14th & Constitution Ave., NW Washington, DC 20230		12. Type of Report and Period Covered
14. SUPPLEMENTARY NOTES		

15. ABSTRACT

Digital video transmission systems consisting of a video encoder, a digital transmission method (e.g., Internet Protocol – IP), and a video decoder can produce pauses in the video presentation, after which the video may continue with or without skipping video frames. This time varying delay of the output (or processed) video frames can present a challenge for some video quality measurement systems. The reason is that time alignment errors between the output video sequence and the input (or reference) video sequence may produce measurement errors for full reference measurements like Peak Signal to Noise Ratio (PSNR) that greatly exceed the perceptual impact of the time varying video delays. This document presents a Full Reference (FR) method for estimating variable video delays. The algorithm can optionally execute a sophisticated causality processing algorithm to improve the robustness of the delay estimates. The delay estimates produced by this algorithm can be utilized by a FR quality measurement system to remove variable video delay as a calibration step before computing the quality measurement.

16. Key Words

calibration; causality; delay; dropped frames; Full Reference (FR); pausing; skipping; video quality

17. AVAILABILITY STATEMENT ☐ UNLIMITED.	18. Security Class. (This report) Unclassified	20. Number of pages 68
	19. Security Class. (This page) Unclassified	21. Price:

NTIA FORMAL PUBLICATION SERIES

NTIA MONOGRAPH (MG)

A scholarly, professionally oriented publication dealing with state-of-the-art research or an authoritative treatment of a broad area. Expected to have long-lasting value.

NTIA SPECIAL PUBLICATION (SP)

Conference proceedings, bibliographies, selected speeches, course and instructional materials, directories, and major studies mandated by Congress.

NTIA REPORT (TR)

Important contributions to existing knowledge of less breadth than a monograph, such as results of completed projects and major activities. Subsets of this series include:

NTIA RESTRICTED REPORT (RR)

Contributions that are limited in distribution because of national security classification or Departmental constraints.

NTIA CONTRACTOR REPORT (CR)

Information generated under an NTIA contract or grant, written by the contractor, and considered an important contribution to existing knowledge.

JOINT NTIA/OTHER-AGENCY REPORT (JR)

This report receives both local NTIA and other agency review. Both agencies' logos and report series numbering appear on the cover.

NTIA SOFTWARE & DATA PRODUCTS (SD)

Software such as programs, test data, and sound/video files. This series can be used to transfer technology to U.S. industry.

NTIA HANDBOOK (HB)

Information pertaining to technical procedures, reference and data guides, and formal user's manuals that are expected to be pertinent for a long time.

NTIA TECHNICAL MEMORANDUM (TM)

Technical information typically of less breadth than an NTIA Report. The series includes data, preliminary project results, and information for a specific, limited audience.

For information about NTIA publications, contact the NTIA/ITS Technical Publications Office at 325 Broadway, Boulder, CO, 80305 Tel. (303) 497-3572 or e-mail info@its.bldrdoc.gov.

This report is for sale by the National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161,Tel. (800) 553-6847.