VIDEO RESOLUTION ENHANCEMENT VIA FRAME INTERPOLATION

by

Ashish Gehani

Department of Computer Science
Duke University

Date: _____

Approved:

_____
John Reif, Professor, Supervisor

_____
Tassos Markas, Adjunct Professor

_____
Gershon Kedem, Associate Professor

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in the Department of Computer Science
in the Graduate School of
Duke University

1997

Abstract

In this thesis we propose a novel use of the temporal similarity in frames of a video sequence. Traditionally, video compression algorithms exploit this with interframe coding. Instead, we use the information to enhance the resolution of the video sequence.

We use a standard motion estimation technique followed by an analytical method of sub-pixel motion re-estimation that we develop, on each of the blocks in a set of temporally adjacent frames, to obtain a larger data set representing each image in the video sequence. The extra information is then incorporated into the reference image in a useful form by constructing a bivariate spline surface that approximates the enlarged data set, and evaluating this at intermediate points to provide a higher resolution frame.

An application GROW, that takes a video sequence as input and produces a higher resolution sequence as output, was developed as proof of concept.

Acknowledgements

I owe much to many (in reverse chronological order):

- John Reif - for introducing me to this unique area of research.
- Paul Dierckx - whose Netlib FITPACK routines formed a basis that spanned my splining requirements.
- Laszlo Babai - for exposing me to an ideal field.
- Anjana and Bhagwan Gehani - no metric can measure the uncountably infinite number of instances when they have assisted me.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Video films are an important data format today. A wide range of fields make use of the medium to record events for further analysis than would be possible in real time. Image sequences serve as an important scientific tool. Prominent areas where videos are taken include medicine - during internal procedures such as gastroenterological endoscopies, space exploration - during a spacecraft's visit to planets, and geographical surveys - for recording topographical imagery. In our daily life, video is used for a variety of purposes from surveillance cameras used to monitor places such as banks and shopping establishments, to sports events where a referee may use video footage to assist in making difficult decisions. The ability to enhance the resolution of an image stream on a per frame basis could be tremendously useful in many such situations. For example, an astronomer may be interested in obtaining more detail than is visible from the raw video about a planet's surface, or a security officer may want more detail of a thief's face than was visible from any given frame of the original film. We seek to provide a means to achieve this efficiently.

1.2 Background

Objects in the real world exist in three dimensions, but most media used for recording and reproducing visual data represent images in two dimensions. Usually the *object space* is transformed to the *image space* using either a *parallel projection* or a *perspective projection*. Technically, a projection is defined as mapping a single point in

the *domain* to a single point in the *range*. Thus, projecting an object would require the mapping of an infinite number of points [PG89]. What occurs in practice is the projection of only a finite number of points on an object. This is an important fact in the context of the development of this thesis.

A parallel projection of a point (x, y, z) in object space, a subset of $\mathbf{R}^3$, is obtained by adding the appropriate multiple of the *projection vector* so as to obtain a point $(x, y, 0)$ in object space, which is identical to the point (x, y) in image space, a subset of $\mathbf{R}^2$. An *orthographic projection* is one in which the projection vector is normal to the *projection plane*, defined as the plane $z = 0$. This class arises when the camera is at a very great distance from the objects being filmed. (Various classes of *oblique projections*, ones where the projection vector is not parallel to the z axis, such as the *cavalier projection* and the *cabinet projection* are of importance due to the frequency of their occurrence.)

A perspective projection maps a point (x, y, z) in object space to a point $(x_p, y_p, 0)$ on the projection plane, such that this point lies on the line connecting (x, y, z) to (x_c, y_c, z_c), the *center of projection*, a fixed point assumed to be on the opposite side of the projection plane from where the object is. When a camera is close to an object the effect of perspective on a projection is introduced.

Each point that is mapped spatially also has a color attribute, which can be represented as a point in a *color space*. For example the *RGB* color space can be visualized as $\mathbf{R}^3$ with the x, y and z axes representing the intensity of red, green and blue light. Without loss of generality, we work with gray scale representations of this attribute, since the techniques developed may be applied to each of the orthogonal components of the color attribute separately to obtain a full color generalization. We use the *YUV* color space to represent the attribute color since it is most convenient to work with the Y (intensity) component of each point in the image space.

In the context of digital video, two further approximations occur during the recording process. First, the image space, $\mathbf{R}^2$ is approximated with a *representation grid* in $\mathbf{N}^2$, which we call the *approximated image space*. Thus, regions in $\mathbf{R}^2$ are represented by a single data point, a *pixel*. Second, the intensity of a pixel is not recorded with arbitrary precision, but as a level in a fixed range in $\mathbf{N}$. The first approximation is of particular import to the discussion.

By repeating the spatial registration process at discrete points in time, a *video sequence* is formed. Recording the instant in time with a temporal index is accomplished by a strictly increasing monotonic mapping from $\mathbf{R}$ to $\mathbf{N}$.

Thus, through a process of sampling at a finite number of points in time and space, the video coding used has been defined as the representation in $\mathbf{N}^2 \times \mathbf{N} \times \mathbf{N}$ (image space $\times$ pixel intensity $\times$ instant in time) of $\mathbf{R}^3 \times \mathbf{R}^3 \times \mathbf{R}$ (object space $\times$ color space $\times$ instant in time).

Finally, we note that as an object moves, the set of points on it that undergoes spatial projection will vary from *frame* to frame, where a frame is defined as a complete representation of the approximated image space at a given instant in time. Video compression algorithms rely on the fact that this variation is small and choose to neglect it. This is effectively what happens by the process of *quantization*, (the representation of a range of pixel intensity levels, corresponding to a set of points close to each other in obect space, with a single intensity level representing that area,) and *dequantization*, the (approximately) inverse process. We seek to use the information in this variation, rather than ignore it, to obtain a better representation of the original object space.

1.3 Problem Specification

Given a video sequence, we wish to efficiently enhance the resolution of the representation grid in approximated image space, such that the result is the equivalent of having mapped a larger set of points, from the object space to the image space, than actually occurred in any given frame.

1.4 Relation to Previous Work

Identifying points in object space using image sequences has been dealt with extensively in the literature on computer vision, geometric modeling, image processing, and pattern recognition. Work has even been done on tracking objects with a moving camera [BBH+89]. To deal with the problems of discerning rotation from translation when the motion is small, as well the sensitivity of the measurements of depth (to determine object shape) to noise, factorization methods that analytically separate the motion from the object have been developed for the cases where the motion is planar [TK90], or in $\mathbf{R}^3$ [TK91], as well as when the projection is either orthographic [TK92a, TK92b], under perspective [Tom93] or the paraperspective case [PK]. The analyses have been carried out within a framework where either there exists a fixed model of camera or object motion, or the study seeks to identify the parameters [WC92] that constitute the mapping $\mathcal{M}$, from object space to image space.

Applying past methods would involve identifying $\mathcal{M}$ explicitly and computing $\mathcal{M}^{-1}$, or at the very least determining $\mathcal{M}^{-1}$ empirically, in order to capture representations in object space, that could then be transformed to image space, to increase the information available, which in turn would allow a higher resolution frame to be constructed. However, we seek to develop an *efficient algorithm for video resolution enhancement*, by which we mean that we wish to obviate the need for explicit map-

pings back to object space. Our algorithm will achieve the extraction of extra data points in any given frame by working directly with the frame's image space, and the associated *temporal context*, the past and future frames.

1.5 Approach

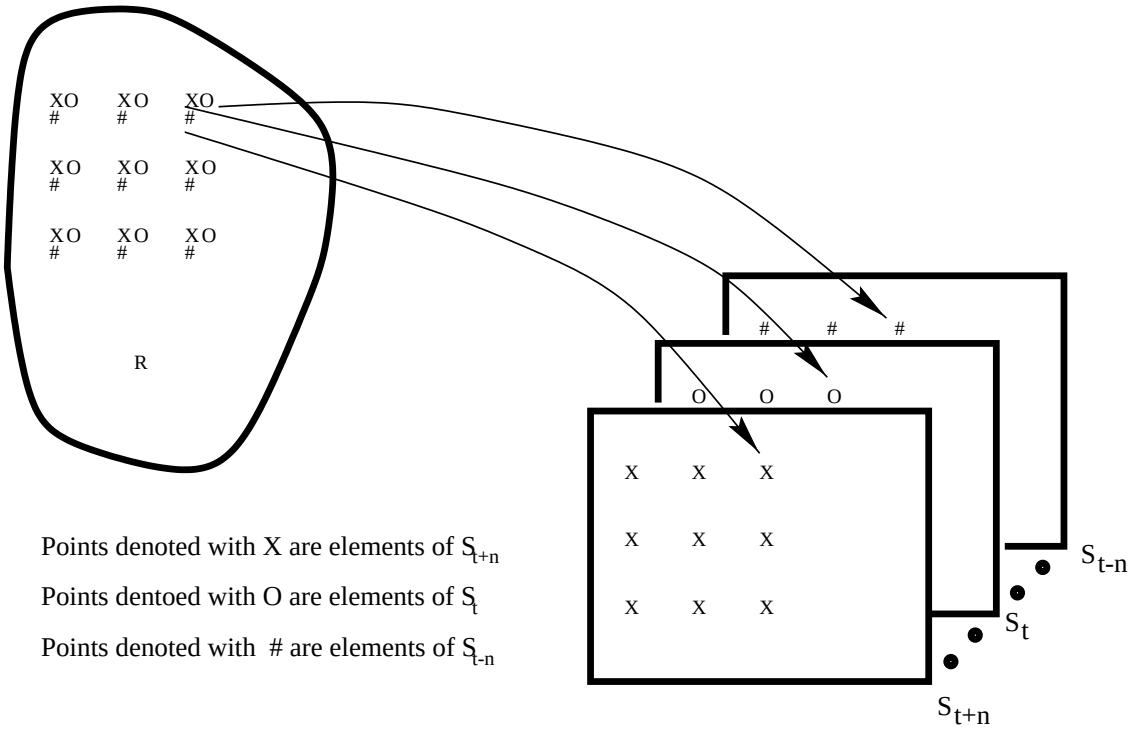


Figure 1.1: The image sequence that records the region R, registers similar but not identical sets of points in object space.

Initially, we note that by approaching the problem using only spatial context, we can obtain only guesses of the the intensity values that should be assigned to the newly defined points (that result from the higher resolution we desire) in the approximated image space, not data that is known to be the result of a mapping from the object space. Numerous methods exist to do this such as *linear interpolation*, *natural spline interpolation*, or the application of a *Gaussian filter* to a region of points around the one whose intensity is being estimated, for each of the new points.

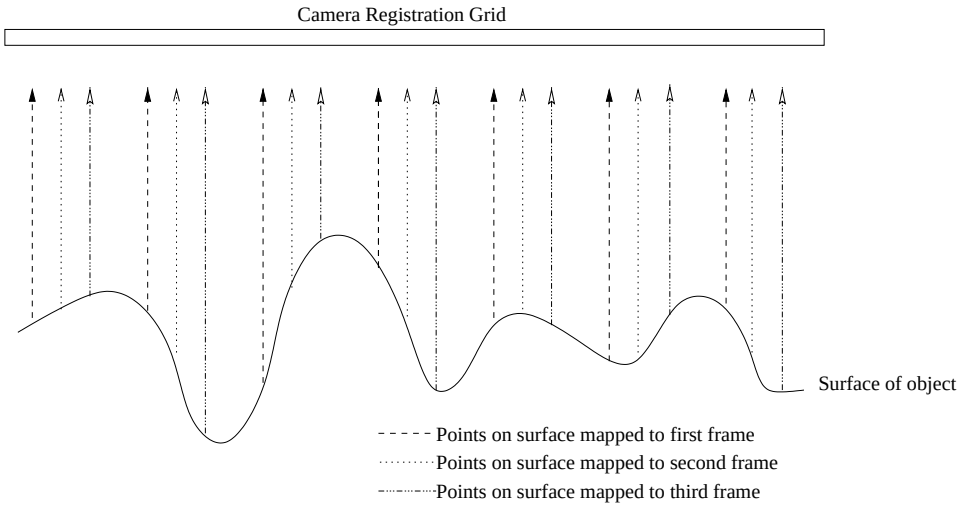


Figure 1.2: We assume that the camera samples a different set of points on the object in each frame.

Instead we use the insight that in consecutive frames, $f_t, \ldots, f_{t+n}$, of the video sequence, the same region R in the object space is captured during the recording process, by mapping different sets of points, $S_t, \ldots, S_{t+n}$, of the object space onto the image space. Each frame is the discrete sampling of the object with the filter being a characteristic of the camera that is being used. Thus the sets $S_t, \ldots, S_{t+n}$, are very close approximations of each other, but not identical, since the nature of analogue to digital conversion is that of a *many to one* mapping of an *uncountable set*, (one that is not countable,) onto a *countable set*, (one that is empty, finite or can be placed in one to one correspondence with $\mathbf{N}$). In this case, the object space contains a set of points with a cardinality that is large enough that it is reasonable to consider the set uncountable. For any given video frame, we will seek to exploit the above by retrieving information about the region in the object space R, corresponding to the image space I_t under consideration, from representations of the same region R in previous and future frames, $I_{t-n}, \ldots, I_{t-1}, I_{t+1}, \ldots, I_{t+n}$. With a larger set of points mapped from the object space to the image space, we will be in a position to create a higher resolution video frame.

1.6 Algorithm

Each frame of the video sequence is enhanced independently in serial order. To process a given frame, first a set of frames is formed by choosing several frames that are immediately previous and several that are immediately after the frame under consideration. The frame that is being improved is then broken up into rectangular blocks. Extra data about each block in the frame is then extracted. For a given block, this is done by performing a search for one block from each of the frames in the above defined set that best matches it. A sub-pixel displacement that will minimize the error between each of the contextual blocks and the block under consideration is then analytically calculated for each of the blocks in the frame. If the data thus obtained is viewed as a set of points of the form (x, y, z), where z is the image intensity at a point (x, y) in the XY plane, then the result is data of the form of a set of intensities over a scattered set of points in the XY plane. A bivariate spline surface, composed of tensor product basis splines, that approximates this data is computed and evaluated on a fine grid, to produce the increased resolution frame. Since this is done for each frame in the sequence, except the first and last few which do not have enough context, a sequence of similar length is produced.

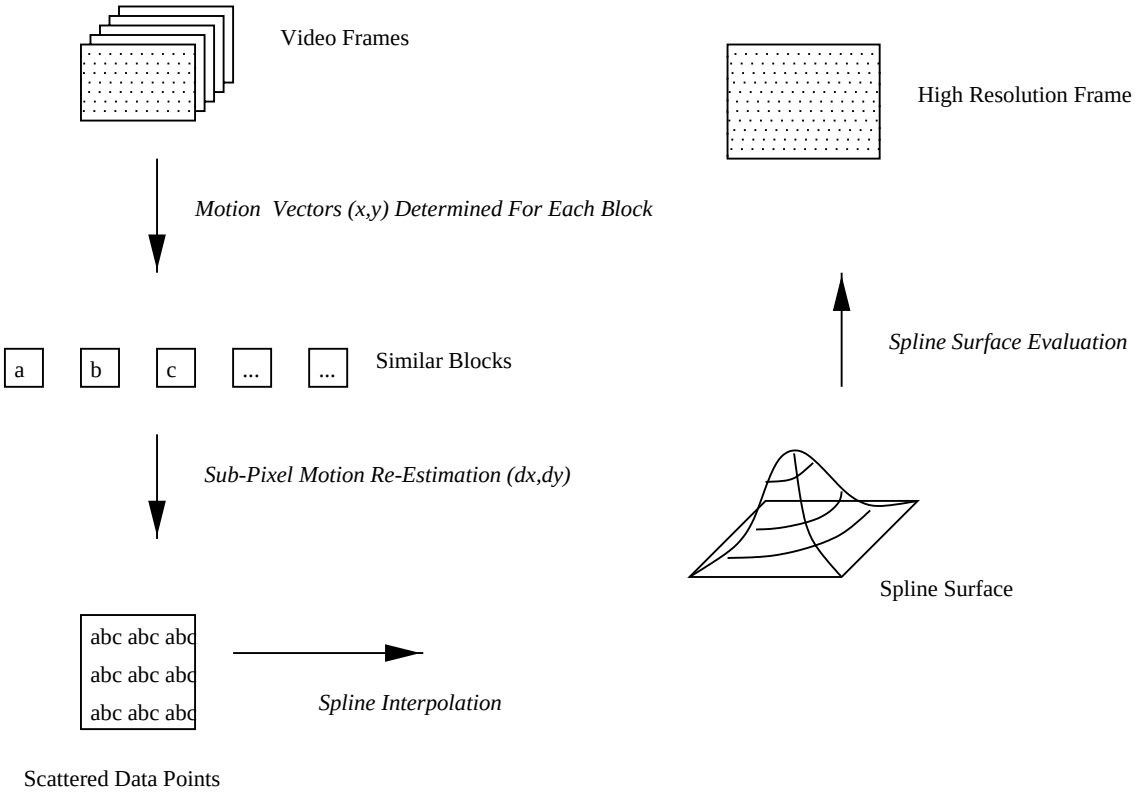


Figure 1.3: Overview of algorithm.

Chapter 2

Extraction of Temporal Context

The algorithm that we develop enhances the resolution of the video sequence by increasing the size of the data set associated with each frame. Considering one *reference frame*, whose resolution we are currently improving, without loss of generality we will consider the problem of extracting information of relevance from only one other frame, the *contextual frame*. The same process may be applied to a set of other frames immediately before and after the reference frame in the video sequence in order to obtain the increased size data set representing the region in object space that was mapped to the reference frame.

Ideally, whenever an object in the reference frame appears in the contextual frame, we would like to be able to identify the exact set of pixels that correspond to the object, extract the set and incorporate it into the representation of the object in the reference frame. To effect this search, we use a technique from video compression, called *motion estimation*.

2.1 Initial Motion Estimation

2.1.1 Framework

Video compression achieves higher compression ratios than still image *intraframe* coding, the process of reducing spatial redundancy in a frame, by hybridization with *interframe* coding, which exploits temporal correlation of data. This is done by compensating for the motion of objects, from frame to frame, by identifying an object, estimating its motion, and then encoding the movement as a vector representing an *affine transformation*. Currently research on computing optical flow that would yield

such representations exists, but for computationally efficient solutions [BK96] that give a good approximation, standards such as MPEG [CLL+95] and H.261 have adopted a different approach.

In the general case, we would allow for a range of motion by the object, taking into account scaling, rotation, and warping in addition to translation. The implication of this is that the geometry of the domain and range of the motion mapping need not be the same. This leads to complex geometrical concerns as well as intensive computational requirements. Standards such as MPEG deal with this by arbitrarily decomposing a frame into blocks and working at the granularity of blocks [Gal91] with the assumption that there is no rotation occurring. This is the equivalent of imposing a uniform motion vector field on all the pixels in the block. To ensure the correctness of the algorithm, a means of encoding the motion of the pixels that do not have the same motion as the block is introduced in the form of a *residual code*, which is the difference of the actual representation and that imposed on the pixel by the block granularity motion vector.

The reference frame is broken into blocks of fixed size. (The MPEG standard specifies the use of 16×16 size blocks [Chi95]. We leave the block dimensions as a parameter of the algorithm to determine the best dimensions for our purpose.) The size of the block must be determined as a function of several factors that necessitate a tradeoff:

- By using smaller blocks, the finer granularity allows for local movements to be better represented. The uniform motion field is forced on a smaller number of pixels.
- The disadvantage is the increase in computational requirements.
- Depending on the matching criterion, smaller blocks may also result in more

incorrect matches, since bigger blocks provide more context increasing the probability of correctly determining an object's motion vector.

For each of these blocks, a search for a block of identical dimensions is performed in the contextual frame. A metric that associates cost inversely with the closeness of the match is used to identify the block with the lowest cost. The vector connecting a point on the reference block to the corresponding point of the best match block found in the contextual frame, is determined as the initial motion estimate for that block.

2.1.2 Matching Criteria

If a given video frame is of width F_w and height F_h, and the sequence is T frames long, then a pixel at position (m, n) in the f^{th} frame, is specified by $VS(m, n, f)$, where $m \in \{1, \ldots, F_w\}$, $n \in \{1, \ldots, F_h\}$, and $f \in \{1, \ldots, T\}$. If a given block is of width B_w and height B_h, then a block in the reference frame with its top left pixel at (x, y) is denoted $RB(x, y, t)$, where $x \in \{1, \ldots, (F_w - B_w + 1)\}$, $y \in \{1, \ldots, (F_h - B_h + 1)\}$, and t is the temporal index. For a fixed reference frame block denoted by $RB(x, y, t)$, the block in the contextual frame that has its top left pixel located at the position $(x + a, y + b)$ is denoted by $CB(a, b, u)$, where $(x + a) \in \{1, \ldots, (F_w - B_w + 1)\}$, $(y + b) \in \{1, \ldots, (F_h - B_h + 1)\}$, and where the contextual frame has temporal index $(t + u)$. The determination that the object, represented in the block with its top left corner at (x, y) at time t, has been translated in exactly u frames by the vector (a, b), is associated with a cost given by the function $\mathcal{C}(RB(x, y, t), CB(a, b, u))$.

We chose to use *Mean Absolute Error (MAE)*, or *Mean Absolute Difference (MAD)* as the cost function on the basis of empirical studies in the literature on MPEG standards [Mar93, BK96] that have shown it works as effectively as more computationally

intensive criteria, where

$$MAD(RB(x,y,t),CB(a,b,u)) = \tag{2.1}$$
$$\frac{1}{B_w B_h}\sum_{i=0}^{(B_w-1)}\sum_{j=0}^{(B_h-1)}|VS(x+i,y+j,t)-VS(x+a+i,y+b+j,t+u)|$$

is the specific cost function.

Other valid heuristics include computing the *Mean Square Error (MSE)*, where

$$MSE(RB(x,y,t),CB(a,b,u)) = \tag{2.2}$$
$$\frac{1}{B_w B_h}\sum_{i=0}^{(B_w-1)}\sum_{j=0}^{(B_h-1)}[VS(x+i,y+j,t)-VS(x+b+i,y+b+j,t+u)]^2$$

or computing the correlation between the blocks $RB(x,y,t)$ and $CB(a,b,u)$.

2.1.3 Algorithm for Finding the Motion Vector

2.1.3.1 Vector Determination Criterion

Given a reference block $RB(x,y,t)$, we wish to determine the block $CB(a,b,u)$, that is the best match possible in the $(t+u)^{th}$ frame, with the motion vector (a,b) constrained to a fixed rectangular search space centered at (x,y). If this space's dimensions are specified as the parameters p and q of the algorithm, then $a \in \{(x-p),\ldots,(x+p)\}$, and $b \in \{(y-q),\ldots,(y+q)\}$ must hold. For sequences with little motion these values can be made small for computational efficiency. If the motion characteristics of the sequence are unknown, p and q may be set so that the entire frame is searched.

2.1.3.2 Full Search

The only method that can determine the best choice $CB(a_{min},b_{min},u)$ with probability one, such that

$$\forall a \in \{(-x+1),\ldots,(F_w-B_w-x+1)\}, \tag{2.3}$$

$$\forall b \in \{(-y+1), \ldots, (F_h - B_h - y + 1)\},$$
$$\mathcal{C}(RB(x,y,t), CB(a_{min}, b_{min}, u)) \leq \mathcal{C}(RB(x,y,t), CB(a,b,u))$$

is *full search* [WWY], which calculates $\mathcal{C}(RB(x,y,t), CB(a,b,u))$ for all values of a and b in the above mentioned ranges, and keeps track of the pair (a,b) that has resulted in the lowest value of the cost function. This algorithm has complexity $\mathcal{O}(pqB_wB_h)$ if p and q are specified, and $\mathcal{O}(F_wF_hB_wB_h)$ if the most general form is used.

2.1.3.3 Alternatives

For general purpose video compression, numerous other algorithms have been developed that can reduce the complexity of the search. We outline some of them below.

Two-dimensional logarithmic search works by computing the cost at the nine positions (a,b) where $a \in \{(x-p), x, (x+p)\}$ and $b \in \{(y-q), y, (y+q)\}$, then determining the point at which the cost function attained the minimum value. This point is then used as the center and $\frac{p}{2}$ and $\frac{q}{2}$ are used in place of p and q and the process is repeated. The recursion reduces the $\mathcal{O}(pq)$ aspect to $\mathcal{O}(\log pq)$.

Parallel hierarchical one-dimensional search (PHODS) works in a similar manner. In parallel, we find the value of $(a,y) \in \{(x-p,y), (x,y), (x+p,y)\}$, and $(x,b) \in \{(x,y-q), (x,y), (x,y+q)\}$ that yields the lowest value of the cost function. Use the determined a and b instead of x and y as the center, and repeat the process with $\frac{p}{2}$ and $\frac{q}{2}$ used instead p and q. This improves over the logarithmic search by creating regular data flow as the search points remain on the same axes, and allows parallelization of the determination of a and b.

Pixel subsampling, is the process of dividing the image into geometrically identical regions, such as square grids of pixels, and then using one representative value for

each region, such as the mean of the pixel intensities in the region. This division into regions must be constrained by certain conditions so that the sub-regions are balanced representations of the original [BK96]. Full search can be performed on the lower resolution image, yielding a constant factor improvement in the complexity bound.

This may be improved upon by recursing in this manner several times, then using the vector determined on the lowest resolution image as a starting point for searching at the next higher resolution, and repeating this till the original image is used. This too will improve the complexity only by a constant factor, since multiple searches at each of the various resolutions are needed and a fixed number of resolutions is used. This is called *hierarchical search*.

Searching by projection of the image grid values onto a line, then searching by comparing the cost function at various points on the line to determine the best choice and repeating this with different lines, will also yield a constant factor improvement over full search.

2.1.3.4 Implementation Choice

The problem with the various alternatives to full search is that they do not guarantee that the correct best match will be returned. They will be useful to us only when there is almost no similarity between the various possible blocks within a given frame, minimizing the chance of incorrect motion estimation. In the case that video compression is being performed and the motion vector is used in conjunction with a residual code, this is not a problem since the error will be compensated for. However, in our application we are searching for the specific part of image space that represents the area of object space under consideration. We would therefore prefer to use a full search algorithm with limits on p and q if there are computational constraints, since this yields a more effective strategy for our purpose. In addition, the complexity of

context extraction [PDG95] is significantly lower than that of the context integration stage of the algorithm, making it possible to use full search here without noticeably affecting the performance of the implementation.

2.2 Re-estimation With Subpixel Accuracy

Recalling the discussion of Section 1.2, the actual motion of the object that has occurred between frames has no correspondence with the sampling grid (that is, the object does not move in integral pixel displacements). As a result we wish to estimate the motion vector at a finer resolution. Therefore, effectively the initial estimate as outlined above is the first part of a two stage hierarchical search. We

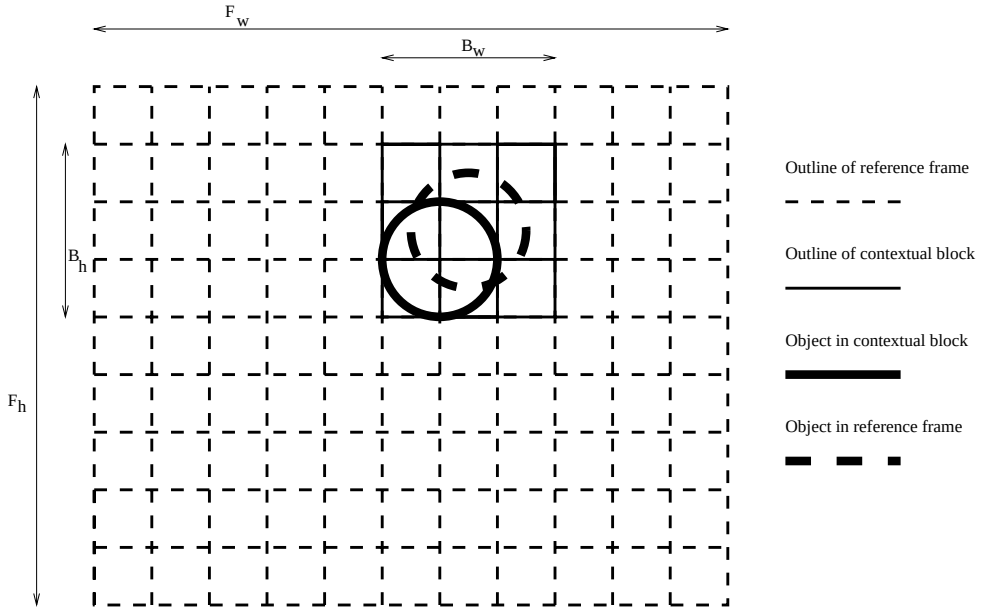


Figure 2.1: During the first stage, the contextual block containing the object in the reference block is determined with integral pixel accuracy.

have developed an analytical technique to estimate the motion of a block with subpixel accuracy. The implementation of this completes the second part. By modeling the cost function, MSE, as a separable function, we develop a measure for what real valued displacements along each orthogonal axis will reduce the value of the mean

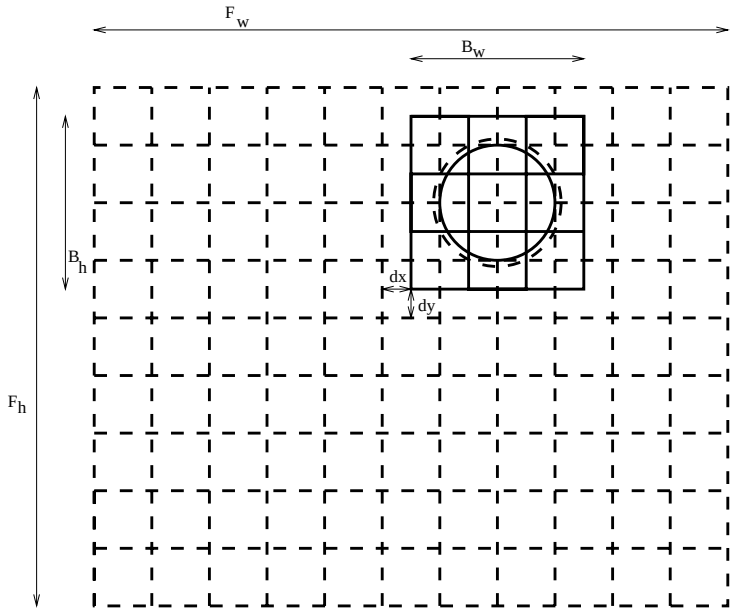


Figure 2.2: The second stage determines a sub-pixel displacement of the contextual block such that there is an improved match with the reference area.

absolute error.

Let the function $\mathcal{F}(x, y)$ contain a representation of the reference frame with temporal index t, that is

$$\forall x \in \{1, \ldots, F_w\}, \forall y \in \{1, \ldots, F_h\} \tag{2.4}$$
$$\mathcal{F}(x, y) = VS(x, y, t)$$

and let the function $\mathcal{G}(x, y)$ contain a representation of the contextual block for which we are determining the motion vector with sub-pixel accuracy. If $RB(x_0, y_0, t)$ was the reference block for which the previous stage of the algorithm found $CB(a_{min}, b_{min}, u)$ to be the contextual block with the closest match in frame $(t + u)$, then define

$$x_{min} = x_0 + a_{min},\ x_{max} = x_0 + a_{min} + B_w - 1 \tag{2.5}$$
$$y_{min} = y_0 + b_{min},\ y_{max} = y_0 + b_{min} + B_h - 1$$

and further note that the appropriate definition of $\mathcal{G}(x, y)$ that is required is given

by

$$\forall x \in \{x_{min}, \ldots, x_{max}\}, \forall y \in \{y_{min}, \ldots, y_{max}\} \qquad (2.6)$$
$$\mathcal{G}(x, y) = VS(x, y, u)$$

Instead of (a_{min}, b_{min}), which is the estimate that results from the first stage of the search, we seek to determine the actual motion of the object, which can be represented by $(a_{min}+\delta a_{min}, b_{min}+\delta b_{min})$. Therefore, at this stage the vector $(\delta a_{min}, \delta b_{min})$, which we call the *sub-pixel motion displacement*, must be found.

Lemma 1 *δx can be calculated in $\mathcal{O}(\log(YB_wB_h))$ space and $\mathcal{O}(B_wB_h)$ time.*

Proof This statement holds true upto the approximation that intensity values between two adjacent pixels vary linearly. Above, Y is the range of possible values of the function (or, equivalently the range of possible values that a pixel's color can assume).

If we estimate the sub-pixel motion displacement as $(\delta x, \delta y)$, then the cost function is $\mathcal{C}(RB(x, y, t), CB(a + \delta x, b + \delta y, u))$. Thus if we use mean square error as our matching criterion, and we treat it as an independent function of δx and δy, we obtain the equation

$$MSE(\delta x) = \sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} [\mathcal{F}(x, y) - \mathcal{G}(x + \delta x, y)]^2 \qquad (2.7)$$

For the purpose of calculating $\mathcal{F}$ and $\mathcal{G}$ with non-integral parameters, we will use linear interpolation. In order to perform such an estimation, the function must be known at the closest integral valued parameters. Since we have values of the pixels adjacent to the reference block easily available, and since this is not true for the contextual block, we reformulate the above equation without loss of generality as

$$MSE(\delta x) = \sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} [\mathcal{F}(x + \delta x, y) - \mathcal{G}(x, y)]^2 \qquad (2.8)$$

By expanding Equation 2.8 we obtain

$$MSE(\delta x) = \tag{2.9}$$
$$\sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} \left[\mathcal{F}^2(x+\delta x, y) + \mathcal{G}^2(x,y) - 2\mathcal{F}(x+\delta x, y)\mathcal{G}(x,y)\right]$$

This is the equivalent of using a fixed grid (the contextual block) and variably displacing the entire reference frame, in order to find the best sub-pixel displacement, (instead of the intuitive fixing of the image frame and movement of the block whose motion we are determining).

Assuming that the optimal sub-pixel displacement along the x axis of the contextual block under consideration is the δx that minimizes the function $MSE(\delta x)$ in Equation 2.9 which models $\mathcal{C}(RB(x_1, y_1, t), CB(a+\delta x, b))$, we proceed to calculate

$$\frac{d}{d(\delta x)} MSE(\delta x) = \tag{2.10}$$
$$\sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} \left[2\mathcal{F}(x+\delta x, y)\frac{d}{d(\delta x)}\mathcal{F}(x+\delta x, y) - 2\mathcal{G}(x,y)\frac{d}{d(\delta x)}\mathcal{F}(x+\delta x, y)\right]$$

Since we wish to determine the minimum of $MSE(\delta x)$, we will solve for δx using the constraint

$$\frac{d}{d(\delta x)} MSE(\delta x) = 0 \tag{2.11}$$
$$= \sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} \left[\left[\mathcal{F}(x+\delta x, y) - \mathcal{G}(x,y)\right]\frac{d}{d(\delta x)}\mathcal{F}(x+\delta x, y)\right]$$

Since $\mathcal{F}(x,y)$ is a discrete function, we use linear interpolation to approximate it as a continuous function for the purpose of representing $\mathcal{F}(x+\delta x, y)$ and computing $\frac{d}{d(\delta x)}\mathcal{F}(x+\delta x, y)$.

We consider the case of a positive δx first.

$$\forall \delta x,\ 0 \leq \delta x \leq 1,$$

$$\mathcal{F}(x+\delta x, y) = \mathcal{F}(x,y) \; + \; \delta x \, [\mathcal{F}(x+1,y) - \mathcal{F}(x,y)] \tag{2.12}$$

$$\frac{d}{d(\delta x)} \mathcal{F}(x+\delta x, y) = \mathcal{F}(x+1,y) - \mathcal{F}(x,y) \tag{2.13}$$

Applying Equations 2.12 and 2.13 to Equation 2.11, we obtain

$$\{ \sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} [\, [\mathcal{F}(x+1,y) \; - \; \mathcal{F}(x,y)] \tag{2.14}$$

$$[\, \mathcal{F}(x,y) \; - \; \mathcal{G}(x,y) \; + \; \delta x \, [\mathcal{F}(x+1,y) \; - \; \mathcal{F}(x,y)] \,] \,] \, \} \; = \; 0$$

Further expanding, and grouping to separate terms with δx,

$$\{ \sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} [\, \delta x \, [\mathcal{F}(x+1,y) - \mathcal{F}(x,y)]^2 \tag{2.15}$$

$$+ \, \mathcal{F}(x+1,y)\mathcal{F}(x,y) \; - \; \mathcal{F}^2(x,y)$$

$$+ \, \mathcal{F}(x,y)\mathcal{G}(x,y) \; - \; \mathcal{F}(x+1,y)\mathcal{G}(x,y) \,] \, \} \; = \; 0$$

Therefore, by rearranging terms and rewriting the above, we can obtain the closed form solution

$$\delta x = \tag{2.16}$$

$$\frac{\sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} [\, [\mathcal{F}(x+1,y) - \mathcal{F}(x,y)] [\mathcal{F}(x,y) - \mathcal{G}(x,y)] \,]}{\sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} [\mathcal{F}(x+1,y) - \mathcal{F}(x,y)]^2}$$

Similarly an independent calculation can be performed to attempt to ascertain what negative value of δx is optimal. The solution in the the case of a negative δx is a minor variation of the above and is outlined below

$$\forall \delta x, \; -1 \leq \delta x \leq 0,$$

$$\mathcal{F}(x+\delta x, y) = \mathcal{F}(x,y) \; + \; \delta x \, [\mathcal{F}(x,y) - \mathcal{F}(x-1,y)] \tag{2.17}$$

$$\frac{d}{d(\delta x)} (\mathcal{F}(x+\delta x, y)) = \mathcal{F}(x,y) - \mathcal{F}(x-1,y) \tag{2.18}$$

Apply Equations 2.17 and 2.18 to Equation 2.11 to obtain

$$\{ \sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} [\,[\mathcal{F}(x,y) \;-\; \mathcal{F}(x-1,y)] \tag{2.19}$$

$$[\,\mathcal{F}(x,y) \;-\; \mathcal{G}(x,y) \;+\; \delta x\,[\mathcal{F}(x,y) \;-\; \mathcal{F}(x-1,y)]\,]\,\} \;=\; 0$$

After algebraic manipulation, we obtain $\delta x =$

$$\frac{\sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} [\,[\mathcal{F}(x,y) - \mathcal{F}(x-1,y)]\,[\mathcal{F}(x,y) - \mathcal{G}(x,y)]\,]}{\sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} [\mathcal{F}(x,y) - \mathcal{F}(x-1,y)]^2} \tag{2.20}$$

Finally, we compute the MSE between the reference and contextual block, with each of the two δx's. The δx that results in the lower MSE is determined to be the correct one.

The closed form solution for each δx adds $B_w B_h$ terms in both the numerator and the denominator. Each term requires two subtractions and one multiplication. This is followed by computing the final quotient of the numerator and denominator summations. The time complexity is therefore $\mathcal{O}(B_w B_h)$. We note that computing the MSE is possible within this bound as well. Since the function values range upto Y, and only the running sum of the numerator and denominator need to be stored, the space needed is $\mathcal{O}(\log Y B_w B_h)$. □

Next, we note that it is possible to obtain a better representation of a block in the reference frame by completing the analysis of sub-pixel displacements along the orthogonal axis.

Lemma 2 *The cardinality of the set of points that represents the block under consideration from the reference frame is non-decreasing.*

Proof After a δx has been determined, the block must be translated by a quantity δy along the orthogonal axis. It is important to perform this calculation after the δx

translation has been applied, since it guarantees that the MSE after both translations is no more than the MSE after the first translation, ensuring the correctness of the algorithm. We can determine the δy in a manner analogous to the one used to determined δx, using the representation of the MSE below, with the definition $x' = x + \delta x$:

$$MSE(\delta y) = \sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} \left[\mathcal{F}(x', y) - \mathcal{G}(x', y + \delta y)\right]^2 \tag{2.21}$$

This is in turn equivalent to (by the same argument provided for the δx case):

$$MSE(\delta y) = \sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} \left[\mathcal{F}(x', y + \delta y) - \mathcal{G}(x', y)\right]^2 \tag{2.22}$$

Solving for δy is achieved using:

$$\forall \delta y,\ 0 \leq \delta y \leq 1,\ \delta y = \tag{2.23}$$
$$\frac{\sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} \left[\left[\mathcal{F}(x', y+1) - \mathcal{F}(x', y)\right]\left[\mathcal{F}(x', y) - \mathcal{G}(x', y)\right]\right]}{\sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} \left[\mathcal{F}(x', y+1) - \mathcal{F}(x', y)\right]^2}$$

$$\forall \delta y,\ -1 \leq \delta y \leq 0,\ \delta y = \tag{2.24}$$
$$\frac{\sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} \left[\left[\mathcal{F}(x', y) - \mathcal{F}(x', y-1)\right]\left[\mathcal{F}(x', y) - \mathcal{G}(x', y)\right]\right]}{\sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} \left[\mathcal{F}(x', y) - \mathcal{F}(x', y-1)\right]^2}$$

If the sub-pixel displacement $(\delta x, \delta y)$ determined results in an MSE between the contextual and reference blocks exceeding a given threshold, then the extra information is not incorporated into the current set representing the reference block. This prevents the contamination of the set with spurious information. It also completes the proof that either a set of points that enhances the current set is added, or none are - yielding a non-decreasing cardinality for the set representing the block that is being processed. □

By performing this analysis independently for each of the contextual blocks that corresponds to each of the reference blocks, we obtain a scattered data set (aside from

the uniformity of the scatter pattern within individual blocks) representing the frame whose resolution we are enhancing. If the sampling grid had infinite resolution, and we inspected the values registered on it at the points we have determined above to be in the image space, we would find that these data points are a good approximation. By repeating this process with a number of contextual frames for each of the reference frames, we can extract a considerable sized set of extra data points in the image space of the reference frame. At this stage we have theoretically enhanced the resolution of the video frame. However, the format of scattered data is not an acceptable format for most display media such as video monitors. Therefore we must process this information further to make it easily usable.

Chapter 3

Creating a Coherent Frame

3.1 Framework

We shall define a *uniform grid* to be the set $H_{P,k} \cup V_{Q,l}$ of lines in $\mathbf{R}^2$, where $H_{P,k} = \{x = k\alpha \mid k \in \{0, 1, 2, \ldots, P\},\ \alpha \in \mathbf{R},\ P \in \mathbf{N}\}$ specifies a set of vertical lines and $V_{Q,l} = \{y = l\beta \mid l \in \{0, 1, 2, \ldots, Q\},\ \beta \in \mathbf{R},\ Q \in \mathbf{N}\}$ specifies a set of horizontal lines. Specifying the values of P, Q, α, and β determines the uniform grid uniquely. Given a set S of points of the form (x, y, z), where z represents the intensity of the point (x, y) in image space, if there exist M, N, α, and β such that all the (x, y) of the points in the set S lie on the associated uniform grid, and if every data point (x, y) on the uniform grid has an intensity (that is a z component) associated with it, then we call the set S a *coherent frame*. Each of the frames in the original video sequence was a coherent frame. We seek to create a coherent frame from the data set D, that we have obtained through the process described in Chapter 2, with the constraint that the number of points in the coherent frame should be the same as that of the data set D.

The general principle we use to effect the transformation from scattered data to a coherent frame is to first construct a surface in $\mathbf{R}^3$ that passes through the input data. By representing this surface in a functional form with the x and y coordinates as parameters, it can then be evaluated at uniformly spaced points in the XY plane for the production of a coherent frame.

There are numerous methods available for the purpose, the tradeoff being the increased computational complexity needed to guarantee a greater level of accuracy of the mapping. While the efficiency of the procedure is important, in the light of

our concerns in Section 1.5, we would like to make our approximation as close a fit as possible. We note that although we will be forced to use interpolation techniques at this stage, we are doing so with a data set that has been increased in size as described in Chapter 2, so this is not equivalent to performing interpolation at the outset and is certainly an improvement over that.

Linear interpolation of points yields a very fast, completely scalable algorithm with the disadvantage that the result is optically suboptimal due the nature in which the eye perceives images. We resort to this only in the case of data that is not amenable, due to features like high variance, to interpolation by more sophisticated techniques. The next option considered was the bivariate generalization of univariate *polynomial interpolation*, the process of determining the $(n+1)$ coefficients of a n degree polynomial that passes through $(n+1)$ specified points, followed by its evaluation at the points of interest in the domain. Given a set of points $\{(x_0, y_0), \ldots, (x_n, y_n)\}$, the unique *Newton interpolating polynomial* [CK94] takes the form

$$p_n = \sum_{l=0}^{n} \left[\Delta_x^l(x_0, \ldots, x_l) f(x) \prod_{j=0}^{l-1} (x - x_j) \right] \tag{3.1}$$

if we adopt the convention that $\prod_{j=0}^{-1} (x - x_j) = 1$. We also note that $\Delta_x^0(x_0) f(x) = f(x_0)$ by definition and that the *divided differences* obey the recursive formulation

$$\Delta_x^l(x_i, \ldots, x_{i+l}) f(x) = \frac{\Delta_x^{l-1}(x_{i+1}, \ldots, x_{i+l}) f(x) - \Delta_x^{l-1}(x_i, \ldots, x_{i+l-1}) f(x)}{x_{i+l} - x_i} \tag{3.2}$$

However, we are concerned with the graphical characteristics of the approximation rather than the mathematical properties, and polynomial interpolation frequently results in a function that takes on values that may stray considerably in the intervals between the points that have been specified. This is a property that we wish to

avoid. The next alternative is piecewise polynomial interpolation, along with the imposition of continuity conditions at the boundaries of the various polynomials used to interpolate the subsets of data points, such that the transition from one polynomial to another appears smooth visually. This brings us to the use of *spline functions*, which have the following characteristics. They

1. Are defined on a finite interval $[a, b]$,
2. Have a *degree* $k > 0$, (or *order* $k+1$),
3. Have a strictly increasing sequence of *knots*, $\lambda_j,\ j \in \{0, \ldots, (g+1)\}$, where $\lambda_0 = a$ and $\lambda_{g+1} = b$,
4. Are specified within individual knot intervals, $[\lambda_j, \lambda_{j+1}]$, by polynomials of degree at most k, which are denoted by $s_{[\lambda_j, \lambda_{j+1}]} \in \mathcal{P}_k$, where $j \in \{0, \ldots, g\}$,
5. Are continuous, along with their derivatives of order up to $k-1$, on the interval $[a, b]$.

Given the above set of knots, the collection of functions that satisfy the specified properties form a vector space, denoted by $\eta_k(\lambda_0, \ldots, \lambda_{g+1})$, in which any element can be expressed as

$$p_{k,j}(x) = \sum_{i=0}^{k} a_{i,j}(x - \lambda_j)^i \tag{3.3}$$

$$\text{where } \lambda_j \leq x \leq \lambda_{j+1},\ j \in \{0, \ldots, g\}$$

subject to the conditions that

$$p^{(l)}_{k,j-1}(\lambda_j) = p^{(l)}_{k,j}(\lambda_j) \tag{3.4}$$

$$\text{where } j \in \{1, \ldots, g\},\ l \in \{0, \ldots, k-1\}$$

A simple and useful example of a spline function is the *truncated power function.* For a given constant c it is defined as

$$(x-c)_+^k = \begin{cases} (x-c)^k & \text{if } x \geq c \\ 0 & \text{if } x < c \end{cases} \tag{3.5}$$

Spline functions are most suitable for our purpose since

- Polynomials can be evaluated quickly,
- Splines are very flexible as they are piecewise defined functions,
- Past use in graphics has confirmed the applicability - the literature [Die93, dBH87] on the subject contains results that can be used to make decisions where tradeoffs are required.

3.2 General B-Spline Interpolation

At the outset, we note that the theory of splines is a very rich field and that the properties of these functions are numerous. We will not attempt to cover the topic in depth, preferring to outline only the concepts required for an understanding of the task at hand. Consequently, subtleties that must be understood for an implementation but are not central to understanding the ideas are not introduced. This section is included for completeness, and further details may be found in [Die93].

By representing an arbitrary spline in terms of a weighted sum of normalized basis splines, we can perform *general B-spline interpolation* [dB] through the specified set of points.

3.2.1 B-Splines

A normalized B-spline of degree k with the knots $\lambda_i, \ldots, \lambda_{i+k+1}$ is defined as

$$N_{i,k+1}(x) \;=\; (\lambda_{i+k+1} \;-\; \lambda_i)\Delta_t^{k+1}(\lambda_i, \ldots, \lambda_{i+k+1})(t-x)_+^k \tag{3.6}$$

We note that $N_{i,k+1}(x) = 0$ if $x \notin [\lambda_i, \lambda_{i+k+1}]$ and that a B-spline is always positive. Since there are a total of gk constraints placed by Equation 3.4 on the possible values that are acceptable for the $(g+1) \times (k+1)$ coefficients $a_{i,j}$ in Equation 3.3,

$$\dim(\eta_k(\lambda_0, \ldots, \lambda_{g+1})) = g + k + 1 \tag{3.7}$$

Given a set of knots $\lambda_0, \ldots, \lambda_{g+1})$, and since each B-spline of the form in Equation 3.6 is determined by $(k+1)$ knots, we can construct $(g-k+1)$ linearly independent B-splines. From Equation 3.7 we determine that we need $2k$ more linearly independent basis functions to form a spanning set for the vector space η_k. We introduce the knots $\lambda_{-k}, \ldots, \lambda_{-1}$ and $\lambda_{g+2}, \ldots, \lambda_{g+k+1}$ with arbitrary values such that

$$\begin{aligned} \lambda_{-k} < \lambda_{-k+1} < \cdots < \lambda_{-1} < \lambda_0 = a \\ b = \lambda_{g+1} < \lambda_{g+2} < \cdots < \lambda_{g+k} < \lambda_{g+k+1} \end{aligned} \tag{3.8}$$

Now we can represent any spline as

$$s(x) = \sum_{i=-k}^{g} c_i N_{i,k+1}(x) \tag{3.9}$$

where the c_i are called the *B-spline coefficients* of $s(x)$. As previously mentioned the basis is a normalized one, that is $\forall x \in [a, b]$,

$$\sum_{i=-k}^{g} N_{i,k+1}(x) \;=\; 1 \tag{3.10}$$

Evaluation of the spline can be performed by a direct recursive formulation due to Carl de Boor [dBH87]. $s(x) = c_j^{[k]}(x)$ where

$$c_j^{[k]}(x) = \begin{cases} c_j & \text{if } i = 0 \\ \frac{(x-\lambda_j)c_j^{[i-1]} + (\lambda_{j+k+1-i} - x)c_{j-1}^{[i-1]}(x)}{\lambda_{j+k+1-i} - \lambda_j} & \text{if } i > 0 \end{cases} \tag{3.11}$$

3.2.2 Tensor Product Splines

Since our data domain is bivariate, we consider the various generalizations of the theory of univariate splines. The most direct extension is through the use of *tensor product splines*, which yield efficient calculations. They

1. Are defined on R, a finite region $[a, b] \times [c, d]$,

2. Have a degree $k > 0$ in x, and degree $l > 0$ in y,

3. Have two independent strictly increasing knot sequences:
   $\lambda_i,\ i \in \{0, \ldots, (g+1)\}$, where $\lambda_0 = a$ and $\lambda_{g+1} = b$,
   $\mu_j,\ j \in \{0, \ldots, (h+1)\}$, where $\mu_0 = c$ and $\mu_{h+1} = d$,

4. Are specified within the subrectangle $R_{i,j} = [\lambda_i, \lambda_{i+1}] \times [\mu_j, \mu_{j+1}]$ by a polynomial of degree k in x and another of degree l in y. Thus the specified tensor product spline is denoted by $s_{R_{i,j}} \in \mathcal{P}_k \times \mathcal{P}_l$, where $i \in \{0, \ldots, g\}$ and $j \in \{0, \ldots, h\}$,

5. Are continuous over R, along with their partial derivatives as specified by $\frac{\partial^{i+j} s(x,y)}{\partial x^i y^j} \in \mathcal{C}(R)$ where $i \in \{0, \ldots, (k-1)\}$ and $j \in \{0, \ldots, (l-1)\}$.

Given the above set of knots, the collection of functions that satisfy the specified properties form a vector space, denoted by $\eta_{k,l}(\lambda_0, \ldots, \lambda_{g+1}; \mu_0, \ldots, \mu_{h+1})$, or $\eta_{k,l}$. Applying general rule for calculating the dimension of tensor product spaces, we have

$$\begin{aligned} \dim(\eta_{k,l}) &= \dim(\eta_k(\lambda_0, \ldots, \lambda_{g+1}))\dim(\eta_l(\mu_0, \ldots, \mu_{g+1})) \\ &= (g+k+1)(h+l+1) \end{aligned} \tag{3.12}$$

Therefore in analogy to the univariate case, we introduce the boundary knots

$$\lambda_{-k} < \lambda_{-k+1} < \cdots < \lambda_{-1} < \lambda_0 = a \tag{3.13}$$

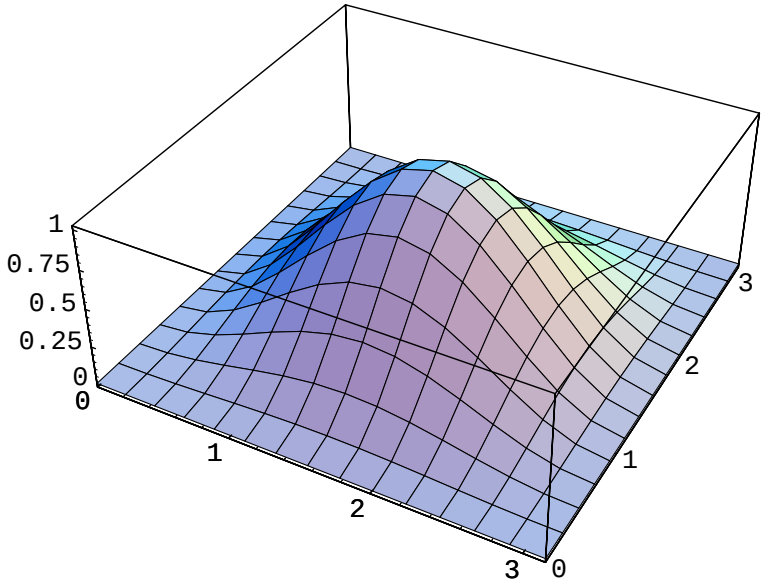


Figure 3.1: A tensor product basis spline.

$$b = \lambda_{g+1} < \lambda_{g+2} < \cdots < \lambda_{g+k} < \lambda_{g+k+1}$$

$$\mu_{-l} < \mu_{-l+1} < \cdots < \mu_{-1} < \mu_0 = c$$

$$d = \mu_{h+1} < \mu_{h+2} < \cdots < \mu_{h+l} < \mu_{h+l+1}$$

so that we may represent any tensor product spline as

$$s(x,y) = \sum_{i=-k}^{g} \sum_{j=-l}^{h} c_{i,j} N_{i,k+1}(x) M_{j,l+1}(y) \tag{3.14}$$

where $M_{j,l+1}(y) = (\mu_{j+l+1} - \mu_j)\Delta_t^{l+1}(\mu_j, \ldots, \mu_{j+l+1})(t-y)_+^l$. We also note that $N_{i,k+1}(x)M_{j,l+1}(y) = 0$ if $x \notin [\lambda_i, \lambda_{i+k+1}]$ or $y \notin [\mu_j, \mu_{j+l+1}]$ and that the tensor product spline is always positive. The basis is normalized, that is $\forall x \in [a,b]$, $\forall y \in [c,d]$

$$\sum_{i=-k}^{g} \sum_{j=-l}^{h} N_{i,k+1}(x) M_{j,l+1}(y) = 1 \tag{3.15}$$

To evaluate the spline at a point $(x, y) \in [\lambda_r, \lambda_{r+1}) \times [\mu_s, \mu_{s+1})$, we can use

$$s(x, y) = \sum_{i=r-k}^{r} a_i N_{i,k+1}(x) \tag{3.16}$$

where

$$a_i = \sum_{j=s-l}^{s} c_{i,j} M_{j,l+1}(y) \tag{3.17}$$

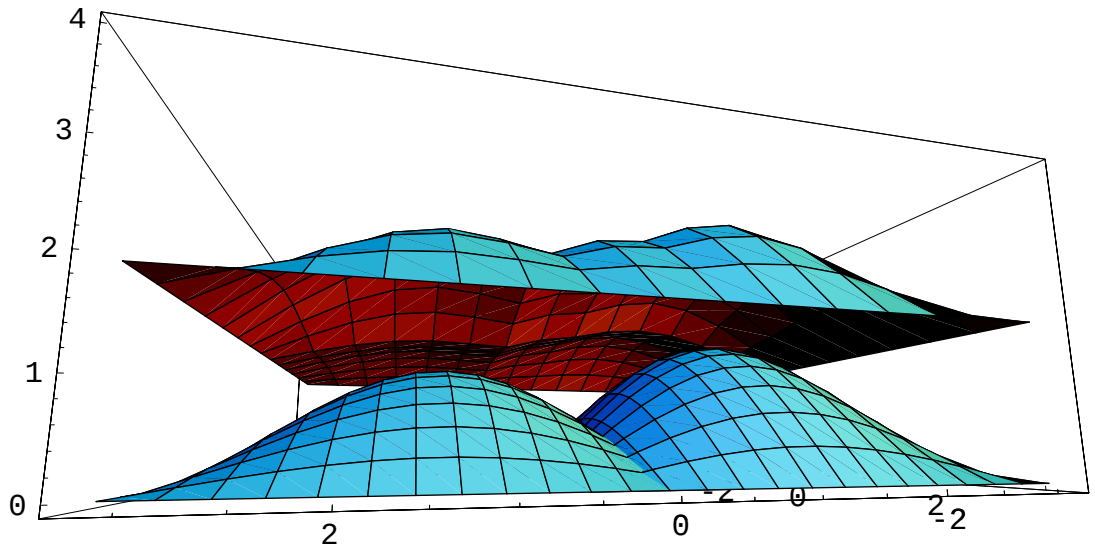


Figure 3.2: A surface constructed using tensor product basis splines.

3.2.3 Smoothing Criterion

Since the measurement of intensities of pixels is subject to some error, along with the fact that the motion estimation algorithm imposes constraints of local uniform motion fields that do not exist in object space, resulting in some amount of error, we must use a *smoothing norm* that will allow us to find the bivariate tensor product spline that will best fit the arbitrary data values.

Dierckx specifies one possible form for the norm as

$$\eta(\mathbf{c}) = \sum_{\mathbf{q=1}}^{\mathbf{g}} \sum_{\mathbf{j=-l}}^{\mathbf{h}} \left[\sum_{\mathbf{i=-k}}^{\mathbf{g}} \mathbf{c_{i,j} a_{i,q}} \right]^{\mathbf{2}} + \sum_{\mathbf{r=1}}^{\mathbf{h}} \sum_{\mathbf{i=-k}}^{\mathbf{g}} \left[\sum_{\mathbf{j=-l}}^{\mathbf{h}} \mathbf{c_{i,j} b_{j,r}} \right]^{\mathbf{2}} \tag{3.18}$$

$$\text{where } a_{i,q} = N^{(k)}_{i,k+1}(\lambda_q +) - N^{(k)}_{i,k+1}(\lambda_q -), \tag{3.19}$$

$$\text{and } b_{j,r} = M^{(l)}_{j,l+1}(\mu_r +) - M^{(l)}_{j,l+1}(\mu_r -) \tag{3.20}$$

Here $\mathbf{c}$ is the set of B-spline coefficients of $s(x, y)$, and $a_{i,q}$ is the discontinuity of k^{th} derivative of the B-spline $N_{i,k+1}(x)$ at the knot λ_q, and similarly $b_{j,r}$ is the discontinuity of l^{th} derivative $M_{j,l+1}(y)$ at the knot μ_r. This norm must be minimized subject to the additional constraint $\delta(\mathbf{c}) \leq \mathbf{S}$, where

$$\delta(\mathbf{c}) = \sum_{\mathbf{r=1}}^{\mathbf{m}} \left[\mathbf{w_r z_r} - \sum_{\mathbf{i=-k}}^{\mathbf{g}} \sum_{\mathbf{j=-l}}^{\mathbf{h}} \mathbf{c_{i,j} w_r N_{i,k+1}(x_r) M_{j,l+1}(y_r)} \right]^{\mathbf{2}} \tag{3.21}$$

where (x_r, y_r, z_r) for $r \in \{1, \ldots, m\}$ are the data values for which we are trying to find an approximating surface and w_r are the weights attached to the points during splining. For the data set to be valid, the conditions $\forall x_r,\ x_r \in [a, b],\ \forall y_r,\ y_r \in [c, d]$ must hold.

Minimizing Equation 3.18 with a constraint on Equation 3.21 is an optimization problem that can be solved by the method of Lagrange, with the required coefficients $c_{i,j}$ as the least squared solution of and overdetermined system. Details of the algorithm are easily available in the literature [Die93].

3.3 Implementation Issues

We have seen that given an arbitrary scattered data set, we can construct a coherent frame that provides a very good approximation of the surface specified by the original data set. However, if we were to work with the entire data set at hand, our algorithm

would not be scalable. This is due to the fact that memory requirement would exceed that which is available on today's computer systems, even for processing a typical MPEG sized video sequence.

Keeping in mind the fact that splines use only a fixed number of neighbouring points, we employ the technique of decomposing the data set into spatially related sets of fixed size. Each set contains all the points within a block in image space. The disadvantage of working with such subsets is that visible artifacts develop at the boundaries in the image space of these blocks. To avoid this we *compose* the blocks by using data from adjacent blocks to create a border of data points around the block in question, so that the spline surface constructed for a block is continuous with the surfaces of the adjacent blocks. Working with a block at a time in this manner, we construct surfaces for each region in the image space, and evaluate the surface on a uniform grid, to obtain the representation we desire. When this is done for the entire set, we have obtained a coherent frame.

Lemma 3 *Creating a block in a coherent frame using data extracted from temporal context requires* $\mathcal{O}(B_w B_h \sigma(k) T)$ *operations.*

Proof Here T frames are incorporated, and degree k polynomial based B-spline tensor products are used to perform the transformation from scattered to gridded data. The complexity of splining depends on the number of points, which is $B_w B_h$, the product of the block width and height. $\sigma(k)$ is the cost to perform the splining operations per point in the data set.

Since B-splines have the property of *local support*, that is only a fixed number of adjacent B-splines are required for the evaluation of any given point in the space spanned by them (such as the surface being represented), and each B-spline can be represented as a fixed length vector of coefficients, the approximation of a surface specified by a set of points has time complexity that is only bound by the degree of

the polynomials used and the multiplicity of the knots [dB78, Sch81].

While in theory B-spline interpolation has complexity $\mathcal{O}(k \log^2 k)$, constructing a B-spline as well as evaluating it along with all its derivatives can be done in $\mathcal{O}(k^2)$ operations in practice. Thus $\sigma(k)$ is k^2. The above result follows immediately from this. □

Chapter 4

Results

4.1 Complexity of Algorithm

Theorem 1 *Processing an L frame video sequence using T frames of temporal context to enhance the resolution of each frame, yields the higher resolution version in $\mathcal{O}(\frac{F_w F_h}{B_w B_h} L[\mu(B_w, B_h) + B_w B_h \sigma(k) T])$ time.*

Proof There are $\frac{F_w F_h}{B_w B_h}$ blocks. For each block, it takes $\mu(B_w, B_h)$ time to perform motion estimation (assuming fixed range exhaustive search) and re-estimation of the motion with sub-pixel accuracy, by Lemma 1. Transforming the data set obtained for a block takes $\mathcal{O}(B_w B_h \sigma(k) T)$ time, by Lemma 3. Finally, there are L frames to process. The above stated time complexity bound follows from this. □

4.2 GROW

The application software, GROW, was developed as an implementation of the algorithm. It provides a flexible command line interface that allows the individual specification of numerous parameters such as the horizontal and vertical dimensions of the blocks used for motion estimation, the blocks used for spline interpolation, the extra border used for composing blocks for spline interpolation, the degrees of the splines used, the factors by which the output frame is scaled up from the original, the maximum range in which to perform motion estimation, the number of previous and later frames used to enhance any given frame, the error threshold that is acceptable during motion estimation and that for spline interpolation.

The parameters entered serve to guide but not enforce the algorithm. For example, splining starts with the degree the user enters but automatically drops to lower degrees (as less context is used) when the surface returned is not close enough to the points it is approximating. The error threshold specified for splining is used to scale up bounds that are calculated using heuristics from the literature [Die93]. Intermediate results, such as the sub-pixel displacements, are calculated and kept in high precision floating point format. When error thresholds are crossed, the data in question is not used. Thus occlusion and peripheral loss of objects is dealt with by the effective result of using only reference image data for the relevant region.

4.3 Experiments

The methodology employed to test the program is specified below. A video sequence was sub-sampled, on a frame by frame basis. The new lower resolution sequence was used as data, and the output compared to the original sequence.

If $\mathcal{G}$ represents the high resolution video sequence, and $\mathcal{F}$ represents the high resolution sequence obtained by running GROW on the low resolution version associated with $\mathcal{G}$, where the low resolution version was obtained through the sub-sampling of $\mathcal{G}$, then the *Signal to Noise Ratio* is defined as:

$$SNR = 10 \log \frac{\sqrt{\frac{1}{B_w B_h} \sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} \mathcal{F}(x,y)}}{\frac{1}{(B_w B_h)^2} \sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} \left[\mathcal{F}(x,y) - \mathcal{G}(x,y)\right]^2} \tag{4.1}$$

Using this metric to measure distortion, we compare the strength of the signal of the sequence produced by GROW under various cicrumstances.

As an example, note below the parts of five smaller frames in Figure 4.3 that are part of the original video sequence, **garden**. By using all of them as input to GROW, the higher resolution output corresponding to the the third original frame is

Figure 4.1: Original frames.

generated and shown. Figure 4.3 shows versions constructed with GROW as well as with spline interpolation, along with the original frame. Artifacts that arise in the final images are due to splining.

We note that the improvement over spline interpolation by the addition of data obtained from temporal context is done at little practical cost. This is demonstrated in the data in Figure 4.5 by the fact that the amount of time spent on this is minimal compared to time spent on the spline construction and evaluation.

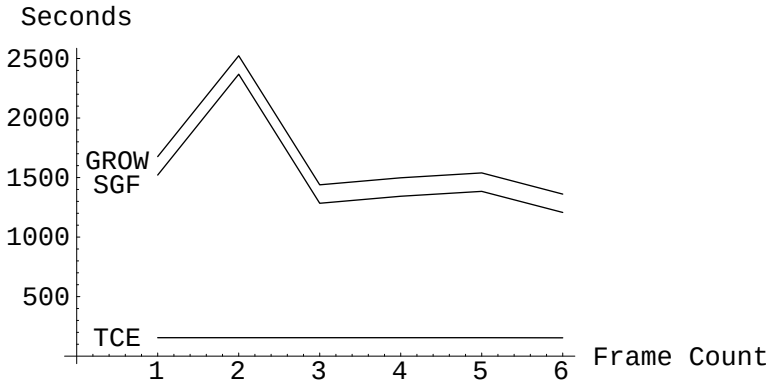
The effect of varying the parameters of the algorithm is an observable as well as measurable change in the quality of the output images, as measured by the SNR heuristic. The meaning of each of the parameters is explained in detail in Chapter 6.

Figure 4.2: Higher resolution version of the central frame obtained using GROW.

Figure 4.3: The same frame with its resolution increased using only spline interpolation.

Figure 4.4: The frame whose resolution is being increased.

Figure 4.5: Time spent broken down by splining and temporal context extraction.



s	t	j	k	x	y	c	d	p	l	m	n	a	b	z	u	e	SNR
4	4	4	4	8	8	2	2	2	2	64	64	5	5	1	0	4	30.59
8	8	4	4	8	8	2	2	2	2	64	64	5	5	1	0	4	30.40
16	16	4	4	8	8	2	2	2	2	64	64	5	5	1	0	4	28.34
16	16	8	8	8	8	2	2	2	2	64	64	5	5	1	0	4	28.39
16	16	4	4	4	4	2	2	2	2	64	64	5	5	1	0	4	28.33
16	16	4	4	16	16	2	2	2	2	64	64	5	5	1	0	4	28.32
16	16	4	4	8	8	4	4	2	2	64	64	5	5	1	0	4	27.49
16	16	4	4	8	8	2	2	0	0	64	64	5	5	1	0	4	28.39
16	16	4	4	8	8	2	2	1	1	64	64	5	5	1	0	4	28.42
16	16	4	4	8	8	2	2	3	3	64	64	5	5	1	0	4	28.39
16	16	4	4	8	8	2	2	2	2	128	128	5	5	1	0	4	28.34
16	16	4	4	8	8	2	2	2	2	64	64	3	3	1	0	4	28.45
16	16	4	4	8	8	2	2	2	2	64	64	5	5	3	0	4	28.83
16	16	4	4	8	8	2	2	2	2	64	64	5	5	1	0	8	28.35

As mentioned in the overview, certain aspects such as spline degree selection have been automated, while others such as splining error bounds are semi-automatically calculated, using heuristics and minimal user input. To get an optimal image sequence, though, it is necessary for the user to manually adjust the values fed into the algorithm. Finally, we note that by hand tuning GROW, we can obtain a significantly better result than possible by spline interpolation without using temporal context. This can be noted from the table as the row which uses no past and future frames effects this kind of spline interpolation, but does not yield as strong a signal as the best case output of GROW.

Chapter 5

Future Work

The goal of this thesis was to demonstrate that the use of temporal context in an image stream can indeed provide an improvement in individual frame resolution. In the future, advanced techniques for effecting this goal may be incorporated to produce results that may be applied to real world scenarios in real time.

The currently suggested methods are primitive by comparison to what could potentially be used. We restricted our work to these methods because they were computationally efficient and enough to provide a proof of concept. Some of the possible extensions that would improve the results at a cost of much higher computational requirements are suggested below.

During motion estimation, arbitrary blocks need not be used. Instead, by using hierarchical clustering techniques on local motion fields of the image stream, objects may be identified more accurately, and tracking may be performed on the exact shape. Rather than using the simple linear translation model, one can incorporate a model where a full range of affine transformations can be represented, including rotation, scaling, and warping. During sub-pixel motion re-estimation, we ignore the correlation between the orthogonal components. Instead, the convolution of these could be figured into the analytical calculations used to estimate these. Finally, replacing the use of splines could be explored with suitable image processing filtering techniques.

For an implementation that operates in real time, even our application, leave alone the proposed extensions of it, would require some high performance dedicated hardware.

Chapter 6

Application GROW User Manual

For the purpose of verifying the idea proposed, we created an application program that accepts a video sequence as input, processes it and produces a higher resolution output video sequence. It works with raw greyscale (or equivalently the Y component files of a YUV sequence). It was written and has been tested in the Unix environment and is known to run under Sun's SunOS 4.x, Sun's Solaris 2.x, DEC's OSF1 3.x, IBM's AIX 3, SGI's IRIX 6.1 with the NFS and AFS filesystems. Since it has been written with no device dependencies, it should be easily portable to other platforms.

```
Usage:  grow
        -f <first-frame> -g <last-frame> -h <frame-height>
        -w <frame-width> -i <input-files> -o <output-files>
        [-s <x-dimension-of-splining-blocks>]
        [-t <y-dim-spl-blks>]
        [-j <x-dim-extra-pixels-for-splining>]
        [-k <y-dim-extra-pixels>]
        [-x <x-dim-of-motion-estimation-blocks>]
        [-y <y-dim-mot-est-blks>]
        [-c <horizontal-scaling-factor>]
        [-d <vertical-scaling-factor>]
        [-p <number-of-previous-frames-used-as-context>]
        [-l <num-later-frms>]
        [-m <maximum-x-displacement-during-motion-est>]
        [-n <max-y-disp>]
        [-a <x-dimension-spline-degree>]
        [-b <y-dimension-spline-degree>]
        [-z <spline-acceptable-error-factor>]
        [-u <class-of-max-accept-err>]
        [-e <maximum-error-during-motion-est>]
        [-v (verbose)]
```

Figure 6.1: Command line parameters of program GROW.

GROW makes use of Paul Dierckx's SURFIT routines, available from any Netlib repository. These were converted from Fortran to C using F2C. Thus, if the user

wishes to compile the source, the appropriate library should be available to the compiler.

A large number of parameters of the algorithm are accessible to the user through a command line interface. While it is not necessary for the user to supply these since GROW will use default values in the absence of these being fed in explicitly, the option to adjust them exists and they are therefore explained here.

-f The index number of the first frame where processing of the video sequence is to begin. For example, if the entire sequence of files $frame1.Y, \ldots, frame20.Y$ is to be processed then the value used as a parameter of -f would be 1.

-g The index number of the last frame to be processed. In the above example, the parameter of -g would be 20.

-h The integer number of pixels in the vertical direction of each individual frame in the video sequence.

-w The integer number of pixels in the horizontal direction of each individual frame in the video sequence.

-i The string name of the input sequence of files to be used. For example, to process the sequence $frame1.Y, \ldots, frame20.Y$, the parameter $frame$ would be used for the -i option.

-o The string name of the output video sequence. For example, if the parameter $newframe$ was used for the -o option, then if there are 5 output frames, they will be stored as $newframe0.Y, \ldots, newframe4.Y$.

The above options must be specified for all processing. The below options may be varied at the user's discretion. It is not necessary to use any of these. The application has defaults built in.

-s The integer parameter supplied is the horizontal dimension of the block size used during spline interpolation and evaluation.

-t The integer parameter is the vertical dimension of the block size used during spline interpolation and evaluation.

-j During spline interpolation of a block, pixels from adjacent blocks are used to sew the blocks together. The integer parameter supplied is the horizontal width of the border that runs parallel to the vertical axis.

-k The integer parameter supplied is the vertical width of the above border that runs parallel to the horizontal axis.

-x The integer parameter supplied is the horizontal dimension of the block size used during motion estimation.

-y The integer parameter is the vertical dimension of the block size used during motion estimation.

-c The integer parameter supplied specifies the factor by which the horizontal resolution is increased when evaluating the spline surface representing the image.

-d The integer parameter supplied specifies the factor by which the vertical resolution is increased when evaluating the spline surface.

-p The integer parameter is the number of previous frames that are used during temporal context extraction to add information to the data set representing the frame that is currently being processed.

-l The integer parameter is the number of later frames that are used to add information for the frame that is currently being processed.

-m The integer parameter supplied is the maximum horizontal direction in which to search for a motion vector. If this is set to 0, then a full search throughout the entire image is performed.

-n The integer parameter supplied is the maximum vertical direction in which to search for a motion vector. If this is set to 0, then a full search throughout the entire image is performed.

-a The integer parameter is the maximum degree of the polynomials used in the horizontal direction. If this yields unacceptable results, the application will automatically try lower degree polynomials as well.

-b The integer parameter is the maximum degree of the polynomials used in the vertical direction. If this yields unacceptable results, the application will automatically try lower degree polynomials as well.

-z During spline interpolation the smoothing norm is subject to a least mean squares constraint. The floating point parameter supplied here specifies that value.

-u The spline routines from SURFIT provide a return value indicating what kind of approximation was performed. The integer parameter supplied here specifies the level of approximation considered acceptable. This should usually be left at 0.

-e During motion estimation, if the MAE between the reference and currently considered blocks is below the integer parameter supplied here, then an acceptable match is considered to be found.

-v When this option is used the application sends detailed information about what it is currently doing to the standard output.

Appendix A

Source code of GROW

```
/* Grow.c - Grows the frame size of a video sequence *
 *                                                   *
 * Written by Ashish Gehani                          */

#include<stdlib.h>
#include<stdio.h>
#include<strings.h>
#include<sys/file.h>
#include<sys/time.h>
#include<time.h>
#include<math.h>
#include"f2c.h"
#include"spline.h"

#define MAX_STR_LEN 80

int XSPL=5; /* Degree of the splines in the x dimension */
int YSPL=5; /* Degree of the splines in the y dimension */
int XBL=8; /* Number of pixels in the horizontal dimension that *
            * define a block                                    */
int YBL=8; /* Number of pixels in the vertical dimension that *
            * define a block                                  */
int FFM=2; /* Number of frames ahead of the current that are to *
            * be used for providing contextual information      */
int BFM=2; /* Number of frames after the current one to be *
            * used as context for the spline interpolated  *
            * construction of the current one             */
int XSM=64; /* Number of pixels to search in the horizontal    *
             * direction when attempting to determine a motion *
             * estimation vector, 0 <=> exhaustive search      */
int YSM=64; /* Number of pixels to search in the vertical      *
             * direction when attempting to determine a motion *
             * estimation vector, 0 <=> exhaustive search      */
int ERMC=4; /* Maximum mean acceptable error per pixel *
             * during motion estimation                */
int ERSP=1; /* Acceptable error factor for spline construction */
int THR=0; /* Threshold of class of errors acceptable during splining */
int XSBL=16; /* Number of pixels in the horizontal dimension that          *
              * define a block for the purpose of spline interpolation     *
              * - fragile parameter - should preferably be a factor of the *
              * horizontal dimension of the motion estimated image         */
int YSBL=16; /* Number of pixels in the vertical dimension that            *
              * define a block for the purpose of spline interpolation     *
              * - fragile parameter - should preferably be a factor of the *
              * vertical dimension of the motion estimated image           */
int XIF=2; /* Factor by which horizontal resolution increases *
            * during spline evaluation                        */
int YIF=2; /* Factor by which vertical resolution increases *
```

```
            * during spline evaluation                        */
int XBR=4; /* Number of horizontal direction extra pixels from adjacent *
            * blocks used for spline construction                        */
int YBR=4; /* Number of vertical direction extra pixels from adjacent *
            * blocks used for spline construction                      */

int height, hmc; /* Height of original frame, motion estimated one */
int width, wmc; /* Width of original frame, motion estimated one */
int hsp; /* Height of frame used for spline construction */
int wsp; /* Width of frame used for spline construction */
int begin; /* First frame in sequence */
int last; /* Last frame in sequence */
char input[MAX_STR_LEN], output[MAX_STR_LEN]; /* Names of file sequence */
int verbose=0; /* Set to 1 when verbose output is needed */
unsigned char **frame; /* Pointer to the array of pointers to frames */
unsigned char **mc; /* Pointer to the array of pointers to frames *
                     * created by motion estimation searches      */
doublereal *xhr,*yhr,*zhr; /* Pointers to the coordinate arrays of  *
                            * high resolution version of the images */
doublereal *xvhr,*yvhr,*zvhr; /* Pointers to the coordinate arrays of *
                               * very high resolution blocks of data  *
                               * used for constructing spline surface */
doublereal *w; /* Pointer to array of weights used for splining */
doublereal *xf,*yf,*zf; /* Pointers to the coordinate arrays of *
                         * very high resolution blocks of data  *
                         * used for evaluating spline surface   */
doublereal *vhrbuf; /* Pointer to array of z coordinates of collection *
                     * of blocks being buffered for output to a file   */

void CurrentTime(char *message)
{
    time_t now; /* Current time and date */

    /* Output the current time and date */

    if(verbose){
        now=time(NULL);
        printf("%s: %s",message,ctime(&now));
    }
}

void Usage(char *prog)
{
    fprintf(stderr,"Usage:\t%s\n\t",prog);
    fprintf(stderr,"-f <first-frame> ");
    fprintf(stderr,"-g <last-frame> ");
    fprintf(stderr,"-h <frame-height> ");
    fprintf(stderr,"-w <frame-width>\n\t");
    fprintf(stderr,"-i <input-files> ");
    fprintf(stderr,"-o <output-files>\n\t");
    fprintf(stderr,"[-s <x-dimension-of-splining-blocks>] ");
    fprintf(stderr,"[-t <y-dim-spl-blks>]\n\t");
    fprintf(stderr,"[-j <x-dim-extra-pixels-for-splining>] ");
    fprintf(stderr,"[-k <y-dim-extra-pixels>]\n\t");
    fprintf(stderr,"[-x <x-dim-of-motion-estimation-blocks>] ");
    fprintf(stderr,"[-y <y-dim-mot-est-blks>]\n\t");
```

```
    fprintf(stderr,"[-c <horizontal-scaling-factor>] ");
    fprintf(stderr,"[-d <vertical-scaling-factor>]\n\t");
    fprintf(stderr,"[-p <number-of-previous-frames-used-as-context>] ");
    fprintf(stderr,"[-l <num-later-frms>]\n\t");
    fprintf(stderr,"[-m <maximum-x-displacement-during-motion-est>] ");
    fprintf(stderr,"[-n <max-y-disp>]\n\t");
    fprintf(stderr,"[-a <x-dimension-spline-degree>] ");
    fprintf(stderr,"[-b <y-dimension-spline-degree>]\n\t");
    fprintf(stderr,"[-z <spline-acceptable-error-factor>] ");
    fprintf(stderr,"[-u <class-of-max-accept-err>]\n\t");
    fprintf(stderr,"[-e <maximum-mean-error-during-motion-est>]\n\t");
    fprintf(stderr,"[-v (verbose)]\n");
}

void Parse(int argc, char** argv)
{
    extern char *optarg; /* Argument to an option */
    int opt; /* Command line option */

    /* Parse the command line options */

    if(argc==1){
        Usage(argv[0]);
        exit(1);
    }
    if(strcpy(input,"") != input){
        fprintf(stderr,"Error when initializing input filename\n");
        exit(1);
    }
    if(strcpy(output,"") != output){
        fprintf(stderr,"Error when initializing output filename\n");
        exit(1);
    }
    while((opt=getopt(argc,argv,
                      "f:g:h:w:i:o:x:y:s:t:j:k:c:d:p:l:m:n:a:b:e:z:u:v"))!=-1){
        switch(opt){
            case 'f':
                begin=atoi(optarg);
                break;
            case 'g':
                last=atoi(optarg);
                break;
            case 'h':
                height=atoi(optarg);
                break;
            case 'w':
                width=atoi(optarg);
                break;
            case 'i':
                strcpy(input,optarg);
                break;
            case 'o':
                strcpy(output,optarg);
                break;
            case 'x':
                XBL=atoi(optarg);
```

```
    break;
case 'y':
    YBL=atoi(optarg);
    break;
case 's':
    XSBL=atoi(optarg);
    break;
case 't':
    YSBL=atoi(optarg);
    break;
case 'j':
    XBR=atoi(optarg);
    break;
case 'k':
    YBR=atoi(optarg);
    break;
case 'c':
    XIF=atoi(optarg);
    break;
case 'd':
    YIF=atoi(optarg);
    break;
case 'p':
    BFM=atoi(optarg);
    break;
case 'l':
    FFM=atoi(optarg);
    break;
case 'm':
    XSM=atoi(optarg);
    break;
case 'n':
    YSM=atoi(optarg);
    break;
case 'a':
    XSPL=atoi(optarg);
    break;
case 'b':
    YSPL=atoi(optarg);
    break;
case 'e':
    ERMC=atoi(optarg);
    break;
case 'u':
    THR=atoi(optarg);
    break;
case 'z':
    ERSP=atoi(optarg);
    break;
case 'v':
    verbose=1;
    break;
case '?':
    Usage(argv[0]);
    exit(1);
default:
```

```
                fprintf(stderr,"Parse error.");
                exit(1);
            }
    }

    /* Check for input, output filenames and that enough frames specified */

    if (strcmp(input,"")==0){
        fprintf(stderr,"Input filename required.\n");
        exit(1);
    }
    if (strcmp(output,"")==0){
        fprintf(stderr,"Output filename required.\n");
        exit(1);
    }
    if((FFM+BFM)>(last-begin)){
        fprintf(stderr,"%d more frames needed.\n",FFM+BFM+begin-last);
        exit(1);
    }
    if(verbose) printf("Options parsed.\n");
}

void Allocate()
{
    int count;

    /* Allocate memory for pointers to frames being processed */

    frame=(unsigned char **) calloc(BFM+1+FFM, sizeof(unsigned char *));
    if(frame==NULL){
        fprintf(stderr,"Memory request for pointers to frames failed.\n");
        exit(1);
    }
    if(verbose) printf("Allocated memory for pointers to frames.\n");

    /* Allocate memory for frames */

    for(count=0;count<BFM+1+FFM;count++){
        frame[count]=(unsigned char*) calloc(height*width,
                                             sizeof(unsigned char));
        if(frame[count]==NULL){
            fprintf(stderr,"Memory request for frame %d failed.\n",count);
            exit(1);
        }
    }
    if(verbose) printf("Allocated memory for frames.\n");

    /* Allocate memory for pointers to motion estimated frames */

    mc=(unsigned char **) calloc(BFM+1+FFM, sizeof(unsigned char *));
    if(mc==NULL){
        fprintf(stderr,"Memory request for pointers to motion ");
        fprintf(stderr,"estimated frames failed.\n");
        exit(1);
    }
    if(verbose){
```

```
    printf("Allocated space for pointers to frames created by ");
    printf("motion estimation searches.\n");
}

/* Allocate memory for frames created by motion estimation searches */

hmc=(height/YBL)*YBL; /* Motion estimation is not performed on border */
wmc=(width/XBL)*XBL;  /* area which has width or height less than that  *
                       * of a single macroblock                          */
if(verbose){
    printf("Input frame dimensions: %d x %d\n",height,width);
    printf("Working with frame dimensions: %d x %d\n",hmc,wmc);
}

for(count=0;count<BFM+1+FFM;count++){
    mc[count]=(unsigned char *) calloc(hmc*wmc, sizeof(unsigned char));
    if(mc[count]==NULL){
        fprintf(stderr,"Memory request for motion estimated frame ");
        fprintf(stderr,"%d failed.\n",count);
        exit(1);
    }
}
if(verbose){
    printf("Allocated memory for frames created by motion ");
    printf("estimation searches.\n");
}

/* Allocate memory for high resolution versions of images */

xhr=(doublereal *) calloc((BFM+1+FFM)*hmc*wmc, sizeof(doublereal));
if(xhr==NULL){
    fprintf(stderr,"Memory request for x coordinates of ");
    fprintf(stderr,"high resolution image failed.\n");
    exit(1);
}
if(verbose){
    printf("Allocated memory for x coordinates of high resolution ");
    printf("frames.\n");
}

yhr=(doublereal *) calloc((BFM+1+FFM)*hmc*wmc, sizeof(doublereal));
if(yhr==NULL){
    fprintf(stderr,"Memory request for y coordinates of ");
    fprintf(stderr,"high resolution image failed.\n");
    exit(1);
}
if(verbose){
    printf("Allocated memory for y coordinates of high resolution ");
    printf("frames.\n");
}

zhr=(doublereal *) calloc((BFM+1+FFM)*hmc*wmc, sizeof(doublereal));
if(zhr==NULL){
    fprintf(stderr,"Memory request for z coordinates of ");
    fprintf(stderr,"high resolution image failed.\n");
    exit(1);
```

```
}
if(verbose){
    printf("Allocated memory for z coordinates of high resolution ");
    printf("frames.\n");
}

hsp=(hmc/YSBL)*YSBL; /* Spline construction is not performed on border */
wsp=(wmc/XSBL)*XSBL; /* area which has width or height less than that  *
                      * of a single spline interpolation block         */
if(verbose) printf("Spline frame dimensions: %d x %d\n",hsp,wsp);

/* Allocate memory used for representing the block as evaluated at *
 * desired points on the spline surface                            */

xf=(doublereal *) calloc(XIF*XSBL, sizeof(doublereal));
if(xf==NULL){
    fprintf(stderr,"Memory request for x coordinate very high ");
    fprintf(stderr,"res block used for spline evaluation failed.\n");
    exit(1);
}
if(verbose){
    printf("Allocated mem for x coordinate very high res ");
    printf("block used for spline eval.\n");
}

yf=(doublereal *) calloc(YIF*YSBL, sizeof(doublereal));
if(yf==NULL){
    fprintf(stderr,"Memory request for y coordinate very high ");
    fprintf(stderr,"res block used for spline evaluation failed.\n");
    exit(1);
}
if(verbose){
    printf("Allocated mem for y coordinate very high res ");
    printf("block used for spline eval.\n");
}

zf=(doublereal *) calloc(XIF*XSBL*YIF*YSBL, sizeof(doublereal));
if(zf==NULL){
    fprintf(stderr,"Memory request for z coordinate very high ");
    fprintf(stderr,"res block used for spline evaluation failed.\n");
    exit(1);
}
if(verbose){
    printf("Allocated mem for z coordinate very high res ");
    printf("block used for spline eval.\n");
}

/* Buffers blocks till a row of blocks have been determined, at which *
 * point they may be outputted to the appropriate file                */

vhrbuf=(doublereal *) calloc(XIF*YIF*wsp*YSBL,
                             sizeof(doublereal));
if(vhrbuf==NULL){
    fprintf(stderr,"Memory request for output buffer of ");
    fprintf(stderr,"very high resolution coordinates failed.\n");
    exit(1);
```

```
    }
    if(verbose){
        printf("Allocated memory for output buffer of very high resolution ");
        printf("coordinates.\n");
    }
}

void ReadInit()
{
    char name[MAX_STR_LEN];
    FILE *stream; /* File pointer multiply used for reads and writes */
    int bytes; /* Return value for number of bytes read */
    int count;

    /* Read initial files in */

    for(count=0;count<BFM+1+FFM;count++){
        sprintf(name,"%s%d.Y",input,begin+count);
        stream=fopen(name,"r");
        if(stream==NULL){
            fprintf(stderr,"Can not open file %s\n",name);
            exit(1);
        }
        bytes=fread(frame[count],sizeof(unsigned char),height*width,stream);
        if(bytes<height*width){
            fprintf(stderr,"Could not read enough data from file %s.\n", name);
            exit(1);
        }
        fclose(stream);
    }
    if(verbose) printf("Read initial files %s%d-%d in.\n",
                       input,begin,begin+BFM+FFM);
}

int CumAbsErr(int xs, int ys, int xb, int yb, int count)
{
    int x,y;
    int error; /* Reused for calcultaing motion estimated block errors  */
    unsigned char *orig,*xtra; /* Pointers to the original frame (whose   *
                               * resolution we are improving), & the one *
                               * from which more data is being obtained  */

    /* Use cumulative absolute error to determine match closeness */

    orig=frame[BFM];
    xtra=frame[count];
    error=0;
    for(y=0;y<YBL;y++){
        for(x=0;x<XBL;x++){
            error+=
              abs(orig[((yb+y)*width)+(xb+x)]-xtra[((ys+y)*width)+(xs+x)]);
        }
    }
    return error;
}
```

```
void UpdateBlock(int xs, int ys, int xb, int yb, int src, int dest)
{
    int x,y;
    unsigned char *xtra,*new; /* Pointers to the frame from which more data *
                              * is being obtained and the one being        *
                              * constructed                                */

  /* Used when motion estimated block has less than maximum *
   * acceptable error, this routine copies the block        */

    xtra=frame[src];
    new=mc[dest];
    for(y=0;y<YBL;y++){
        for(x=0;x<XBL;x++){
            new[((yb+y)*wmc)+(xb+x)]=xtra[((ys+y)*width)+(xs+x)];
        }
    }
    if(verbose){
        printf("Using block (%d,%d) as motion estimated block",xs,ys);
        printf("(%d,%d) in frame %d.\n",xb,yb,dest);
    }
}

void FindRange(int xb, int yb, int *left, int *right, int *top, int *bottom)
{
    /* Specify the range for motion estimation searches */

    if(!XSM){ /* No horizontal range specified so do exhaustive search */
        *left=0;
        *right=wmc-XBL;
    } else { /* Range specified - check that image boundaries are not *
              * crossed                                                */
        if((xb-XSM)<0){
            *left=0;
        } else {
            *left=xb-XSM;
        }
        if((xb+XSM)>(wmc-XBL)){
            *right=wmc-XBL;
        } else {
            *right=xb+XSM;
        }
    }

    if(!YSM){ /* No vertical range specified so do exhaustive search */
        *top=0;
        *bottom=hmc-YBL;
    } else { /* Range specified - check that image boundaries are not *
              * crossed                                                */
        if((yb-YSM)<0){
            *top=0;
        } else {
            *top=yb-YSM;
        }
        if((yb+YSM)>(hmc-YBL)){
            *bottom=hmc-YBL;
```

```
        } else {
            *bottom=yb+YSM;
        }
    }
    if(verbose){
        printf("Search range for block at (%d,%d) is (%d,%d) to (%d,%d).\n",
               xb,yb,*left,*top,*right,*bottom);
    }
}

void FindBlock(int count, int xb, int yb,
               int left, int right, int top, int bottom)
{
    int ermin; /* Reused for keeping track of current best match (min error) */
    int error; /* Reused for calcultaing motion estimated block errors */
    int xs,ys; /* Coordinates of block currently being tried in search */
    int xf,yf; /* Current estimate for final match */

    ermin=ERMC*XBL*YBL; /* Set initial current error to maximum acceptable */

    /* Compare the block under consideration to all possible blocks *
     * in search range to determine best match                      */

    for(ys=top;ys<=bottom;ys++){
        for(xs=left;xs<=right;xs++){
            error=CumAbsErr(xs,ys,xb,yb,count);
            if(error<ermin){
                xf=xs;
                yf=ys;
                ermin=error;
            }
        } /* End of search for best motion estimated match */
    }     /* for current block                               */
    if(ermin<ERMC*XBL*YBL){ /* Found acceptable match */
        UpdateBlock(xf,yf,xb,yb,count,count);
    } else { /* Forced to used data from reference frame */
        UpdateBlock(xb,yb,xb,yb,BFM,count);
    }
    if(verbose) printf("Search for block (%d,%d) in frame %d completed.\n",
                       xb,yb,count);
}

void CopyOrigFrame()
{
    int x,y;
    unsigned char *new, *orig; /* Pointer to the frame that is being *
                                * constructed and the original one   */

    /* Copy the original frame without motion estimation */

    orig=frame[BFM];
    new=mc[BFM];
    for(y=0;y<hmc;y++){
        for(x=0;x<wmc;x++){
            new[(y*wmc)+x]=orig[(y*width)+x];
        }
```

```
    }
    if(verbose) printf("Original frame %d copied to new one.\n",BFM);
}

void FillBlocks(int count)
{
    int xb,yb,left,right,top,bottom; /* Boundaries for motion search */

    /* Fill each block of the currently being constructed motion *
     * estimated frame                                           */

    for(yb=0;yb<hmc;yb+=YBL){
        for(xb=0;xb<wmc;xb+=XBL){
            FindRange(xb,yb,&left,&right,&top,&bottom);
            FindBlock(count,xb,yb,left,right,top,bottom);
        }
    } /* End of construction of pixel granularity motion estimated frame */
    if(verbose) printf("Motion estimated blocks determined for frame %d.\n",
                       count);
}

void FillHighResXBlock(int xb, int yb, int count, doublereal dx)
{
    int x,y;

    /* Increase the precision of x coordinates */

    for(y=yb;y<yb+YBL;y++){
        for(x=xb;x<xb+XBL;x++){ /* vvvvv- bytes in a line */
            xhr[(count*hmc*wmc) + (y*wmc)+x]=x+dx;
        } /*     ^^^^^^^^^^^^^^- bytes in a frame */
    }
    if(verbose){
        printf("Precision of x coords in block (%d,%d) in frame %d updated.\n",
               xb,yb,count);
    }
}

void FillHighResYBlock(int xb, int yb, int count, doublereal dy)
{
    int x,y;

    /* Increase the precision of y coordinates */

    for(y=yb;y<yb+YBL;y++){
        for(x=xb;x<xb+XBL;x++){
            yhr[(count*hmc*wmc)+(y*wmc)+x]=y+dy;
        }
    }
    if(verbose){
        printf("Precision of y coords in block (%d,%d) in frame %d updated.\n",
               xb,yb,count);
    }
}

void CopyHighResZFrame(int count)
```

```
{
    int x,y;
    unsigned char *new; /* Pointer to the frame from which data is being *
                         * obtained                                      */

    /* Increase the precision of z coordinates from integer to doublereal */

    new=mc[count];
    for(y=0;y<hmc;y++){
        for(x=0;x<wmc;x++){
            zhr[(count*hmc*wmc)+(y*wmc)+x]=new[(y*wmc)+x];
        }
    }
    if(verbose){
        printf("Precision of z coordinates of frame %d increased.\n",count);
    }
}

doublereal FindOptimalDx(doublereal dxp, doublereal dxn,
                         int xb, int yb, int count)
{
    int x,y;
    doublereal caep,caen,error; /* Cumulative absolute error */
    doublereal fxy,fx1y,fxdxy,gxy; /* Function values fxy=f(x,y)      *
                                    * fx1y=f(x+-1,y) fxdxy=f(x+-dx,y) */

    /* Use cumulative absolute error to determine match closeness */

    caep=0;
    for(y=yb;y<yb+YBL;y++){
        for(x=xb;x<xb+XBL;x++){
            gxy=zhr[(count*hmc*wmc)+(y*wmc)+x];
            fxy=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
            if(x+1<wmc){
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x+1];
            } else {
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
            }
            fxdxy=fxy+(dxp*(fx1y-fxy));
            caep+=abs(fxdxy-gxy);
        }
    }

    caen=0;
    for(y=yb;y<yb+YBL;y++){
        for(x=xb;x<xb+XBL;x++){
            gxy=zhr[(count*hmc*wmc)+(y*wmc)+x];
            fxy=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
            if(x-1>=0){
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x-1];
            } else {
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
            }
            fxdxy=fxy+(dxn*(fxy-fx1y));
            caen+=abs(fxdxy-gxy);
        }
```

```
    }

    if(caep>caen){ /* Use the subpixel displacement that yields a lower *
                    * cumulative absolute error                         */
        error=dxn;
    } else {
        error=dxp;
    }

    /* Disallow subpixel estimates of magnitude 1 or greater */

    if(abs(dxp)>=1 && abs(dxn)>=1){
        error=0;
    }
    if(abs(dxp)>=1 && abs(dxn)<1){
        error=dxn;
    }
    if(abs(dxp)<1 && abs(dxn)>=1){
        error=dxp;
    }

    return error;
}

doublereal FindOptimalDy(doublereal dyp, doublereal dyn,
                         int xb, int yb, int count)
{
    int x,y;
    doublereal caep,caen,error; /* Cumulative absolute error */
    doublereal fxy,fxy1,fxdxy,fxydy,gxy; /* Function values fxy=f(x,y)    *
                                          * fxy1=f(x,y+-1) fxdxy=f(x+dx,y) *
                                          * fxydy=f(x,y+-dy)               */

    /* Use cumulative absolute error to determine match closeness */

    caep=0;
    for(y=yb;y<yb+YBL;y++){
        for(x=xb;x<xb+XBL;x++){
            gxy=zhr[(count*hmc*wmc)+(y*wmc)+x];
            fxy=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
            if(y+1<hmc){
                fxy1=zhr[(BFM*hmc*wmc)+((y+1)*wmc)+x];
            } else {
                fxy1=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
            }
            fxydy=fxy+(dyp*(fxy1-fxy));
            caep+=abs(fxdxy-gxy);
        }
    }

    caen=0;
    for(y=yb;y<yb+YBL;y++){
        for(x=xb;x<xb+XBL;x++){
            gxy=zhr[(count*hmc*wmc)+(y*wmc)+x];
            fxy=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
            if(y-1>=0){
```

```
                fxy1=zhr[(BFM*hmc*wmc)+((y-1)*wmc)+x];
            } else {
                fxy1=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
            }
            fxydy=fxy+(dyn*(fxy-fxy1));
            caen+=abs(fxydy-gxy);
        }
    }

    if(caep>caen){ /* Use the subpixel displacement that yields a lower *
                    * cumulative absolute error                          */
        error=dyn;
    } else {
        error=dyp;
    }

    /* Disallow subpixel estimates of magnitude 1 or greater */

    if(abs(dyp)>=1 && abs(dyn)>=1){
        error=0;
    }
    if(abs(dyp)>=1 && abs(dyn)<1){
        error=dyn;
    }
    if(abs(dyp)<1 && abs(dyn)>=1){
        error=dyp;
    }

    return error;
}

doublereal FindDx(int xb, int yb, int count)
{
    int x,y;
    doublereal coeff,cnst,dxp,dxn,dx;
    doublereal fxy,fx1y,gxy; /* Function values - fxy=f(x,y) fx1y=f(x+1,y) */

    /* Assuming dx and dy are independent, determine best dx for block *
     * with (coeff x dx) + cnst = 0                                    */

    coeff=0;
    cnst=0;
    for(y=yb;y<yb+YBL;y++){
        for(x=xb;x<xb+XBL;x++){
            gxy=zhr[(count*hmc*wmc)+(y*wmc)+x];
            fxy=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
            if(x+1<wmc){
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x+1];
            } else {
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
            }
            cnst+=(fxy-gxy)*(fx1y-fxy);
            coeff+=((fx1y-fxy)*(fx1y-fxy));
        }
    }
    if(coeff==0){
```

```
        dxp=1; /* dxp=1 indicates no optimal positive displacement found */
    } else {
        dxp=abs(cnst/coeff);
    }

    coeff=0;
    cnst=0;
    for(y=yb;y<yb+YBL;y++){
        for(x=xb;x<xb+XBL;x++){
            gxy=zhr[(count*hmc*wmc)+(y*wmc)+x];
            fxy=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
            if(x-1>=0){
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x-1];
            } else {
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
            }
            cnst+=(fxy-gxy)*(fx1y-fxy);
            coeff+=((fx1y-fxy)*(fx1y-fxy));
        }
    }
    if(coeff==0){
        dxn=-1; /* dxn=-1 indicates no optimal negative displacement found */
    } else {
        dxn=-1*(abs(cnst/coeff));
    }

    dx=FindOptimalDx(dxp,dxn,xb,yb,count);

    if(verbose) printf("dx=%.3f returned for block at (%d,%d) in frame %d.\n",
                       dx,xb,yb,count);
    return dx;
}

doublereal FindDy(doublereal dx, int xb, int yb, int count)
{
    int x,y;
    doublereal coeff,cnst,dyp,dyn,dy;
    doublereal fxy,fxy1,fx1y,fx1y1,fxdxy,fxdxy1,gxy;
        /* Function values - fxy=f(x,y) fx1y=f(x+1,y) fxy1=f(x,y+1) *
         * fx1y1=f(x+1,y+1) fxdxy=f(x+dx,y) fxdxy1=f(x+dx,y+1)      */

    /* Assuming the optimal dx is known, determine best dy for block *
     * with (coeff x dy) + cnst = 0                                   */

    /* Determine what positive y subpixel displacement gives least error */

    coeff=0;
    cnst=0;
    for(y=yb;y<yb+YBL;y++){
        for(x=xb;x<xb+XBL;x++){

            /* Determine g(x,y), f(x,y), f(x,y+1) */

            gxy=zhr[(count*hmc*wmc)+(y*wmc)+x];
            fxy=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
            if(y+1<hmc){
```

```
            fxy1=zhr[(BFM*hmc*wmc)+((y+1)*wmc)+x];
        } else {
            fxy1=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
        }

        /* Determine f(x+-dx,y), f(x+-dx,y+1) */

        if(dx==0){
            fxdxy=fxy;
            fxdxy1=fxy1;
        }
        if(dx>0){
            if(x+1<wmc){
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x+1];
                if(y+1<hmc){
                    fx1y1=zhr[(BFM*hmc*wmc)+((y+1)*wmc)+x+1];
                } else {
                    fx1y1=zhr[(BFM*hmc*wmc)+(y*wmc)+x+1];
                }
            } else {
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
                if(y+1<hmc){
                    fx1y1=zhr[(BFM*hmc*wmc)+((y+1)*wmc)+x];
                } else {
                    fx1y1=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
                }
            }
            fxdxy=fxy+(dx*(fx1y-fxy));
            fxdxy1=fxy1+(dx*(fx1y1-fxy1));
        }
        if(dx<0){
            if(x-1>=0){
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x-1];
                if(y+1<hmc){
                    fx1y1=zhr[(BFM*hmc*wmc)+((y+1)*wmc)+x-1];
                } else {
                    fx1y1=zhr[(BFM*hmc*wmc)+(y*wmc)+x-1];
                }
            } else {
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
                if(y+1<hmc){
                    fx1y1=zhr[(BFM*hmc*wmc)+((y+1)*wmc)+x];
                } else {
                    fx1y1=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
                }
            }
            fxdxy=fxy+(dx*(fxy-fx1y));
            fxdxy1=fxy1+(dx*(fxy1-fx1y1));
        }

        cnst+=(fxdxy-gxy)*(fxdxy1-fxdxy);
        coeff+=((fxdxy1-fxdxy)*(fxdxy1-fxdxy));
    }
}
if(coeff==0){
    dyp=1; /* dyp=1 indicates no optimal positive displacement found */
```

```
} else {
    dyp=abs(cnst/coeff);
}

/* Determine what negative y subpixel displacement gives least error */

coeff=0;
cnst=0;
for(y=yb;y<yb+YBL;y++){
    for(x=xb;x<xb+XBL;x++){

        /* Determine g(x,y), f(x,y), f(x,y-1) */

        gxy=zhr[(count*hmc*wmc)+(y*wmc)+x];
        fxy=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
        if(y-1>=0){
            fxy1=zhr[(BFM*hmc*wmc)+((y-1)*wmc)+x];
        } else {
            fxy1=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
        }

        /* Determine f(x+-dx,y), f(x+-dx,y-1) */

        if(dx==0){
            fxdxy=fxy;
            fxdxy1=fxy1;
        }
        if(dx>0){
            if(x+1<wmc){
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x+1];
                if(y-1>=0){
                    fx1y1=zhr[(BFM*hmc*wmc)+((y-1)*wmc)+x+1];
                } else {
                    fx1y1=zhr[(BFM*hmc*wmc)+(y*wmc)+x+1];
                }
            } else {
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
                if(y-1>=0){
                    fx1y1=zhr[(BFM*hmc*wmc)+((y-1)*wmc)+x];
                } else {
                    fx1y1=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
                }
            }
            fxdxy=fxy+(dx*(fx1y-fxy));
            fxdxy1=fxy1+(dx*(fx1y1-fxy1));
        }
        if(dx<0){
            if(x-1>=0){
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x-1];
                if(y-1>=0){
                    fx1y1=zhr[(BFM*hmc*wmc)+((y-1)*wmc)+x-1];
                } else {
                    fx1y1=zhr[(BFM*hmc*wmc)+(y*wmc)+x-1];
                }
            } else {
                fx1y=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
```

```
                    if(y-1>=0){
                        fx1y1=zhr[(BFM*hmc*wmc)+((y-1)*wmc)+x];
                    } else {
                        fx1y1=zhr[(BFM*hmc*wmc)+(y*wmc)+x];
                    }
                }
                fxdxy=fxy+(dx*(fxy-fx1y));
                fxdxy1=fxy1+(dx*(fxy1-fx1y1));
            }

            cnst+=(fxdxy-gxy)*(fxdxy-fxdxy1);
            coeff+=((fxdxy-fxdxy1)*(fxdxy-fxdxy1));
        }
    }
    if(coeff==0){
        dyn=-1; /* dyn=-1 indicates no optimal negative displacement found */
    } else {
        dyn=-1*(abs(cnst/coeff));
    }

    dy=FindOptimalDy(dyp,dyn,xb,yb,count);

    if(verbose) printf("dy=%.3f returned for block at (%d,%d) in frame %d.\n",
                       dy,xb,yb,count);
    return dy;
}

void SubPixel(int count)
{
    int xb,yb;
    doublereal dx,dy; /* Sub-pixel motion re-estimation differences */

    /* Place the pixel granularity motion estimated image in *
     * high resolution z coordinate array                    */

    CopyHighResZFrame(count);

    /* Perform sub-pixel granularity motion estimation re-estimation */

    for(yb=0;yb<hmc;yb+=YBL){
        for(xb=0;xb<wmc;xb+=XBL){
            dx=FindDx(xb,yb,count);
            dy=FindDy(dx,xb,yb,count);
            FillHighResXBlock(xb,yb,count,dx);
            FillHighResYBlock(xb,yb,count,dy);
        }
    }
    if(verbose){
        printf("Sub-pixel motion re-estimation for frame %d done.\n",count);
    }
}

void ConstructRef()
{
    int xb,yb;
```

```
    /* Construct reference (original) frame in motion estimated *
     * as well as high resolution arrays                        */

    CopyOrigFrame();
    for(yb=0;yb<hmc;yb+=YBL){
        for(xb=0;xb<wmc;xb+=XBL){
            FillHighResXBlock(xb,yb,BFM,0);
            FillHighResYBlock(xb,yb,BFM,0);
        }
    }
    CopyHighResZFrame(BFM);
    if(verbose) printf("Constructed high resolution reference frame.\n");
}

void ExtractTemporalContext(int done)
{
    int count;

    /* Loop through once for each motion estimated frame constructed */

    ConstructRef();
    for(count=0;count<BFM+1+FFM;count++){
        if (count != BFM){ /* Do pixel and sub-pixel motion estimation *
                           * only if this is not the original frame   */
            FillBlocks(count);
            SubPixel(count);
        }
    } /* End of construction of set of frames */
    if(verbose){
        printf("All motion estimated frames for high resolution ");
        printf("version of frame %d created.\n",(BFM+done));
    }
}

void SplineRange(integer line, integer block,
                 integer *left, integer *right, integer *top, integer *bottom)
{
    /* Given the spline block position, figure out the edges of the range *
     * to be used for spline construction - needed due to extra border    */

    if((block-XBR)<0){
        *left=0;
    } else {
        *left=block-XBR;
    }
    if((block+XSBL+XBR)>wmc){
        *right=wmc;
    } else {
        *right=block+XSBL+XBR;
    }

    if((line-YBR)<0){
        *top=0;
    } else {
        *top=line-YBR;
    }
```

```
    if((line+YSBL+YBR)>hmc){
        *bottom=hmc;
    } else {
        *bottom=line+YSBL+YBR;
    }

    if(verbose){
        printf("Spline range for block at (%d,%d) determined.\n",block,line);
        printf("Boundaries - left: %d, right: %d, top: %d, bottom: %d\n",
               *left,*right,*top,*bottom);
    }
}

void SplineAllocate(integer start, integer end,
                    integer left, integer right, integer top, integer bottom)
{
    /* Allocate memory to hold data set used to construct spline surface */

    xvhr=(doublereal *) calloc((end-start)*(right-left)*(bottom-top),
                               sizeof(doublereal));
    if(xvhr==NULL){
        fprintf(stderr,"Memory request for x coordinate very high ");
        fprintf(stderr,"res block used for spline construction failed.\n");
        exit(1);
    }
    if(verbose){
        printf("Allocated mem for x coord very hi res ");
        printf("block used for spline construction.\n");
    }

    yvhr=(doublereal *) calloc((end-start)*(right-left)*(bottom-top),
                               sizeof(doublereal));
    if(yvhr==NULL){
        fprintf(stderr,"Memory request for y coordinate very high ");
        fprintf(stderr,"res block used for spline construction failed.\n");
        exit(1);
    }
    if(verbose){
        printf("Allocated mem for y coord very hi res ");
        printf("block used for spline construction.\n");
    }

    zvhr=(doublereal *) calloc((end-start)*(right-left)*(bottom-top),
                               sizeof(doublereal));
    if(zvhr==NULL){
        fprintf(stderr,"Memory request for z coordinate very high ");
        fprintf(stderr,"res block used for spline construction failed.\n");
        exit(1);
    }
    if(verbose){
        printf("Allocated mem for z coord very hi res ");
        printf("block used for spline construction.\n");
    }

    /* Allocate array to contain weights of points used in spline *
     * construction                                               */
```

```
    w=(doublereal *) calloc((end-start)*(right-left)*(bottom-top),
                            sizeof(doublereal));
    if(w==NULL){
        fprintf(stderr,"Memory request for array of weights for surface ");
        fprintf(stderr,"fitting failed.\n");
        exit(1);
    }
    if(verbose){
        printf("Allocated mem for weights used for spline construction.\n");
    }
}

void SplineDataFill(integer line, integer block, integer start, integer end,
                    integer left, integer right, integer top, integer bottom)
{
    integer count,x,y;

    /* Fill in weights for splining */

    for(count=0;count<(end-start)*(right-left)*(bottom-top);count++)
      w[count]=1;
    if(verbose){
        printf("Allocated the weights for splining block at (%d,%d)\n",
               block,line);
    }

    /* Fill in the data used for constructing spline surface */

    for(count=0;count<(end-start);count++){
        for(y=top;y<bottom;y++){
            for(x=left;x<right;x++){
                xvhr[(count*(right-left)*(bottom-top))+
                     ((y-top)*(right-left))+(x-left)]
                       =xhr[((start+count)*hmc*wmc)+(y*wmc)+x];
                yvhr[(count*(right-left)*(bottom-top))+
                     ((y-top)*(right-left))+(x-left)]
                       =yhr[((start+count)*hmc*wmc)+(y*wmc)+x];
                zvhr[(count*(right-left)*(bottom-top))+
                     ((y-top)*(right-left))+(x-left)]
                       =zhr[((start+count)*hmc*wmc)+(y*wmc)+x];
            }
        }
    }
    if(verbose){
        printf("Copied data for splining block at (%d,%d).\n",
               block,line);
    }
}

doublereal SplineBlockMSE(integer line, integer block, integer start,
        integer end, integer left, integer right, integer top, integer bottom)
{
    integer count,x,y;
    doublereal total=0,mean,mse;
```

```
    /* Compute the mean */

    for(count=0;count<(end-start);count++){
        for(y=top;y<bottom;y++){
            for(x=left;x<right;x++){
                total+=zvhr[(count*(right-left)*(bottom-top))+
                            ((y-top)*(right-left))+(x-left)];
            }
        }
    }

    mean=total/((end-start)*(bottom-top)*(right-left));

    /* Compute the MSE */

    total=0;

    for(count=0;count<(end-start);count++){
        for(y=top;y<bottom;y++){
            for(x=left;x<right;x++){
                total+=pow((zvhr[(count*(right-left)*(bottom-top))+
                            ((y-top)*(right-left))+(x-left)]-mean),2.0);
            }
        }
    }

    mse=total/((end-start)*(bottom-top)*(right-left));

    if(verbose){
        printf("MSE for block at (%d,%d) is %.3f.\n",block,line,mse);
    }

    return mse;
}

void Spline(int line, int block)
{
    integer temporal=1; /* This indicates that temporal context is used */

    integer count,km,ne,b1,b2,left,right,top,bottom,thrsh;

    integer iopt,m;
    doublereal xb,xe,yb,ye;
    integer kx,ky;
    doublereal s;
    integer nxest,nyest,nmax;
    doublereal epspar1,epspar2,eps;
    integer nx,ny;
    doublereal *tx,*ty,*c,fp,*wrk1,*wrk2;
    integer lwrk1,lwrk2,*iwrk,kwrk,ier;

    integer mx,my;
    integer lwrkev,kwrkev,*iwrkev;
    doublereal *wrkev;

    doublereal num,den;
```

```
/* Set up the error tolerance for spline construction */

epspar1=10;
epspar2=-20;
eps=pow(epspar1,epspar2);

/* Set knot determination to be done automatically */

iopt=0;

/* Figure out edges of spline being constructed */

SplineRange(line,block,&left,&right,&top,&bottom);

xb=left-0.5;
xe=right+0.5;
yb=top-0.5;
ye=bottom+0.5;

/* Allocate memory for very high resolution version of block used for *
 * constructing spline surface                                        */

SplineAllocate(0,(BFM+1+FFM),left,right,top,bottom);

/* Set up parameters independent of spline degree, for constructing *
 * the spline surface                                               */

thrsh=THR+1;
m=(BFM+1+FFM)*(right-left)*(bottom-top);

c=(doublereal *) calloc(m, sizeof(doublereal));
if(c==NULL){
    fprintf(stderr,"Memory request for array of spline coefficients ");
    fprintf(stderr,"failed.\n");
    exit(1);
}

kx=XSPL;
ky=YSPL;

while((kx>0) && (ky>0) && (thrsh>THR)){

    /* Copy data used to construct the spline surface at (block,line) */

    SplineDataFill(line,block,0,(BFM+1+FFM),left,right,top,bottom);

    /* Scale the smoothing factor according to the variation in the data */

    s=ERSP*m*SplineBlockMSE(line,block,BFM,BFM+1,left,right,top,bottom)/16;
    if(verbose){
      printf("Using smoothing constraint of %.3f for block at (%d,%d).\n",
             s,block,line);
    }

    /* Set up parameters dependent on spline degree */
```

```
nxest=kx+1+sqrt(m);
nyest=ky+1+sqrt(m);
nmax=nxest+nyest;
nx=kx*(right-left);
ny=ky*(bottom-top);
tx=(doublereal *) calloc(nmax, sizeof(doublereal));
if(tx==NULL){
    fprintf(stderr,"Memory request for array of x dimension knots ");
    fprintf(stderr,"failed.\n");
    exit(1);
}
ty=(doublereal *) calloc(nmax, sizeof(doublereal));
if(ty==NULL){
    fprintf(stderr,"Memory request for array of y dimension knots ");
    fprintf(stderr,"failed.\n");
    exit(1);
}
if((kx*nyest+ky)>(ky*nxest+kx)){
    b1=ky*nxest+kx+1;
    b2=b1+nxest;
} else {
    b1=kx*nyest+ky+1;
    b2=b1+nyest;
}
km=kx+ky+1;
ne=nxest+nyest;
lwrk1=(nxest*nyest*(2+b1+b2))+(2*(nxest+nyest+km*(m+ne)+ne))+b2+1;
wrk1=(doublereal *) calloc(lwrk1, sizeof(doublereal));
if(wrk1==NULL){
    fprintf(stderr,"Memory request for first real workspace array ");
    fprintf(stderr,"failed in spline interpolation.\n");
    exit(1);
}
lwrk2=(nxest*nyest*(b2+1))+b2;
wrk2=(doublereal *) calloc(lwrk2, sizeof(doublereal));
if(wrk2==NULL){
    fprintf(stderr,"Memory request for second real workspace array ");
    fprintf(stderr,"failed in spline interpolation.\n");
    exit(1);
}
kwrk=m+(nxest*nyest);
iwrk=(integer *) calloc(kwrk, sizeof(integer));
if(iwrk==NULL){
    fprintf(stderr,"Memory request for integer workspace array ");
    fprintf(stderr,"failed in spline interpolation.\n");
    exit(1);
}

surfit_(&iopt,&m,xvhr,yvhr,zvhr,w,&xb,&xe,&yb,&ye,&kx,&ky,&s,&nxest,
        &nyest,&nmax,&eps,&nx,tx,&ny,ty,c,&fp,wrk1,&lwrk1,wrk2,
        &lwrk2,iwrk,&kwrk,&ier);
if(verbose){
    printf("Surface fit at (%d,%d) done with splines of deg (%d,%d). ",
           block,line,kx,ky);
    printf("Return value: %d\n",ier);
```

```
    }

    if(ier>0){
        fprintf(stderr,"Error when constructing spline surface - ");
        fprintf(stderr,"fp: %.2e, s: %.2e\n",fp,s);
    }
    thrsh=ier;

    /* Free the temporarily used arrays */

    free(wrk1);
    free(wrk2);
    free(iwrk);

    /* Free these arrays if another pass is needed */

    if(thrsh>THR){
        free(tx);
        free(ty);
    }

    /* Use lower order splines */

    kx-=2;
    ky-=2;
}

if(thrsh>THR){

  temporal=0; /* Temporal context is no longer being used */

  /* Free these arrays so they may be used for spline surface *
   * construction using only the reference frame data         */

  free(xvhr);
  free(yvhr);
  free(zvhr);

  free(w);
  free(c);

  /* Allocate memory for very high resolution version of block used for *
   * constructing spline surface                                        */

  SplineAllocate(BFM,BFM+1,left,right,top,bottom);

  /* Set up parameters independent of spline degree, for constructing *
   * the spline surface                                               */

  thrsh=THR+1;
  m=(right-left)*(bottom-top);

  c=(doublereal *) calloc(m, sizeof(doublereal));
  if(c==NULL){
      fprintf(stderr,"Memory request for array of interpolation ");
      fprintf(stderr,"coefficients failed.\n");
```

```
    exit(1);
}

kx=XSPL;
ky=YSPL;

while((kx>0) && (ky>0) && (thrsh>THR)){

    /* Copy data used to construct the spline surface at (block,line) */

    SplineDataFill(line,block,BFM,BFM+1,left,right,top,bottom);

    /* Scale the smoothing factor according to variation in data */

    s=ERSP*m*
      SplineBlockMSE(line,block,BFM,BFM+1,left,right,top,bottom)/16;
    if(verbose){
      printf("Using smoothing const of %.3f for block at (%d,%d).\n",
             s,block,line);
    }

    /* Set up parameters dependent on spline degree */

    nxest=kx+1+sqrt(m);
    nyest=ky+1+sqrt(m);
    nmax=nxest+nyest;
    nx=kx*(right-left);
    ny=ky*(bottom-top);
    tx=(doublereal *) calloc(nmax, sizeof(doublereal));
    if(tx==NULL){
        fprintf(stderr,"Memory request for array of x dimension ");
        fprintf(stderr,"interpolation knots failed.\n");
        exit(1);
    }
    ty=(doublereal *) calloc(nmax, sizeof(doublereal));
    if(ty==NULL){
        fprintf(stderr,"Memory request for array of y dimension ");
        fprintf(stderr,"interpolation knots failed.\n");
        exit(1);
    }
    if((kx*nyest+ky)>(ky*nxest+kx)){
        b1=ky*nxest+kx+1;
        b2=b1+nxest;
    } else {
        b1=kx*nyest+ky+1;
        b2=b1+nyest;
    }
    km=kx+ky+1;
    ne=nxest+nyest;
    lwrk1=(nxest*nyest*(2+b1+b2))+(2*(nxest+nyest+km*(m+ne)+ne))+b2+1;
    wrk1=(doublereal *) calloc(lwrk1, sizeof(doublereal));
    if(wrk1==NULL){
        fprintf(stderr,"Memory request for first real workspace array ");
        fprintf(stderr,"failed in interpolation.\n");
        exit(1);
    }
```

```
      lwrk2=(nxest*nyest*(b2+1))+b2;
      wrk2=(doublereal *) calloc(lwrk2, sizeof(doublereal));
      if(wrk2==NULL){
          fprintf(stderr,"Memory request for second real workspace ");
          fprintf(stderr,"array failed in interpolation.\n");
          exit(1);
      }
      kwrk=m+(nxest*nyest);
      iwrk=(integer *) calloc(kwrk, sizeof(integer));
      if(iwrk==NULL){
          fprintf(stderr,"Memory request for integer workspace array ");
          fprintf(stderr,"failed in interpolation.\n");
          exit(1);
      }

      surfit_(&iopt,&m,xvhr,yvhr,zvhr,w,&xb,&xe,&yb,&ye,&kx,&ky,&s,&nxest,
              &nyest,&nmax,&eps,&nx,tx,&ny,ty,c,&fp,wrk1,&lwrk1,wrk2,
              &lwrk2,iwrk,&kwrk,&ier);
      if(verbose){
          printf("Surf fit at (%d,%d) done with interpol of deg (%d,%d). ",
                 block,line,kx,ky);
          printf("Return value: %d\n",ier);
      }

      if(ier>0){
          fprintf(stderr,"Error when constructing interpol surf - ");
          fprintf(stderr,"fp: %.2e, s: %.2e\n",fp,s);
      }
      thrsh=ier;

      /* Free the temporarily used arrays */

      free(wrk1);
      free(wrk2);
      free(iwrk);

      /* Free these arrays if another pass is needed */

      if(thrsh>THR){
          free(tx);
          free(ty);
      }

      /* Use lower order splines */

      kx-=2;
      ky-=2;
  }
}

/* In case of unrecoverable splining error, zero out the block */

if(thrsh>THR){
    for(count=0;count<XIF*YIF*XSBL*YSBL;count++){
        zf[count]=0;
    }
```

```
}

/* Spline surface routine exits with kx,ky set to 2 below *
 * the degree that the last construction was performed at */

kx+=2;
ky+=2;

if(thrsh<=THR){

    /* Set up the parameters for evaluating the spline surface */

    den=XIF;
    for(count=0;count<XIF*XSBL;count++){
        num=count;
        xf[count]=block+(num/den);
    }
    den=YIF;
    for(count=0;count<YIF*YSBL;count++){
        num=count;
        yf[count]=line+(num/den);
    }
    if(verbose){
        printf("Created very high res x-y grid for block at (%d,%d).\n",
               block,line);
    }

    mx=XIF*XSBL;
    my=YIF*YSBL;
    lwrkev=2*(mx*(kx+1)+my*(ky+1));
    wrkev=(doublereal *) calloc(lwrkev, sizeof(doublereal));
    if(wrkev==NULL){
        fprintf(stderr,"Memory request for real workspace array ");
        fprintf(stderr,"failed in spline evaluation.\n");
        exit(1);
    }
    kwrkev=2*(mx+my);
    iwrkev=(integer *) calloc(kwrkev, sizeof(integer));
    if(iwrkev==NULL){
        fprintf(stderr,"Memory request for integer workspace array ");
        fprintf(stderr,"failed in spline evaluation.\n");
        exit(1);
    }

    /* Evaluate the spline surface */

    bispev_(tx,&nx,ty,&ny,c,&kx,&ky,xf,&mx,yf,&my,zf,wrkev,&lwrkev,
            iwrkev,&kwrkev,&ier);
    if(verbose){
        printf("Very hi res evaluation for block at (%d,%d) done. ",
               block,line);
        printf("Return value: %d\n",ier);
    }
    if(ier>0){
        fprintf(stderr,"Error when evaluating spline surface.\n");
    }
```

```
        /* Free the temporarily used arrays */

        free(tx);
        free(ty);

        free(wrkev);
        free(iwrkev);
    }

    /* Cross out diagonal to show no temporal context used */

    if(!temporal){
        for(count=0;count<(YIF*YSBL);count++){
            if(count<(XIF*XSBL)){
                zf[(count*YIF*YSBL)+count]=0;
            }
        }
    }

    free(xvhr);
    free(yvhr);
    free(zvhr);

    free(w);
    free(c);
}

void ConstructBuf(int line)
{
    int block,x,y;

    for(block=0;block<wsp;block+=XSBL){
        Spline(line,block);

        /* Copy the high resolution evaluation of spline surface into buffer *
         * and flip x and y coordinates back since bispev_ gives data with   *
         * coordinates switched                                              */

        for(y=0;y<YIF*YSBL;y++){
            for(x=0;x<XIF*XSBL;x++){
                vhrbuf[(block*XIF*YIF*YSBL)+(x*YIF*YSBL)+y]=zf[(y*XIF*XSBL)+x];
            }
        }
    }
}

void GenerateFrame(int done)
{
    char name[MAX_STR_LEN];
    FILE *stream; /* File pointer multiply used for reads and writes */
    int bytes; /* Return value for number of bytes read */
    int line,count,base,x,y;
    unsigned char *lrbuf;

    sprintf(name,"%s%d.Y",output,done);
```

```
    lrbuf=(unsigned char *) calloc(XIF*YIF*wsp*YSBL, sizeof(unsigned char));
    if(lrbuf==NULL){
        fprintf(stderr,"Memory allocation for low resolution buffer of ");
        fprintf(stderr,"frames used to create output file %s failed.\n",name);
        exit(1);
    }

    /* Output the final high resolution frame */

    stream=fopen(name, "w");
    if(stream==NULL){
        fprintf(stderr,"Can not write to file %s\n",name);
        exit(1);
    }

    for(line=0;line<hsp;line+=YSBL){
        ConstructBuf(line);
        count=0;
        for(y=0;y<YIF*YSBL;y++){
            for(base=y*XIF*XSBL;base<XIF*YIF*wsp*YSBL;base+=XIF*YIF*XSBL*YSBL){
                for(x=0;x<XIF*XSBL;x++){
                    lrbuf[count]=vhrbuf[base+x];
                    count++;
                }
            }
        }
        bytes=fwrite(lrbuf,sizeof(unsigned char),XIF*YIF*wsp*YSBL,stream);
        if(bytes<XIF*YIF*wsp*YSBL){
            fprintf(stderr,"Error when writing to file %s.\n", name);
            exit(1);
        }
        fflush(stream);
    }
    fclose(stream);
    if(verbose) printf("Wrote file %s.\n",name);
    free(lrbuf);
}

void ReadFile(int done)
{
    char name[MAX_STR_LEN];
    FILE *stream; /* File pointer multiply used for reads and writes */
    int bytes; /* Return value for number of bytes read */
    int count;

    /* Skip this section when the last input file is reached - since there *
     * is no new data to be loaded - otherwise load new file               */

    if(done<(last-begin-FFM-BFM)){

        sprintf(name,"%s%d.Y",input,begin+BFM+1+FFM+done);

        /* Shift the pointers so that the oldest frame is thrown out */

        free(frame[0]);
```

```
        for(count=0;count<BFM+FFM;count++){
            frame[count]=frame[count+1];
        }
        frame[BFM+FFM+1]=(unsigned char *) calloc(height*width,
                                                  sizeof(unsigned char));
        if(frame[BFM+FFM+1]==NULL){
            fprintf(stderr,"Memory request for new frame being read in ");
            fprintf(stderr,"from file %s failed.\n",name);
            exit(1);
        }
        if(verbose) printf("Frame pointers updated.\n");

        /* Read in the next input file */

        stream=fopen(name, "r");
        if(stream==NULL){
            fprintf(stderr,"Can not open file %s\n",name);
            exit(1);
        }
        bytes=fread(frame[BFM+FFM+1],
                    sizeof(unsigned char),height*width,stream);
        if(bytes<height*width){
            fprintf(stderr,"Could not read enough data from file %s.\n", name);
            exit(1);
        }
        fclose(stream);
        if(verbose) printf("Read file %s.\n",name);
    }
}

void OutputFrames()
{
    int done; /* Number of higher resolution frames that have been outputted */
    char message[MAX_STR_LEN];

    /* Loop through once for each high resolution frame created */

    for(done=0;done<(last-begin-FFM-BFM+1);done++){
        sprintf(message,"Temporal context extraction of frame %d starting at",
                done);
        CurrentTime(message);
        ExtractTemporalContext(done);
        sprintf(message,"Temporal context extraction of frame %d ending at",
                done);
        CurrentTime(message);
        sprintf(message,"Spline generation of frame %d starting at",done);
        CurrentTime(message);
        GenerateFrame(done);
        sprintf(message,"Spline generation of frame %d ending at",done);
        CurrentTime(message);
        ReadFile(done);
    }
}

void main(int argc, char** argv)
{
```

```
    char message_start[]="Video sequence processing starting at";
    char message_end[]="Video sequence processing ending at";

    Parse(argc, argv);
    CurrentTime(message_start);
    Allocate();
    ReadInit();
    OutputFrames();
    CurrentTime(message_end);
}
```

Bibliography

[BBH+89] Burt, Bergen, Hingorani, Kolczynski, Lee, Leung, Lubin, and Shvayster. Object tracking with a moving camera - an application of dynamic motion analysis. *Proceedings of the IEEE*, 1989.

[BK96] Vasudev Bhaskaran and Konstantinos Konstantinides. *Image and Video Compression Standards: Algorithms and Architectures.* Kulwer Academic Publishers, Boston, MA, 1996.

[Chi95] Leonardo Chiariglione. The development of an integrated audiovisual coding standard: Mpeg. *Proceedings of the IEEE*, 1995.

[CK94] Ward Cheney and David Kincaid. *Numerical Mathematics and Computing.* Brooks/Cole Publishing Company, Pacific Grove, CA, 1994.

[CLL+95] Kiran Challapali, Xavier Lebegue, Jae Lim, Woo Paik, Regis Saint Girons, Eric Petajan, Vinay Sathe, Paul Snopko, and Joel Zdepski. The grand alliance for us hdtv. *Proceedings of the IEEE*, 1995.

[dB] Carl de Boor. B-spline basics.

[dB78] Carl de Boor. *A Practical Guide to Splines.* Springer-Verlag, New York NY, 1978.

[dBH87] Carl de Boor and Höllig. B-splines without divided differences. *Geometric Modeling*, 1987.

[Die93] Paul Dierckx. *Curve and Surface Fitting with Splines.* Oxford University Press, New York, NY, 1993.

[Gal91] Didier Le Gall. Mpeg: A video compression standard for multimedia applications. *Communications of the ACM*, 1991.

[Mar93] Tassos Markas. *Data Compression: Algorithms and Architectures.* PhD thesis, Duke University, 1993.

[PDG95] Peter Pirsch, Nicolas Demassieux, and Winfried Gehrke. Vlsi architectures for video compression - a survey. *Proceedings of the IEEE*, 1995.

[PG89] Cornel Pokorny and Curtis Gerald. *Computer Graphics: The Principles Behind the Art and Science.* Franklin, Beedle and Associates, Irvine, CA,

1989.

[PK] Conrad Poelman and Takeo Kanade. A paraperspective factorization method for shape and motion recovery.

[Sch81] Larry Schumaker. *Spline Functions Basic Theory.* John Wiley and Sons, New York, NY, 1981.

[TK90] Carlo Tomasi and Takeo Kanade. Shape and motion from image streams: A factorization method - planar motion. Technical report, Carnegie Mellon University, 1990.

[TK91] Carlo Tomasi and Takeo Kanade. Shape and motion from image streams: A factorization method - point features in 3d motion. Technical report, Carnegie Mellon University, 1991.

[TK92a] Carlo Tomasi and Takeo Kanade. Shape and motion from image streams: A factorization method - full report on the orthographic case. Technical report, Carnegie Mellon University, 1992.

[TK92b] Carlo Tomasi and Takeo Kanade. Shape and motion from image streams under orthography: A factorization method. *International Journal of Computer Vision*, 1992.

[Tom93] Carlo Tomasi. Pictures and trails: A new framework for the computation of shape and motion from perspecitve image sequences. Technical report, Cornell University, 1993.

[WC92] Ting-Hu Wu and Rama Chellapa. Experiments on estimating motion and structure parameters using long monocular image sequences. Technical report, University of Maryland, College Park, 1992.

[WWY] Kang-Wen Wang, Ru-Li Wang, and Yi-Xun Yan. Optoelectronic hybrid system for the full-search block-matching motion compensation algorithm.

Index