An Edge Matching Approach for Video Motion Estimation

by

Ben Barrow

A thesis submitted in partial satisfaction
of the requirements for the degree
Bachelors of Science in Computer Science

2010

Advisor: Michael Main, Associate Professor, University of Colorado

Thesis Committee: Andrzej Ehrenfeucht, Distinguished Professor, University of Colorado
Robert Frohardt, PhD Candidate, University of Colorado
Michael Main, Associate Professor, University of Colorado
Jane Mulligan, Research Assistant Professor, University of Colorado

Abstract

Motion estimation is an important problem in computer vision. It has applications in video compression, artificial intelligence, stereo matching, 3D scene reconstruction from video, and a number of other areas. I introduce the motion estimation problem and list potential difficulties that a solution must overcome. I then describe previous work related to edge based motion estimation and present a new algorithm to the reader. Finally, I discuss my results and possible improvements that may be made to my algorithm.

1 Introduction

I propose an 4-step algorithm for extracting motion data from a pair of consecutive video frames. In logical order, edges are identified and matched across frames, then using those edges, motion vectors are calculated and lastly interpolated. For the final step, I describe a new method for the interpolation of data in multiple dimensions from irregularly spaced known data points. My algorithm successfully processes a wide selection of image types, and using an average local minimum error analysis to measure results, I found the results to be mixed depending on image complexity and other factors. In general, I was unsatisfied with some of the results and first suspected an interpolation error. But, detailed testing now indicates a failure in the edge processing, so I propose three improvements to my original algorithm.

1.1 What is Motion Estimation

In computer vision, motion estimation is the problem of taking two or more consecutive video frames and estimating how individual pixels or small groups of pixels move from frame to frame. Here, I restrict the problem to finding the motion from one frame to the next, using only those two frames as input. Ideally, an accurate motion vector will be computed for every pixel in the first frame for which motion is defined.

1.2 Applications of Motion Estimation

Motion data is useful in a number of different areas. One of the most wide spread uses of motion data is in video compression.[6] Most modern video compression algorithms do not store every frame of the video as a full image. Instead, they typically store only a few percent of the frames in a video as full images (when a frame is stored as a full image, it is called a key frame). The non-key frames are then approximated by distorting key frames or other previously computed frames using motion data that is stored in the video file. After generating a first approximation of the non-key frame using motion data, the quality of the frame is then improved by subtracting an error image that is also stored in the video file.[11][12]

The motion estimation problem is very similar to the stereo matching problem.[1][6] In stereo matching, however, additional constraints may be placed on the generated vectors. With stereo matching, it is much more likely that the relative positions and orientations of the cameras are known. If this extra information is known and if the two images were taken at the same time, the programmer may use the epipolar constraint to change the problem from a 2-dimensional problem to a 1-dimensional problem.[1]

Given enough motion information, it is also possible to estimate the motion of the camera.[13] With this information and the assumption that no objects in the scene move relative to each other, the motion data can be used to construct a 3-dimensional model of the scene.

Motion data can also be used to track objects in a video or sequence of images.

2 Challenging Aspects of Motion Estimation

There are a number of reasons why computing accurate motion data is difficult. Here, I list and describe a few of those reasons.

2.1 The Aperture Problem

Using only local information, the motion of a pixel often cannot be unambiguously determined. Figure 1 demonstrate this phenomenon.[1]

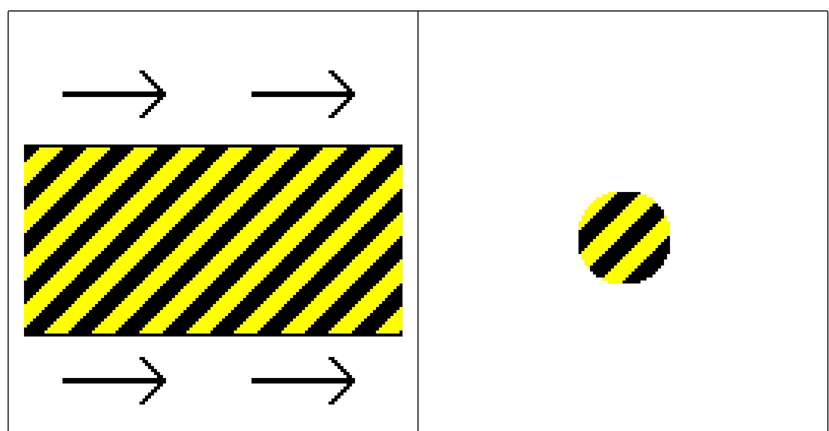


Figure 1: The left image is a full frame from a video and the right image shows only a small region around the center of that frame. The arrows indicate the direction of motion of the full frame.

Although we're only looking at a single frame here, Figure 1 demonstrates that it is impossible to determine the true direction of motion using only a small part of the full frame. In general for very small apertures and black-and-white images, only the component of motion in the direction of the image gradient can be computed accurately.

2.2 Discontinuities in the Motion Vector Field

Discontinuities in the motion vector field can also cause problems for a motion estimation algorithm.[2] Many motion estimation algorithms rely on the assumption that the motion vector field is continuous.

Discontinuities in a motion vector field are typically caused when portions of the scene are uncovered and are not visible in the first frame of the sequence.

2.3 Undefined/Non-Existent Regions in the Motion Vector Field

Undefined regions in the motion vector field can make motion estimation difficult for essentially the same reason that discontinuities are difficult to handle.[2]

Regions of undefined motion are in some ways similar to discontinuities in the motion vector field. These regions occur when portions of the first frame are not visible in the other frames of the sequence. If we're only using two frames to do the motion estimation, undefined regions in the motion from frame 1 to frame 2 correspond to discontinuities in the motion from frame 2 to frame 1.

2.4 Image Noise

As with most algorithms, errors/noise in the input data can reduce the quality of the output data. There are two main sources of noise in images that will be used by a motion estimation algorithm. The first is noise generated by the camera's detector. Additional noise can also be caused by video compression. Since video compressors are usually lossy and typically use motion data to reduce the amount of data that needs to be stored, errors that were made by the video compressor's motion estimator can be imprinted into the video. This imprint increases the chance that a later motion estimator will observe motion in the direction that the motion is stored in the video file.

2.5 Repeating Patterns and Flat Areas

Regions with repeating patterns or almost constant color value can be difficult for a motion estimation algorithm to handle. Flat areas are difficult to handle, because they contain very few features that are needed to match pixels between multiple frames. Similarly, regions with repeating patterns are difficult to handle, because even though they may contain many features, the features cannot be easily matched uniquely between frames.

2.6 Non-Lambertian Surfaces and Changes in Lighting

Most motion estimation algorithms rely on the assumption that all surfaces in the scene are Lambertian. Lambertian surfaces are surfaces that reflect light in a way that causes the surface to look the same brightness (and color) when viewed from any direction.[14] If a surface's appearance depends on the angle that the surface is viewed from, estimating the motion of an individual point on that surface can be difficult. Similarly, matching can also be difficult if the lighting on a surface changes from frame to frame.

2.7 Non-Constant Motion

A common assumption used by most motion estimation algorithms is that the motion changes slowly as a function of position in the motion vector field. Algorithms that use more that two frames may also rely on the assumption that the motion of each point does not change much in time.

2.8 Large Motion

Lastly, it is often assumed that the motion vectors are small (that is, points don't move too quickly in the images). Thus, many motion estimation algorithms have trouble dealing with points that move too quickly.

3 Evaluating a Motion Estimation Algorithm

Once a motion estimation algorithm has been developed and implemented, it's useful to know how well the algorithm performs relative to other motion estimation algorithms. There are several criteria for evaluating a generated motion vector field. These include average angular error, average endpoint error, average interpolation error, and average normalized interpolation error.[10]

The average angular error is the average error in angle of the estimated motion vectors. The average endpoint error is the average error in the endpoints of the motion vectors. That is, the endpoint error is the magnitude of the difference between the estimated motion vector and the ground truth vector at a point. Of course, in order to compute the angular error and endpoint error, one must have access to ground truth motion data for the image sequence she is using.

Computing the interpolation error and normalized interpolation error does not require ground truth data. These error metrics are computed by comparing the first frame in the sequence to the second frame projected to look like the first frame using the estimated

motion. The interpolation error is root-mean-square of the difference between the true first frame and the projected second frame. Simply subtracting the first frame and the projected second frame computes an error image called the residual. The normalized interpolation error is a weighted RMS average of the pixels in the residual, using the image gradient as the weight.

The Middlebury optical flow website (`http://vision.middlebury.edu/flow/`) provides a number of image sequences with ground truth motion data. It also provides an evaluation service. With this service, a researcher may submit her estimated motion vector fields for a set of image sequences. Her motion vector fields are then evaluated and compared to other researchers' methods.

4 Previous Work

The motion estimation algorithm I present later uses edge detection, edge matching, and interpolation algorithms. In this section, I describe previous research into these areas.

4.1 Edge Detection

Edge detection is an important part of my algorithm. Several algorithms have been developed for edge detection. Here I present the Canny edge detector, which is a very popular edge detection algorithm and is in some ways very similar to the algorithm I use (which I describe later).

4.1.1 The Canny Edge Detector

The Canny Edge Detector is an edge detection algorithm developed by John F. Canny in 1986.[5]

The Canny algorithm begins by reducing noise in the image. Noise reduction is done by convolving the image with a Gaussian filter. This reduces the effect that individual noisy pixels will have on the final result. Next, the algorithm finds the intensity gradient of the image by applying two filters to the image. The filters estimate the first partial derivative of the image intensity in the horizontal and vertical directions (call the image I and the partial derivatives G_x and G_y, respectively). The magnitude of the gradient is then given as

$$|\nabla I(x,y)| \approx G(x,y) = \sqrt{{G_x}^2 + {G_y}^2} \tag{1}$$

and the direction of the gradient is given by

$$\theta(x, y) = atan2(G_y, G_x) \tag{2}$$

Each pixel's edge direction is then rounded and is labeled as belonging to one of four possible directions: horizontal, diagonal north-east/south-west, vertical, or diagonal north-west/south-east.[5]

The Canny algorithm then performs a step called Non-Maximum Suppression, which generates a binary image of the edge points in the original image. A pixel is given a value of 1 if and only if its gradient is greater than the gradients of the pixel's two immediate neighbors in the pixel's rounded gradient direction. That is, if the pixel's rounded gradient direction is horizontal, then the pixel is given a value of 1 iff the pixel of interest has a greater gradient value than both its left and right neighbors. The procedure is similar for pixels with vertical or diagonal gradients.

Finally, edges are extracted by tracing the binary edge image. The Canny edge detector uses thresholding with hysteresis to identify important pixels in the binary edge image, then traces those pixels. Not all pixels with value 1 in the binary edge image will eventually be traced as an edge.[5]

4.2 Occlusion Boundary Detection and Edge Matching

Researchers have experimented with edge matching approaches to motion estimation in the past. Edges are important because they often correspond to depth discontinuities in the images. Edges thus often mark the positions of discontinuities and undefined regions in the motion vector field. Here, I describe two research papers that deal with the problem of locating depth discontinuities in image sequences.

The first, *Local Detection of Occlusion Boundaries in Video by Andrew N. Stein and Martial Hubert*, focuses on rating edges by the degree to which those edges exhibit occlusion. The authors' algorithm locates edges using a spatio-temporal edge detector. Parts of the images on both sides of each edge are then compared to determine how strongly that edge looks like an occlusion boundary. The comparison is done by measuring differences in the histogram of the pixel values on either side of the edge.[3]

The second, *Layered Motion Segmentation and Depth Ordering by Tracking Edges by Paul Smith, Tom Drummond, and Roberto Cipolla*, focuses on identifying layers of motion in a sequence of images. The algorithm uses the Canny edge detector to find the edges. It then uses the Expectation-Minimization algorithm to match the edges. The relative depth of segments of the images are then estimated. The method presented in the paper is not limited to just two frames of a sequence.[4]

4.3 Interpolation in 2D

Interpolation in 2D from scattered data points (that is, data points that are not ordered into a regular grid) is useful in a wide variety of areas. As a result, methods have been developed to solve this problem. One approach is to find the Delaunay triangulation of the known data points. The function is then linearly interpolated over each of the triangles.

Another approach, and the approach I use in my algorithm, is essentially to solve Laplace's equation over the domain for which the interpolated function is desired. The boundary values are taken to be the known data points.

5 My Algorithm

To estimate the motion of every pixel in the first frame of the sequence, I break the motion estimation problem into several parts. The first part is to identify a set of edges in each frame of the sequence. Once the edges have been located, my algorithm matches edges between frames. During the edge matching process, only candidate matches with a high likelihood of correctness are retained. Next, motion vectors are estimated for the matching edge pixels. And finally, the motion vectors are interpolated between the matching edge pixels.

5.1 Finding the Edges

To find edges in the original image, my algorithm may (this is optional) first reduce noise in the image. To reduce the image noise, I convolve the image with a Gaussian filter.

Next, to locate the edges in the given image, my algorithm first identifies two special neighbor pixels for each pixel in the image. I call these two special pixels the left-of-gradient neighbor (or just left neighbor) and the right-of-gradient neighbor (or just right neighbor). For a given pixel (call this the pixel of interest), I define the left-of-gradient neighbor as the pixel with maximum gradient magnitude that is one of the eight pixels neighboring the pixel of interest and is left of the pixel of interest's gradient vector. By left of the gradient vector, I mean that the neighboring pixel's center should be between $0°$ and $180°$ counterclockwise of the gradient vector. Any pixel that is not left of the gradient vector is by definition right of the gradient vector. Similarly, I define the right-of-gradient neighbor as the pixel with maximum gradient magnitude that is one of the eight pixels neighboring the pixel of interest and is right of the pixel of interest's gradient vector. Figure 2 demonstrates left/right neighbors.

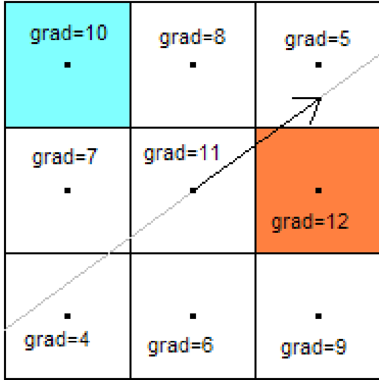


Figure 2: The arrow represents the gradient vector for the center pixel, and the "grad=#" values indicate the gradient magnitude at each pixel. The top three pixels and the middle-left pixel are all left of the center pixel's gradient. The center pixel's left neighbor is colored blue and the center pixel's right neighbor is colored orange.

Each pixel is then given a numeric "score"/classification which corresponds to the likelihood that the pixel belongs to an edge. To compute a given pixel's score, the pixel's score is first set to zero. If the pixel's left neighbor's right neighbor happens to be the given pixel, its score is increased by one. If the pixel's right neighbor's left neighbor happens to be the given pixel, the given pixel's score is increased by one again.

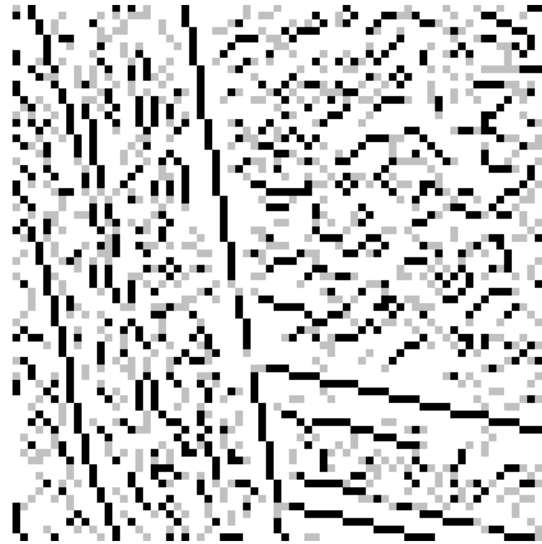


Figure 3: A closeup of a map of the score values for an image. White pixels have score 0, gray pixels have score 1, and black pixels have score 2.

Once each pixel has been classified we may then extract two types of edges from the image. Type 1 edges can be found by tracing consecutive pixels with a score of at least 1. Type 2 edges can be found by tracing consecutive pixels with a score exactly 2.

After the edges have been traced, I recommend that only edges with length greater than some threshold be kept. In my implementation, I discard all edges that contain fewer than 16 pixels.

I also recommend that the edges be stored in a consistent way. In my implementation of this algorithm, I store all edges starting at the pixel that is rightmost with respect to the edge's gradient. That is, when storing a nearly straight edge that has a gradient to the south-east direction, I store the south-western-most pixel of the edge first (imagine standing on the edge facing in the direction of the gradient, the pixels to your right are stored before pixels to your left). If the edge is a loop, I cut the edge at it's leftmost point on the image, then store the edge as if it did not contain a loop.

5.1.1 Properties of Type 2 Edges

Due to the way that type 2 edges are extracted, they have a number of properties. The most important of which is that it is impossible for a type 2 edge to contain a fork (that is, a type 2 edge cannot split into two edges that go in different directions). This property makes the

comparison of edges easier, since without forks comparing two edges can at its most basic level be viewed as a problem of matching sequences of features values. If the edges were to contain forks, matching the edges would be a more complex problem of matching graphs.

Another property of type 2 edges that simplifies matching is that type 2 edges are "double-checked", meaning that only the highest quality pixels are extracted. Although this improves the probability that edges will be correctly matched, it unfortunately also reduces the total number of matches that are possible.

5.1.2 Differences with the Canny Edge Detector

My edge detection algorithm is in some ways similar to the Canny Edge Detector. Both edge detectors use the same noise reduction/blurring step, both algorithms use only the gradient of the image once it is computed, and both algorithms produce a classification image, which is traced to produce the final set of edges. The difference between the two algorithms is the way in which edges are extracted from the gradient image.

It could be said that both algorithms round the gradient direction to a "precision" of 45°. The Canny edge detector rounds to the nearest angle that is a multiple 45° off the x-axis, and throws away the sign of the direction. That is, to the Canny edge detector east is equivalent to west and north-east is equivalent to south-west. The Non-Maximum Suppression step can then be seen as a possible shift in the position of the edge pixels by at most one pixel to maximize the gradient magnitude. Instead of shifting the position of the edge pixels to maximize the gradient, my algorithm rounds the direction of the gradient to maximize the gradient along the traced edge.

The other major difference is in how each algorithm decides which pixels of the classification image to extract as edges. As mentioned before, the Canny edge detector uses thresholding on the gradient magnitude and hysteresis. My method, instead, only extracts pixels from the classification image that have a consistent gradient direction along the edge. Since I'm extracting edges that may later be matched with edges from other images, it is not so important that the edges extracted at this stage be only edges with a large gradient magnitude.

5.1.3 Example Output of Edge Detector

Figure 4 shows an example of the results of my edge detection algorithm. All edges with fewer than 16 pixels have been discarded.

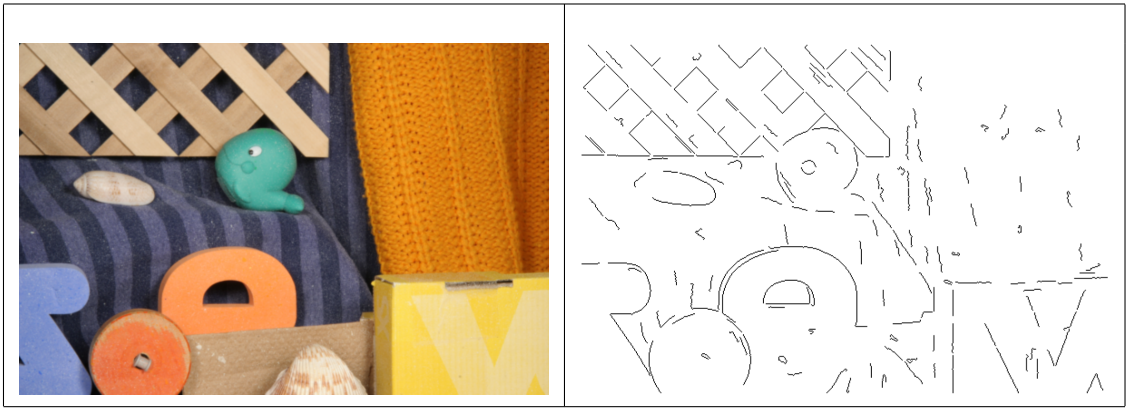


Figure 4: The left image is a full frame from the RubberWhale dataset[9] and the right image shows all edges containing at least 16 pixels extracted using my algorithm.

5.2 Matching the Edges

The next step in my motion estimation algorithm is to match the edges extracted from different frames. For each edge in an input image, the edge matching step tries to find the corresponding edge in the other images. If no such correspondence is found, the edge is discarded. For simplicity, I will restrict the number of input images to two.

The edges produced by my edge detection algorithm have two main properties which make edge matching more difficult. The first is that the edges are typically just fragments of the full edge. And the second is that the endpoints of the edges are not very stable. Figure 5 shows a few of the longest edges extracted from two consecutive frames as an example.

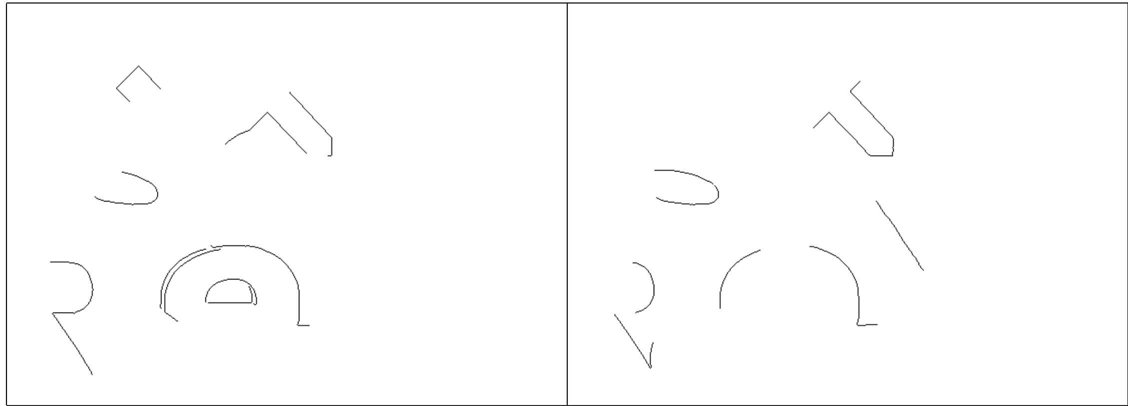


Figure 5: Both images show a few of the longest edges extracted from two consecutive frames of the RubberWhale sequence.[9]

As can be seen in Figure 5, the end points of edges can vary quite a bit between frames. Figure 5 also demonstrates that what a person would call an edge is typically represented by multiple edge fragments by my program.

5.3 Computing the Motion Vectors Along the Edges

Once the edges have been matched, the task of computing the motion vectors along the overlapping portion of matching edges requires that individual pixels in both images be matched. Internally the representations of the matching edges are aligned. To align the edges, my algorithm uses a function for comparing edges assuming different alignments. The function then chooses the possible alignment that minimizes the error. The error between edges assuming a given alignment is computed by matching individual pixels in the edges. Once these individual pixels are matched, the motion vector is simply the difference in position of the matching pixels in consecutive frames.

Figure 6 shows the direction of the gradient vector as a function of position along the edge for the same edge in two different images. The edges are aligned internally by shifting the starting position of the edge in the second frame.

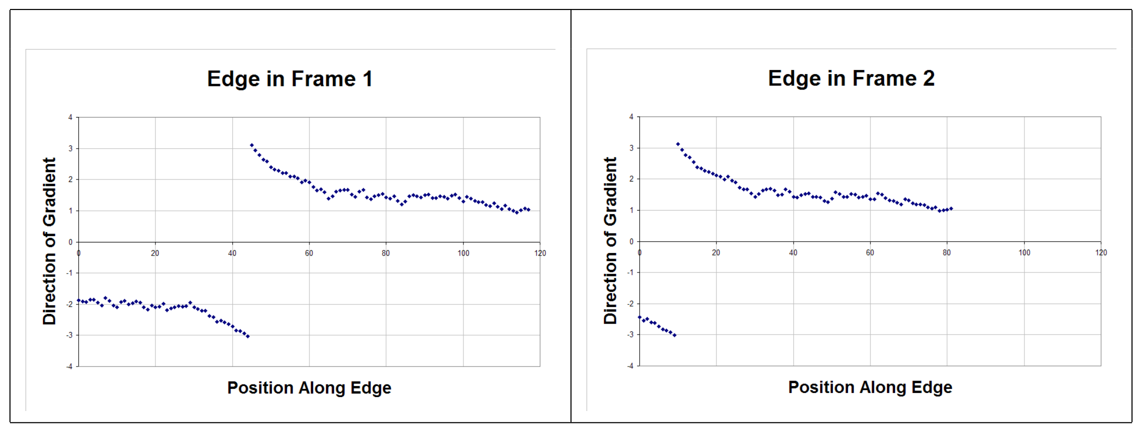


Figure 6: An edge from two different frames.

Figure 7 shows the edge, after it has been aligned.

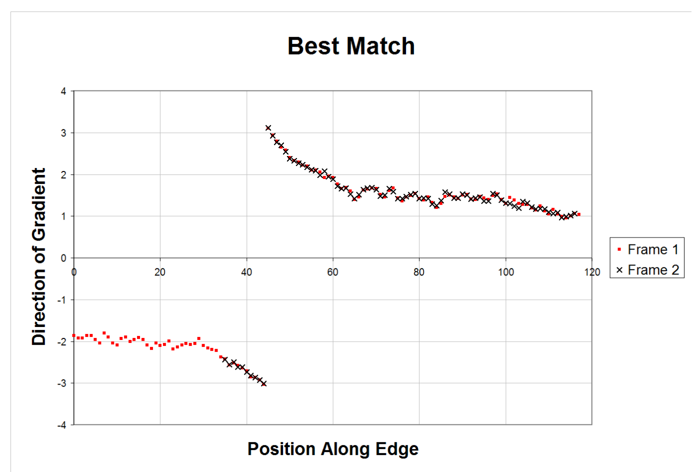


Figure 7: Properly aligned edge representations.

5.4 Interpolating the Motion Vectors

Once motion vectors have been generated for the edge pixels, the final step to generate the full vector field is to interpolate those "known" motion vectors. The x and y components of the motion are each interpolated separately. Each component of the final motion vector field is given as an approximate solution to Laplace's equation, with the known motion vectors

used as the boundary conditions. Laplace's equation is a partial differential equation and is given by

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0 \tag{3}$$

To solve Laplace's equation, I use an iterative fixed point method. Replacing the second partial derivatives by finite difference approximations results in equation (4):

$$(f(x+1,y) - 2f(x,y) + f(x-1,y)) + (f(x,y+1) - 2f(x,y) + f(x,y-2)) = 0 \tag{4}$$

Thus, any pixel that is not given as a boundary value in the interpolated motion vector field should be approximately equal to the average of its four closest neighbors. My interpolation algorithm converges to this approximation.

Initially, I tried repeatedly replacing all unknown motion vectors with the average of their four closest neighbors. Although this method should eventually converge, I've found that it tends to converge very slowly in regions that are not near any boundary value pixels.

To improve convergence, I attach a weight value to each pixel. Initially, all boundary value pixels are given a weight of 1, and pixels with unknown motion are given a weight of 0. Motion vectors are then updated using the following formula:

$$\begin{aligned} f(x,y) \leftarrow & \frac{w(x,y) * f(x,y) + w(x-1,y) * f(x-1,y) + w(x+1,y) * f(x+1,y)}{w(x,y) + w(x-1,y) + w(x+1,y) + w(x,y-1) + w(x,y+1)} \\ & + \frac{w(x,y-1) * f(x,y-1) + w(x,y+1) * f(x,y+1)}{w(x,y) + w(x-1,y) + w(x+1,y) + w(x,y-1) + w(x,y+1)} \end{aligned}$$

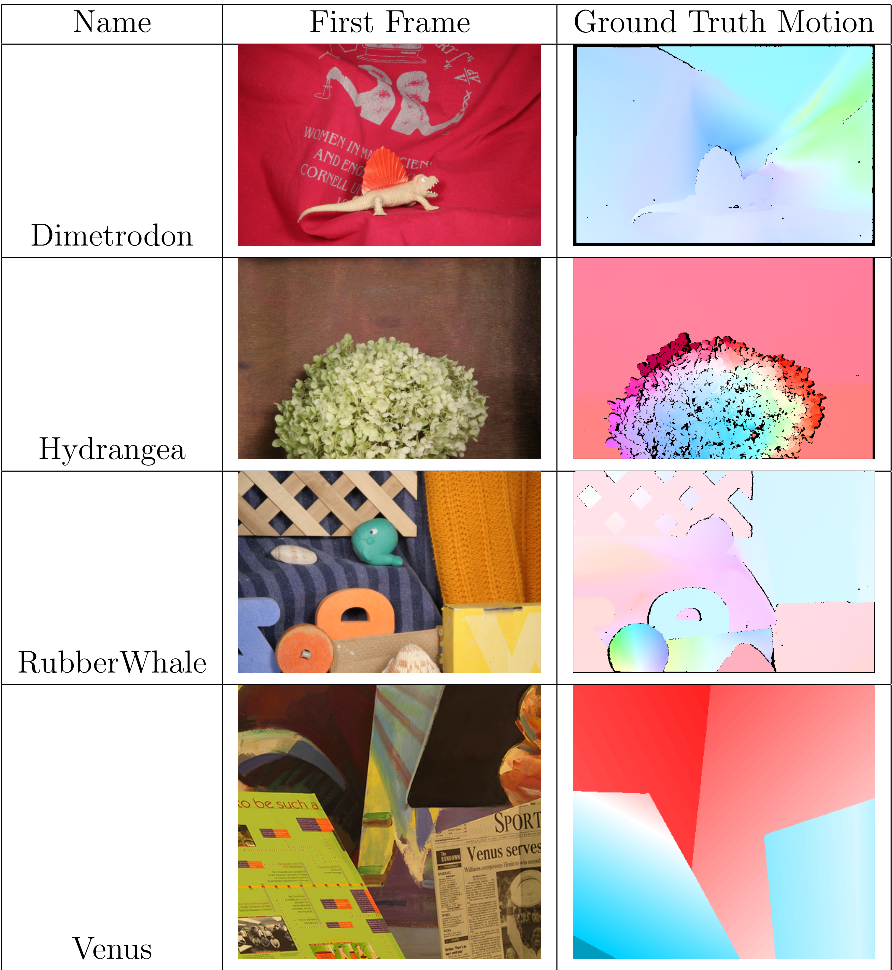
where $w(x,y)$ is the current weight at pixel (x,y) and $f(x,y)$ is the current estimate of the motion at pixel (x,y). That is, each pixel is set to the weighted sum of itself and its four closest neighbors. Motion vectors are only updated when $w(x,y) + w(x-1,y) + w(x+1,y) + w(x,y-1) + w(x,y+1) > \epsilon$ for some small $\epsilon > 0$, to avoid divide by zero problems. In my implementation, I choose $\epsilon = 10^{-10}$. The weight values are then updated using the following formula:

$$w(x,y) \leftarrow \frac{w(x,y) + w(x-1,y) + w(x+1,y) + w(x,y-1) + w(x,y+1)}{5} \tag{5}$$

I update the motion vectors in each of the four possible row orders (increasing column, increasing row; increasing column, decreasing row; ...) multiple times. In my implementation, I do this update a fixed number of times, typically between 64 and 1024.

6 Results and Analysis

To test my algorithm, I used the publicly available image sequences and ground truth motion data provided on the Middlebury optical flow website.[9] For your reference, here are the four natural (non-synthetic) image pairs with public ground truth motion data:



Name	First Frame	Ground Truth Motion
Dimetrodon		
Hydrangea		
RubberWhale		
Venus		

Figure 8: The first frame of each of the four natural image sequences with publicly available ground truth motion on the Middlebury site.[9]

The coloring for all motion vector images was generated using the *color_flow* program (source code provided on the Middlebury website).[9] I used 10 for the *maxmotion* parameter

in all images. Pixels that are colored black have no motion associated with them.

Before interpolation, my algorithm produced the following results for each of the four image sequences:

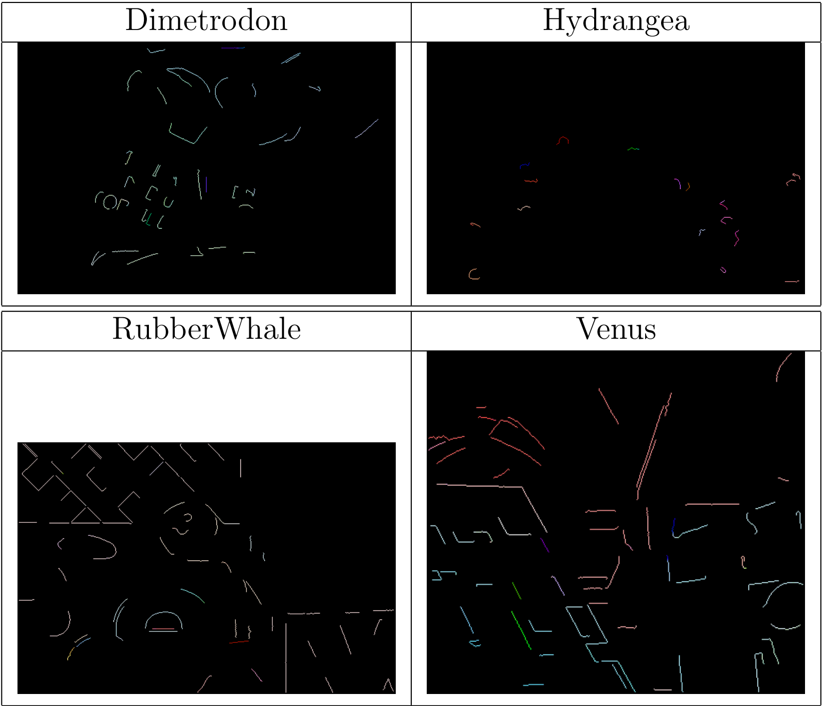


Figure 9: Motion estimated by my algorithm before interpolation. (With minimum allowed edge length of 20 pixels.)

For the most part, the edge motion vectors do visually appear to match the ground truth data fairly well. After interpolation, these motion vector fields become the following:

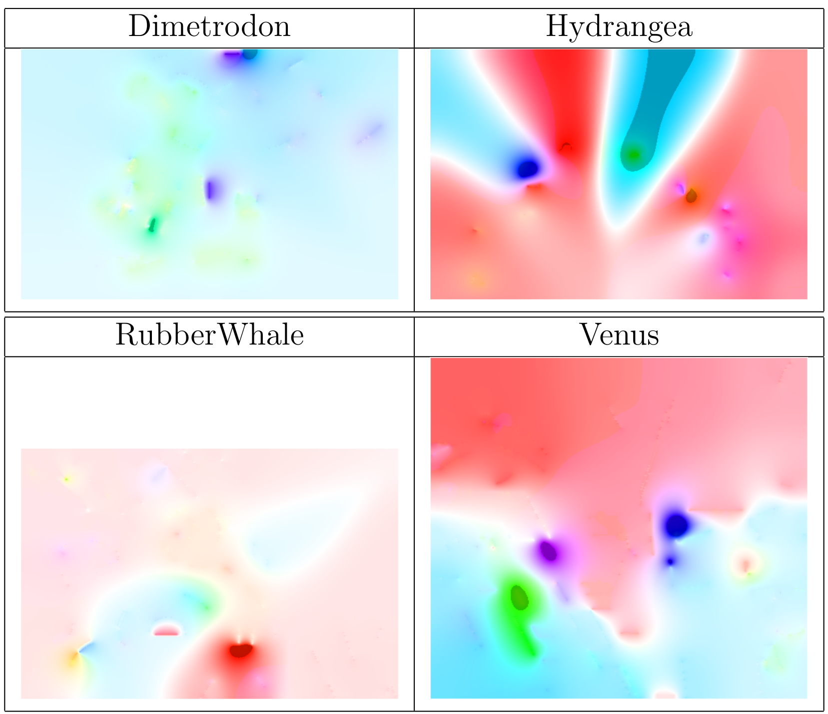


Figure 10: Motion estimated by my algorithm after interpolation. (With minimum allowed edge length of 20 pixels.)

Visually, the Venus motion estimates appear to be the most correct and the Hydrangea estimates appear to be the worst. The Venus estimates likely appear better than the Hydrangea estimates because there are many more matching edges for the Venus dataset and the motion for the Venus dataset is much simpler than that for the Hydrangea dataset.

Ideally, the motion estimates should clearly show boundaries in the motion. Since, in its current form, the interpolation algorithm does not behave differently for occlusion boundaries and non-occlusion boundaries, motion boundaries do not show up in the final motion estimates. Figure 11 shows the average endpoint error of my algorithm for each of the above natural sequences as well as for four other synthetic image sequences. It also shows the average endpoint error for these sequences along only the edge pixels. For comparison, the average magnitude of the ground truth vectors is also shown.

Sequence	Error (edges)	Error (all)	GT Magnitude (edges)	GT Magnitude (all)
Dimetrodon	2.015	1.160	2.208	2.058
Hydrangea	5.110	4.240	5.368	3.731
RubberWhale	0.975	1.179	1.255	1.257
Venus	1.031	1.039	3.376	3.795
Grove2	3.863	2.332	3.290	3.089
Grove3	5.027	3.689	4.390	3.903
Urban2	10.975	8.403	9.257	8.394
Urban3	12.681	11.521	7.095	7.310

Figure 11: The average error in the estimated motion vector fields and the average magnitude of the ground truth vectors. (All numbers are in units of pixels.)

To compute the average errors given in Figure 11, I computed the average *local minimum error*. To compute the local minimum error, I compute the minimum magnitude of the difference between the estimated motion vector and each of the nine ground truth motion vectors in its neighborhood. I use the local minimum error to remove any possibility that at an occlusion boundary I'm comparing the background ground truth motion with the foreground motion generated by my algorithm. Equation (6) shows the formula I used to compute the error values in the above table.

$$E(S) = |S|^{-1} \cdot \sum_{z \in S} min\{||v_{gt}(w) - v_{est}(z)||_2 \colon ||w - z||_2 \leq \sqrt{2}\} \tag{6}$$

Where S is the set of all pixel coordinates for which the error is to be averaged, $E(S)$ is the average error on that set, and $v_{gt}(w)$ and $v_{est}(z)$ are the ground truth and estimated motion vectors at w and z, respectively. $|S|$ is the cardinality of the set S, and $||v||_2$ is the two-norm of a vector v. All vectors have units of pixels. To compute the average ground truth magnitude values in table 11, I use equation (7).

$$M(S) = |S|^{-1} \cdot \sum_{z \in S} ||v_{gt}(z)||_2 \tag{7}$$

Where $M(S)$ is the average ground truth magnitude shown in the table.

As can be seen from the error table, my original hypothesis that most of the error comes from the interpolation is false. In fact, in most cases the interpolated motion vectors are better on average than the edge vectors. I have manually tested to be sure that my implementation does indeed accurately represent my algorithm. For these tests, I manually

created several simple image sequences and verified that the motion vectors are what I expected them to be for those sequences.

7 Conclusion

My motion estimation algorithm successfully processes image data with varying results. It accurately identifies important edges in the source images and does a better than expected job of interpolating motion vectors. But the algorithm can also be improved. By analyzing the results from a variety of public datasets, I now believe the edge processing stages have defects.

In the paper, I discuss three main suggestions for improving the edge matching and detection. First, there should be a high density of ”known” motion vectors. Second, the interpolation algorithm should behave differently for occlusion boundaries and non-occlusion boundaries. And third, more than just edges should be used to generate ”known” motion vectors. I am also interested in the advantages that might be gained by segmenting the images and applying block matching to each segment, but have not yet explored this avenue.

References

[1] Robyn Owens. *Stereo matching*. http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT11/node5.html#SECTION00052000000000000000, 10/29/1997. 12/06/2009.

[2] Jean-Michel JOLION. *Difficulties in motion estimation*. http://rfv.insa-lyon.fr/~jolion/IP2000/report/node11.html, 06/20/2000. 12/06/2009.

[3] Andrew N. Stein and Martial Hebert. *Local Detection of Occlusion Boundaries in Video*.

[4] Paul Smith, Tom Drummond, and Roberto Cipolla. *Layered Motion Segmentation and Depth Ordering by Tracking Edges*.

[5] Wikipedia. *Canny edge detector*. http://en.wikipedia.org/wiki/Canny_edge_detector, 03/07/2010.

[6] Wikipedia. *Motion estimation*. http://en.wikipedia.org/wiki/Motion_estimation, 06/12/2009.

[7] Wikipedia. *Motion compensation*. http://en.wikipedia.org/wiki/Motion_compensation, 01/06/2010.

[8] The Multimedia Coding Group, the University of Warwick. *Block-Matching Motion Compensation*. http://www.dcs.warwick.ac.uk/research/mcg/bmmc/index.html.

[9] Daniel Scharstein. *Optical Flow Datasets*. http://vision.middlebury.edu/flow/data/, 08/16/2009.

[10] Simon Baker, Daniel Scharstein, J.P. Lewis, Stefan Roth, Michael J. Black, and Richard Szeliski. *A Database and Evaluation Methodology for Optical Flow*. http://vision.middlebury.edu/flow/MSR-TR-2009-179.pdf, 12/2009.

[11] Wikipedia. *MPEG-1*. http://en.wikipedia.org/wiki/MPEG-1_Part_2#Part_2:_Video, 04/11/2010.

[12] Wikipedia. *MPEG-4 Part 2*. http://en.wikipedia.org/wiki/MPEG-4_Part_2, 04/14/2010.

[13] Xavier Armangue, Joaquim Salvi. *Overal view regarding fundamental matrix estimation*. http://www.sciencedirect.com/science?_ob=MImg&_imagekey=B6V09-47RBCDP-1-4D&_cdi=5641&_orig=search&_coverDate=02%2F10%2F2003&_

sk=999789997&view=c&wchp=dGLbVtb-zSkWA&_acct=C000053445&_version=1&_userid=1517015&ie=f.pdf&_valck=1&md5=8a19ac12546d62cd1344ed412a20aa6a&ie=/sdarticle.pdf, 09/26/2010.

[14] Wikipedia. *Lambertian reflectance.* http://en.wikipedia.org/wiki/Lambertian_reflectance, 04/01/2010.