

# Near Real-Time Optical flow

Patrick Zhao  
Minas E. Spetsakis

Dept. of Computer Science  
York University  
4700 Keele Street  
North York, ONTARIO  
CANADA, M3J 1P3

## Abstract

*Optical Flow estimation is a central problem in Computer Vision and many applications need to do it in real time. We present an algorithm that can compute flow in almost real time on a regular workstation without any special purpose hardware accelerators. The algorithm can compute flow with large disparities. In terms of noise robustness and accuracy it compares favorably with some of the best but much slower algorithms.*

## 1. Introduction

The continuous improvement of the performance of computer hardware has reached the point where image processing can be done in close to real time without dedicated hardware. Optical flow would take several minutes just a few years ago, but today it takes much less. So research in real time image processing and computer vision can concentrate on improving the performance of algorithms that execute on regular workstations rather than algorithms mapped on expensive dedicated hardware. While dedicated hardware [9] is still faster and in some cases the only way to go, the performance gap is small and constantly shrinking.

There are several improvements in technology that make the hardware so fast today. And the reason that underlies all these is that there is tremendous demand for image and video compression and decompression, graphics and image based rendering. All these reasons led the computer manufacturers develop technologies that improve the image and video performance of their products.

Some of these technologies, like higher clock rate and bus bandwidth improve all applications. Others like special video instructions, are targeted mainly on the video applications.

To a certain extend, real time image motion estimation is part of many packages like Sun MediaLib, but these are designed for video compression where the requirements are different: The overlap of the matching regions is limited, the sampling is not dense, aliasing is not important, etc. In this paper we explore real time implementations of general purpose optical flow.

The main requirement of a real time optical flow algorithm is to be as fast as possible without sacrificing too much precision. Taking advantage of cache memory and special machine instructions is one component of the research. The other is develop fast algorithms or alternatives to existing methods and the third is to decide which popular techniques are too expensive for their benefit and which are worth implementing on a real time system. So we used Sun's XIL library to take advantage of the SPARC VIS instruction set, and rearranged the code so that we make maximal use of the cache. We combined quadratic fitting with the computation of sum of squared differences to reduce the number of convolutions and we replaced the smoothness or regularized operators with much simpler techniques that do not need iterative algorithms. The end result is an algorithm that can work with up to 8 pixels of interframe motion (16 if we use  $640 \times 480$  images), work at about 2 frame pairs a second and produce results

that compare favorably with the best unaccelerated algorithms.

Optical flow techniques can be classified as “differential”, “matching”, “energy based” or “phase based” [2] After exploring the benefits and the shortcomings of each we settled for one that is best described as “matching” technique. The differential techniques, the most well known being Horn and Schunck [7], can suffer from alias in the presence of fine texture. Although this can be mostly solved by incorporating a multi-resolution hierarchical scheme like Anandan’s [1], Bergen’s [5], or Barron and Khurana’s [3], Horn and Schunck are ruled out mainly because of the need to solve the differential equation that comes from the smoothness constraint. The end result is that, no matter what form the solution of the smoothness constraint take, the method is computationally expensive. Another algorithm by Lucas and Kanade [11], is fast and accurate and a variant of it would make an excellent starting point for a real time implementation [6]. It was a close second in our preferences but was rejected mainly because of number of operations needed.

The phase based and energy based techniques were ruled out because of computational efficiency. They require convolution with a bank of expensive filters and there is little hope that they could be implemented efficiently. This leaves us with the matching algorithms.

This class of methods seems most promising. As the name implies they try to establish optical flow by taking one small region from one image and matching it with another region in the next image. They typically involve some form of search, either straight or approximated by a function [2]. The closest relative to our algorithm would be Anandan’s [1]. We do not use Laplacian pyramids though, we have a faster quadratic fitting and use straight search at the lowest level. We found that using the straight image was not only more computationally efficient but also more accurate. We also explored the possibility of using correlation instead of summed square differences, but then the technique that speeds up the quadratic fitting would not be applicable.

The organization of the paper is as follows. Next section gives an overview of the algorithm and the implementation. Section 3. discusses the problems associated with large interframe motion. Section 4. presents the accelerated quadratic fitting. Section 5. discusses the straight search.

Section 6. discusses some problems related to the running sum convolution. Section 7. discusses smoothness and the hierarchical integration. Section 8 presents the experimental results using both real and synthetic images. Finally Section 9. presents the conclusions and future work.

## 2. Overview of the Algorithm

After exploring several alternatives we found that the best scheme should employ a multi-resolution hierarchical approach with quadratic fitting at all levels except the last where we use a straight search with a bigger search area. We use the standard assumptions of constant intensity and approximately uniform motion. We assume that there few discontinuities, so they do not affect the result too much. We do not model them explicitly [13, 8].

We implemented our algorithm in MediaMath where it is easier to develop algorithms and then used this as a reference implementation in terms of correctness.

MediaMath is an interactive system for image and audio analysis, It is a very convenient platform for image processing, software developing, algorithm testing, and debugging. It has its own interpreted language, that has syntax similar to C, provides most of the common image processing functions, and can be extended in C. All these features make MediaMath a very good platform to develop prototypes and test new algorithms.

After implementing the basic algorithm using MediaMath script we replaced most of the components with ones written in C; either raw C or a blend of SUN XIL functions and C functions to make it run in real time.

XIL is a image processing and image compression function library provided by SUN. It provides common image processing functions and it is a connector between high level of image processing applications and system hardware that includes accelerators and frame buffers. The programmer does not need to know the details of the hardware and but can still get the advantage of available hardware acceleration, frame buffers, etc. Most modern processors have built in image processing accelerators in the form of SIMD machine instructions. XIL takes advantage of SPARC’s VIS instruction set.

## 2.1. Why combine XIL and C functions

Although XIL provides commercial quality image processing infrastructure and we used it as much as we could, many things had to be written in C. XIL for example does not have a running sum convolution and if we used ordinary convolution to implement running sums the performance was unacceptable. So we implemented running sum convolution in C and we can now compute several frames per second. At the time of this writing we need around 0.5 second per frame for an image pair of size 320 X 240. This can be considered as close to real time. We expect that with a few improvement in software ( like more extensive use of short integers) and new generation hardware we will get down to 0.1 second per frame.

## 3. Limits of Interframe Motion

Because the time between the images in the sequence is very short, the movement of a region can not be too large, but it is still large enough to cause trouble to most algorithms. The classic solution to this problem is to use either a hierarchical approach [1] or search a bigger area for matches.

To illustrate the advantages and disadvantages of both approaches let's assume that the maximum movement is 16 pixels in each direction.

Using the hierarchical approaches we reduce the resolution. This reduces the amount of motion and the computation time, but also reduces the accuracy. So we use this inaccurate estimation of flow as a guess for the next higher resolution. And do this repeatedly each time reducing resolution to half. After a few reductions in resolution the size of the image and the amount of flow are small enough to allow us to solve the problem. But every time we reduce resolution we throw away information and the the guess might not only be "inaccurate" but totally wrong. A distinct feature 3-4 pixels in size, for example, might completely disappear in the process.

On the other hand we can use bigger search window. For every region we try to match it with  $N^2$  candidates centered in a  $N \times N$  search region. But if we have 16 pixels motion in each direction then  $N = 33$  and the search window is  $N^2 = 1089$  pixels. This is too big.

So we use a combination of the two approaches. We reduce the resolution three times so the resolution of a standard image goes from  $640 \times 480$  to  $80 \times 60$  and the image motion goes to 2

pixels and at the lowest resolution we search a bigger window for up to 2 pixels motion in each direction. We use the result as a guess for the next higher resolution and, assuming our guess is good, the residual image motion is one pixel or less. We then have a search area that is only  $3 \times 3$  and we compute flow using a quadratic fitting algorithm that gives both sub-pixel estimates and needs much less computation than a straight search. Next we present the two methods: the quadratic fitting method and the discrete searching method.

## 4. Quadratic Fitting Method

If we assume 1 pixel motion in each directions, there are 9 possible discrete possibilities (left, left up, left-down, etc) including the original location, so we need to search 9 locations to get the movement of every region. We do this by taking the Sum of Squared Difference (SSD) between the region and its 9 candidate matches and finding the minimum. Motion of course is hardly ever discrete so the actual minimum can be between pixels. So instead of finding the discrete minimum we fit a quadratic and minimize it. Straight-forward application of quadratic fitting would involve computing 9 SSDs, fitting the polynomial and computing the minimum, At first sight it seems like a lot of computation. But we can show that it is even faster than discrete matching.

The input to the program is a set of two images  $I_1$  and  $I_2$ . We assume that the amount of image motion does not exceed 1 pixel in each dimension.

Let  $I_2^{(shift)}$  be  $I_2$  shifted by an amount *shift*. Then

$$(I_1 - I_2^{(shift)})^2$$

is an image every pixel of which is the squared difference of intensity of  $I_1$  and the shifted  $I_2$ . Under the constant intensity assumption this should be zero for the pixels that have motion equal to *shift*.

Let  $g$  be a convolution template (also called convolution kernel) that is

$$g[i, j] = \begin{cases} 1 & \text{if } -k \leq i, j \leq k \\ 0 & \text{otherwise} \end{cases}$$

If we convolve the squared differences with the uniform template  $g$

$$\mathbf{S}(shift) = (I_1 - I_2^{(shift)})^2 \otimes g \quad (4.1)$$

then we get an image every pixel of which is the sum of the squared differences of the intensities in the  $(2k+1) \times (2k+1)$  region centered at this pixel. If the motion of a region is actually *shift* then the corresponding central pixel is zero. In the presence of noise or when the assumptions of uniformity of motion do not hold, then it will not be zero.

A typical value for  $k$  is 5, which makes the regions that we match  $11 \times 11$  so every convolution needs 121 multiplications and about as many additions. And we have to do this 9 times, once for every possible shift. We have a huge number of multiplications and additions in total; about 2000.

Although the results would be better if we used a template other than uniform, we tend to prefer it because it can be implemented very efficiently. It is separable, which means we can convolve in the vertical and horizontal directions separately and second it can be implemented as a running sum that needs one addition and one subtraction per directional convolution. Unfortunately XIL does not have a running sum convolution so we did it using raw C. Despite the disadvantage of not using accelerated functions the performance of raw C was much better.

We compute the *shift* that minimizes  $(I_1 - I_2^{(shift)})^2$  for every pixel by fitting a quadratic and minimizing it. Next we show how it can be accelerated.

#### 4.1. Accelerated Quadratic Fitting

We perform least squares fit of a quadratic to the SSDs of Eq. (4.1) for every *shift* =  $\{u, v\}$

$$P(u, v; a_{00}, a_{01}, a_{02}, a_{10}, a_{11}, a_{20}) = a_{00} + a_{01} v + a_{02} v^2 + a_{10} u + a_{11} u v + a_{20} u^2$$

and we compute the unknown coefficients  $a_{00}, a_{01}$ , etc that minimize the expression

$$\sum_{u,v} (S(u, v) - P(u, v; a_{00}, a_{01}, a_{02}, a_{10}, a_{11}, a_{20}))^2.$$

The interesting thing is that the minimization can be done analytically, simplified symbolically and then translated to C. The resulting code can be optimized and in the end it is more efficient than even simple search. Using the symbolic manipulation program *Maple* we get

$$a_{00} = \frac{2 S(0, 1)}{9} + \frac{2 S(0, -1)}{9} - \frac{S(-1, 1)}{9} - \frac{S(-1, -1)}{9} + \frac{2 S(1, 0)}{9} - \frac{S(1, 1)}{9} - \frac{S(1, -1)}{9} + \frac{2 S(-1, 0)}{9} + \frac{5 S(0, 0)}{9} \quad (4.2)$$

and similar expressions for the other coefficients of the quadratic. The minimum of the polynomial is achieved for

$$u = -\frac{-a_{01} a_{11} + 2 a_{02} a_{10}}{-a_{11}^2 + 4 a_{02} a_{20}} \quad (4.3)$$

$$v = \frac{a_{10} a_{11} - 2 a_{01} a_{20}}{-a_{11}^2 + 4 a_{02} a_{20}}$$

We can notice two things here. First, that the coefficients  $a_{00}, a_{01}$  etc are given by simple sums of various  $S(u, v)$  and since  $S(u, v)$  is given by a convolution, we can change the order of these two linear operations and do the convolutions after the additions. There are only six  $a_{ij}$ s, whereas there are 9  $S(u, v)$ s. Second,  $a_{00}$  does not appear on Eq. (4.3), so the number of convolutions go down to 5. This technique is similar to the ones presented in [10].

This saves much more than a few floating point operations because most of the cost of the running sum convolutions is bringing the image from the memory and back. The other computations involved are local so we do not have the cost of bringing things from the memory all the time.

This works fine for synthetic images with rich texture but it can be problematic with real images that can have extended regions without any intensity variation either due to constant intensity of the image or more often due to saturation of the camera. In these areas the denominator of Eq. (4.3) will be either zero or close to zero and the result meaningless. Furthermore this result will be propagated to higher resolutions and contaminate them. Luckily the solution is not very costly. The first thing to do is to use a “smallness” term, a term that will make the quadratic minimization favor smaller solutions in case there is no information. This simply means to add a constant parabola weighted by the coefficient  $\delta$  to the quadratic. Then Eq. (4.3) becomes:

$$v = -\frac{-a_{10} a_{11} + 2 a_{01} a_{20} + 2 \delta a_{01}}{-a_{11}^2 + 4 \delta^2 + 4 \delta a_{02} + 4 a_{20} \delta + 4 a_{02} a_{20}} \quad (4.4)$$

$$u = \frac{-2 \delta a_{10} - 2 a_{02} a_{10} + a_{01} a_{11}}{-a_{11}^2 + 4 \delta^2 + 4 \delta a_{02} + 4 a_{20} \delta + 4 a_{02} a_{20}}$$

In cases that this fails to produce results within  $[-1 \dots 1]$  we have to check for that and clamp the results to -1 or 1.

## 5. Discrete Search

The previous method can be applied in situations where the expected motion is maximum 1 pixel. If motion is bigger than this the method can not work well and even extending the search window is likely to be problematic because there might be more than one local maxima within this window and the quadratic cannot model them. So for bigger search windows we use discrete search. In our experiments we assumed 2 pixel maximum motion at this resolution.

In order to search for 2-pixel movement, we should compute 25 values for  $S(u, v)$  for  $u, v = -2..2$  and find the minimum. This is a lot of computation (and a lot of cpu-memory traffic) so we can do it only for the lowest resolution. And even for the lowest resolution we would be hard pressed to do 3 pixel movement because this would need 49 convolutions.

## 6. Running Sum Convolution

The fastest way to compute the convolution of an image with a uniform template is running sum convolution separately in the two dimensions. For the horizontal dimension (e.g. convolution with a  $(2k + 1) \times 1$  template) the value of every pixel of the resulting image is the sum of the  $2k + 1$  horizontal neighbors (including the pixel itself).

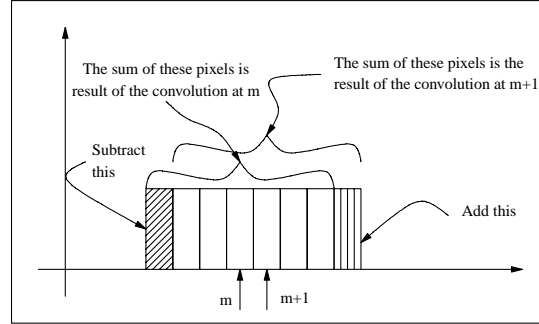
$$I^r[l, m] = \sum_{i=-k}^k I[l, m - i]$$

and

$$\begin{aligned} I^r[l, m + 1] &= \sum_{i=-k}^k I[l, m + 1 - i] \\ &= \sum_{i=-k}^k I[l, m - i] - I[l, m - k] + I[l, m + k + 1] \quad (6.1) \\ &= I^r[l, m] - I[l, m - k] + I[l, m + k + 1]. \end{aligned}$$

From Eq. (6.1) we see that we can compute the running sum for one pixel by subtracting the pixel to the left of the running sum and adding the pixel to the right.

The horizontal running sum convolution works quite fast. But the vertical is about twice as slow because the cache is used very inefficiently. Instead of accessing pixels sequentially in a row



**Figure 6.1:** If we want to compute the result of the convolution at pixel  $m + 1$  we can add  $I[l, m + k + 1]$  and subtract  $I[l, m - k]$

we access them vertically and thus do not use the cache pre-fetching that is available on most modern machines. The solution is to do all computations on whole rows of the image instead of one pixel at a time. This way the vertical convolution is about as fast as the horizontal.

## 7. Applying the Guess and Smoothness

The way we computed flow makes result of the computation independent from pixel to pixel. This means that the effect of noise in the input is pixel-wise independent noise in the output, in other words white noise. While this is not strictly true, because neighboring pixels use mostly the same data, Murphy's Law seems to take precedence over statistics. The end result is that instead of using the guess to bring the second image closer to the first, we wrinkle it instead, making the matching harder. Other algorithms use some form of smoothness constraint, or regularization to iron the wrinkles. Since this is too expensive we used simple 5 tap Gaussian filtering with good results. So before expanding the image to the next higher resolution we smooth it with a  $5 \times 5$  separable convolution.

The other costly part of the hierarchical approach is the cost of scaling down. We have to do low-pass the image to avoid frequency aliasing. We do this with a  $7 \times 7$  separable convolution.

The third part of the hierarchical approach is to combine the "guessed" and the computed residual flow. This is quite complicated but a little careful housekeeping goes a long way to reduce it to little more than three warps, one for the image and two for the  $u$  and  $v$ . In more detail, assume that level 0 is:  $60 \times 80$  size, image is:  $I_1^{R80}$  and  $I_2^{R80}$ .

and flow is:  $U^{R80}$  and  $V^{R80}$ .

level 1 is:  $120 \times 160$  size. image is :  $I_1^{R160}$  and  $I_2^{R160}$   
and flow is:  $U^{R160}$  and  $V^{R160}$ .

level 2 is:  $240 \times 320$  size image is :  $I_1^{R320}$  and  $I_2^{R320}$   
and flow is:  $U^{R320}$  and  $V^{R320}$ .

level 3 is:  $480 \times 640$  size image is :  $I_1^{R640}$  and  $I_2^{R640}$   
and flow is:  $U^{R640}$  and  $V^{R640}$ .

- (1) We compute  $U^{R80}$  and  $V^{R80}$  of level 0, image size is  $60 \times 80$ . This  $U^{R80}$ ,  $V^{R80}$  is a maximum 2-step movement. Since they represent backward warping between images  $I_1$  and  $I_2$ .

$$I_2^{R80}[i, j] = I_1^{R80}[i + U^{R80}[i, j], j + V^{R80}[i, j]]$$

- (2) We scale  $U^{R80}$  and  $V^{R80}$  to size of  $120 \times 160$ ,  $U^{R160,guess}$ ,  $V^{R160,guess}$ , to upper level, that is level 1. Also we need to double every element of  $U^{R80}$ ,  $V^{R80}$  before we scale.
- (3) So we can get a guess destination image ( $I_2^{R160,guess}$ ), using  $U^{R160,guess}$  and  $V^{R160,guess}$ . and warping source image  $I_1^{R160}$  with  $U^{R160,guess}$ , and  $V^{R160,guess}$ .

$$\begin{aligned} I_2^{R160,guess}[i, j] = \\ I_1^{R160}[i + U^{R160,guess}[i, j], j + V^{R160,guess}[i, j]] \end{aligned} \quad (7.1)$$

- (4) We compute the residual U and V at level 1, defined as  $U^{R160,res}$ ,  $V^{R160,res}$  that makes :

$$\begin{aligned} I_2^{R160}[x, y] = \\ I_2^{R160,guess}[x + U^{R160,res}[x, y], y + V^{R160,res}[x, y]] \end{aligned} \quad (7.2)$$

we use maximum 1-pixel movement algorithm to get  $U^{R160,res}$ ,  $V^{R160,res}$ .

- (5) From Eq. (7.1)and Eq. (7.2), we get:

$$\begin{aligned} I_2^{R160}[x, y] = I_1^{R160}[x + U^{R160,res}[x, y] + \\ U^{R160,guess}[x + U^{R160,res}[x, y], y + V^{R160,res}[x, y]], \\ y + V^{R160,res}[x, y] + \\ V^{R160,guess}[x + U^{R160,res}[x, y], y + V^{R160,res}[x, y]]] \end{aligned} \quad (7.3)$$

- (6) So we can combine  $U^{R160,guess}$ ,  $U^{R160,res}$  and  $V^{R160,guess}$  and  $V^{R160,res}$  to obtain  $U^{R160}$ ,  $V^{R160}$   
From Eq. (7.3), we get to know:

$$\begin{aligned} U^{R160}[x, y] = U^{R160,res}[x, y] + \\ U^{R160,guess}[x + U^{R160,res}[x, y], y + V^{R160,res}[x, y]] \\ V^{R160}[x, y] = V^{R160,res}[x, y] + \\ V^{R160,guess}[x + U^{R160,res}[x, y], y + V^{R160,res}[x, y]] \end{aligned}$$

as U and V definition:

$$U^{R160} = U^{R160,res} + \text{warp}(U^{R160,guess}, (U^{R160,res}, V^{R160,res}))$$

$$V^{R160} = V^{R160,res} + \text{warp}(V^{R160,guess}, (U^{R160,res}, V^{R160,res}))$$

- (7) backtrack to upper level

Now we get the final  $U^{R160}$ ,  $V^{R160}$ , which are the combined  $U^{R80}$  and  $V^{R80}$  from low level for a guess, and the residual. We repeat from step 1 to step 6, for higher level of resolution, until we get the final resolution of U and V

## 8. Experiments

We conducted experiments with both real and synthetic images. The real images permit qualitative evaluation in realistic conditions but it is impractical to get ground truth. The synthetic image experiments allow more control over the experiment but are not by definition realistic. We report on them both in terms of performance and in terms of accuracy.

All experiments we report were done on  $320 \times 240$  gray scale images and we used 3 levels of resolution down to  $80 \times 60$  the last level using straight search. The program was standalone, after we replaced the last MediaMath module from it. The timings were with the display features disabled to get more accurate estimate of the performance.

### 8.1. Performance

The total time for image of size  $320 \times 240$  is 0.5 second per frame on a Sun machine Enterprise-450 with 4x300MHz CPU, using one CPU only. The same program on an 300MHz Ultra-10 Sun workstation for an image of size  $320 \times 240$  is 0.7 second / frame. Although the CPUs are similar, the bigger machine appears be have much bigger memory bandwidth. There was of course no difference between real and synthetic images.

### 8.2. Real Images

Two sequences of 30 images were used for the real image experiments. The images had many discontinuities, specularities and noise although they had very little saturation. They were taken with the inexpensive camera the computer came with. The light was a mixture of diffuse sunlight and fluorescent. The scene depicts a small number of toy cars and plastic animals on a large textured piece of paper that is moved by hand. The motion

is a combination of rotation and translation that is at most 5 pixels. Some animals moved smoothly, others are unstable and rock. The motion was mostly horizontal. The car in the second sequence moved smoothly.

The results are shown in Fig. 8.1 and Fig. 8.2. The hippopotamus is moving the most and its head appears as very bright. The contour of its legs as well the contours of the goat, the buffalo and the dinosaur are visible. Considering the difficulty of recovering motion around discontinuities, this result is very good [12, 4]. The second sequence shows the some effects of the aperture problem. Large areas of constant gray can be shown as moving due to slight variations of intensity.

### 8.3. Synthetic Images

We created synthetic images that contain a plaid pattern with varying amount of flow. They were created using MediaMath and saved to files to be used by the optical flow program.

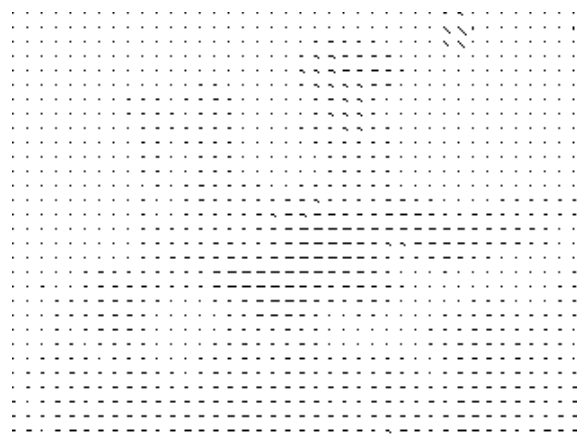
The synthetic image pair and the computed flow are in Fig. . There is one spot in the image that the algorithm failed but the rest is remarkably accurate. The mean square error excluding this spot is about 3% or about evaluated in [2], that had error in the same range.

## 9. Conclusions and Future Work

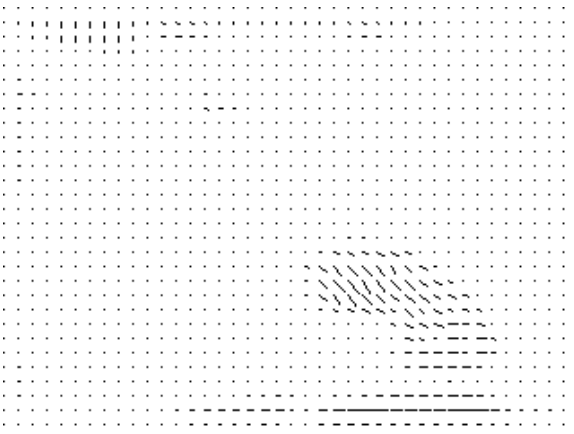
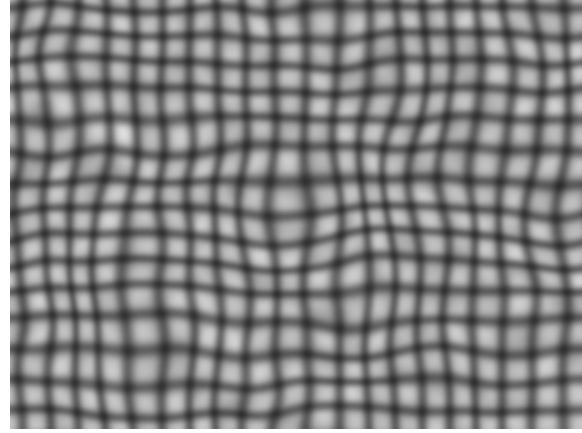
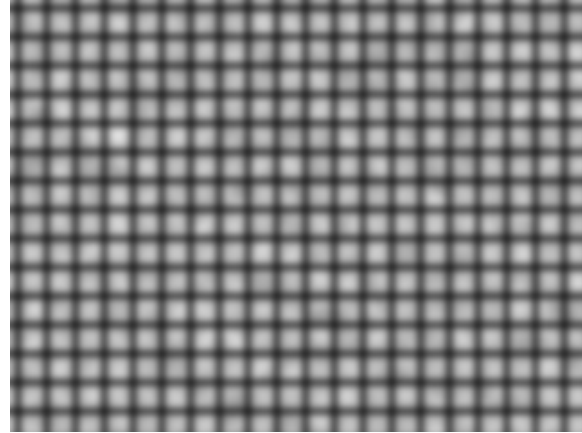
The definition of real time in this context varies with the application. Certainly there are many applications where 2 frames per second is not real time. Can this be done any faster with today's technology?

More than half of the computation was taken by the convolutions. The ones related to the quadratic fitting were done in floating point arithmetic. We expect a considerable improvement when we switch to short integers for two reasons. One is that the data is smaller and can be moved back and forth through the bus faster and integer operations using the VIS instruction set are much faster. We expect to be able to test it soon.

Most new processors come with built in acceleration for video applications. This is true for Sun's SPARC and Intel's new Pentium. This along with Moor's Law will make it possible to do it in TV frame rate soon.

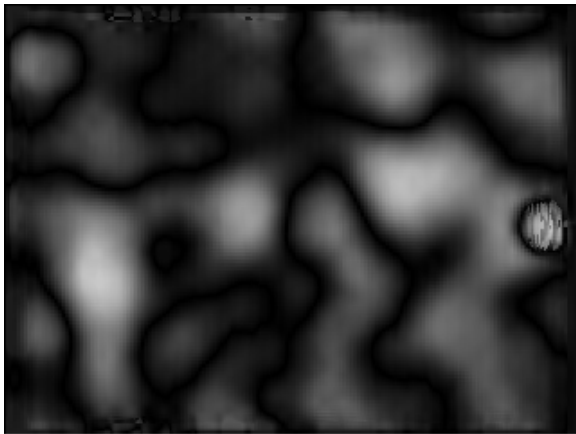
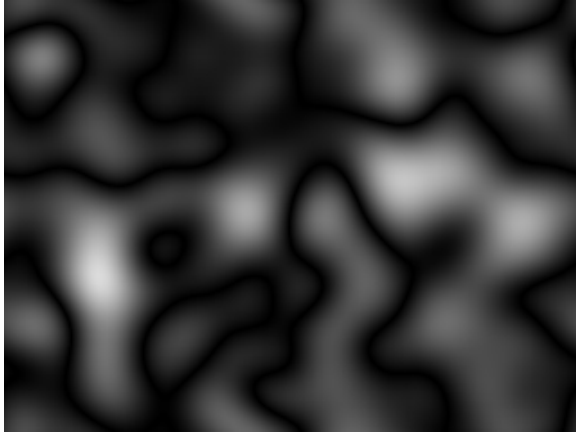


**Figure 8.1:** The top picture is one frame of the sequence of the real images and the middle is the absolute value of the computed horizontal flow as intensity.



**Figure 8.2:** The top picture is one frame of the sequence of the real images for a car moving. The middle picture is the absolute value of the computed horizontal flow as intensity. The bottom one is the needle map of flow.

**Figure 8.3:** These are two synthetic images. The plaid is 16 pixels wide and the maximum flow was 7.5 pixels. Although not clearly visible in print the image contains some fine texture.



**Figure 8.4:** This is the synthetic and the computed horizontal component of the flow. With the exception of the borders and the spot to the right the result was within about 3% of the ground truth.

## References

- [1] P. Anandan, "A Computational Framework and an Algorithm for the Measurement of Visual Motion," *Int'l J. of Computer Vision* **2** pp. 283-310 (1989).
- [2] J. L. Barron, D. J. Fleet, and S. S. Beauchemin, "Performance of Optical Flow Techniques," *Int'l Journal of Computer Vision* **12** pp. 43-77 (1994).
- [3] J. L. Barron and M. Khurana, "Determining optical flow for large motions Using Parametric Models in a Hierarchical Framework," *Vision Interface, Kelowna, B.C.*, pp. 47-56 (1997).
- [4] S. S. Beauchemin and J. L. Barron, "The Computation of Optical Flow," *ACM Computing Surveys* **27**(3) pp. 433-467 (1995).
- [5] J. R. Bergen, P. Anandan, K. J. Hanna, and R. Hingorani, "Hierarchical model-based motion estimation," *ECCV92, Santa Margherita, Italy*, pp. 237-252 (1992).
- [6] D. J. Fleet and K. Langley, "Toward Real Time Optical Flow," *Vision Interface*, pp. 116-124 (1993).
- [7] B. K. P. Horn and B. G. Schunck, "Determining Optical Flow," *Artificial Intelligence* **17** pp. 185-204 (1981).
- [8] A. Jepson and M. Black, "Mixture models for optical flow computation," *CVPR*, pp. 760-761 (1993).
- [9] J. W. Y. Kam, "A Real-time 3D Motion Tracking System," TR 93-16, Dept. of Computer Science, Univ. of British Columbia (1993).
- [10] E. Karabassis and M. E. Spetsakis, "An Analysis of Image Interpolation Differentiation and Reduction using Local Polynomial Fits," *CVGIP: GMIP*, pp. 183-196 (1995).
- [11] B. Lucas, *Generalized Image Matching by the Method of Differences*, PhD Dissertation, Dept. of Computer Science, Carnegie Mellon University (1984).
- [12] M. E. Spetsakis, "Optical flow estimation using discontinuity conforming filters," *British Machine Vision Conference*, (1994).
- [13] W. Yu, K. Daniilidis, S. Beauchemin, and G. Sommer, "Detection and characterization of multiple motion points.," *CVPR*, (1999).