To appear in The Journal of Real-Time Imaging - special issue on Real-Time Motion Analysis

Real-Time Quantized Optical Flow

Ted Camus*
National Institute of Standards and Technology†

Abstract

Algorithms based on the correlation of image patches can be robust in practice but are computationally intensive due to the computational complexity of their search-based nature. Performing the search over time instead of over space is linear in nature, rather than quadratic, and results in a very efficient algorithm. This, combined with implementations which are highly efficient on standard computing hardware, yields performance of 9 frames per second on a scientific workstation. Although the resulting velocities are quantized with resulting quantization error, they have been shown to be sufficiently accurate for many robotic vision tasks such as time-to-collision and robotic navigation. Thus, this algorithm is highly suitable for real-time robotic vision research.

1 Introduction

For robotic vision to be successful and practical in real-world environments, it must be robust, fast, and sufficiently accurate as appropriately defined for a given task. If an algorithm is not robust and only works on well-specified input, it is useless in the uncertainty of a noisy, real-world environment, regardless of any other desirable qualities it may possess. Similarly, an algorithm that may take many minutes to run is clearly inadequate for a reactive system, and must be run off-line or on expensive special-purpose hardware. Finally, a robotic vision algorithm must return correct measurements as appropriately defined for a given task. In some cases such as obstacle avoidance, *qualitative* computer vision may be sufficient (e.g., [1]). Of course more accuracy is preferable to less, however under no circumstances may the first two requirements of robustness and speed be sacrificed. If these first two criteria are satisfied however, accuracy may then be optimized.

Currently two major limitations to applying vision to real robotic vision tasks are robustness in real-world, uncontrolled environments, and the computational resources required for real-time operation. In particular, many current robotic visual motion detection algorithms (optical flow) may not be suited for practical applications such as collision detection on a mobile robot because they either require highly specialized hardware or up to several minutes on a scientific workstation. For example, [2] quotes 4 minutes on a Sun[1] workstation and 10 seconds on a 128-processor Thinking Machines CM5 supercomputer. Although good quantitative results are reported, such an algorithm would need to wait approximately 16 years for the 1000-fold increase in performance necessary for real-time rates on an ordinary workstation, given that computer processing power increases approximately 54 % per year [3]. In addition, many such algorithms depend on the computation of first and in some cases higher numerical derivatives, which are notoriously sensitive to noise. In fact the current trend in optical flow research is to stress accuracy (often under idealized conditions such as modeling the noise as Gaussian) over computational resource requirements and robustness against noise, both of which are essential for real-time robotics. Another trend is to compute only *normal* flow [4] instead of the true optical flow. Although normal flow can be computed much more easily than full optical flow and may have sufficiently robust statistical properties for many useful tasks (e.g., [5, 6]) there will be many cases where full flow is preferable, if it can be computed efficiently and robustly.

Nishihara [7] lists four criteria for a successful computer vision algorithm: noise tolerance, practical speed, competent performance, and simplicity. The first three correspond directly to robustness, real-time performance, and sufficient accuracy as defined for a specific task. *Simplicity* is desirable so that the algorithm in question may be easily analyzed. This paper

*This research was conducted while the author held a National Research Council Research Associateship at NIST.

†Author's address: Intelligent Systems Division, Bldg 220 Rm B-124, Gaithersburg, MD 20899 USA. Email: tac@cme.nist.gov. URL: http://isd.cme.nist.gov/staff/camus/

[1]Certain commercial equipment, instruments, or materials are identified in this paper in order to adequately specify the experimental procedure. Such identification does not imply recommendation or endorsement by NIST nor does it imply that the materials or equipment identified are necessarily the best for the purpose.

suggests an alternative. If the algorithm may be made to run in real-time on standard, general-purpose computing hardware, then the run-time performance of even complicated algorithms may be analyzed using standard software tools and visualization techniques, without having to fight the intricacies of some special-purpose image processor.

Many techniques for optical flow exist (e.g., [2, 8, 9]; see also [10, 11, 12] for reviews and discussions of several techniques). Although many of these techniques can perform very well for certain sequences of images, there are very few that are currently able to support real-time performance without special-purpose hardware. One obvious reason calculating optical flow is so computationally intensive is that images are composed of thousands of pixels (which often motivates massively parallel implementations). One option is to only calculate optical flow at areas of high contrast where the flow measurements are more reliable. However, many applications of optical flow require dense outputs which are difficult to compute in areas of low contrast. For the purposes of real-time robotic vision, it is desirable to find a method of calculating optical flow that is less sensitive to noise in the imaging process, gives a dense output independent of the structure in the image, and is computationally efficient.

Section 2 provides a general description of traditional correlation-based optical flow methods. Section 3 describes the time-space tradeoff used to achieve real-time correlation-based optical flow. Section 4 compares the measurements calculated with the optical flow algorithm described in this paper with those computed by traditional correlation-based algorithms. Section 5 examines some practical applications and gives examples. Section 6 gives a quantitative analysis of the efficiency and accuracy of the algorithm described in this paper including comparisons with other algorithms, and discusses other related issues such as subsampling and latency. Section 7 gives details of the real-time software and hardware implementations of this algorithm.

2 Correlation-based Optical Flow

In correlation-based flow such as in [8, 13, 14] the motion for the pixel at [x,y] in one frame to a successive frame is defined to be the determined motion of the patch P_ν of $\nu * \nu$ pixels centered at [x,y], out of $(2\eta + 1) * (2\eta + 1)$ possible displacements (where η is an arbitrary parameter dependent on the maximum expected motion in the image). We determine the correct motion of the patch of pixels by simulating the motion of the patch for each possible displacement of [x,y] and considering a match strength for each displacement. If ϕ represents a matching function which returns a value proportional to the match of two given features (such as the absolute difference between the two pixels' intensity values E_1 and E_2), then the match strength M(x,y;u,w) for a point [x,y] and displacement (u,w) is calculated by taking the sum of the match values between each pixel in the displaced patch P_ν in the first image and the corresponding pixel in the actual patch in the second image:

$$\forall u, w : M(x, y; u, w) = \sum \phi(E_1(i, j) - E_2(i + u, j + w)), (i, j) \epsilon P_\nu \quad (1)$$

The actual motion of the pixel is taken to be that of the particular displacement, out of $(2\eta + 1) * (2\eta + 1)$ possible displacements, with the maximum neighborhood match strength (equivalently minimum patch difference); thus this is called a "winner-take-all" algorithm.

This algorithm has many desirable properties. Due to the two-dimensional scope of the matching window, this algorithm generally does not suffer from the aperture problem except in extreme cases [15], and tends to be very resistant to random noise. Since the patch of a given pixel largely overlaps with that of an adjacent pixel, match strengths for all displacements for adjacent pixels tend to be similar (except at motion boundaries), and so the resultant optical flow field tends to be relatively smooth, without requiring any additional smoothing steps. Conversely, any noise in a gradient-based flow method usually results in direct errors in the basic optical flow measurements due to the sensitivity of numerical differentiation. In fact the correlation-based algorithm's "winner-take-all" nature does not even require that the calculated match strengths have any relation whatsoever to what their values should theoretically be; it is only necessary that the best match value correspond to the correct motion. For example, a change in illumination between frames would adversely affect the individual match strengths, but need not change the best matching pixel shift. Conversely, a gradient-based algorithm's image intensity constraint equation would not apply since the total image intensity does not remain constant. Finally, since one optical flow vector is produced for each pixel of input (excepting a small $\eta + \lfloor \nu/2 \rfloor$ border where flow may not be calculated), optical flow measurement density is 100 %.

The independence of one pixel's chosen displacement from all other pixels' displacements motivates massive parallel implementations such as the close-to-real-time implementation on the connection machine described in [8]. Dutta and Weems [13] calculate structure-from-motion using the same basic shift-match-winner-take-all algorithm implemented on the

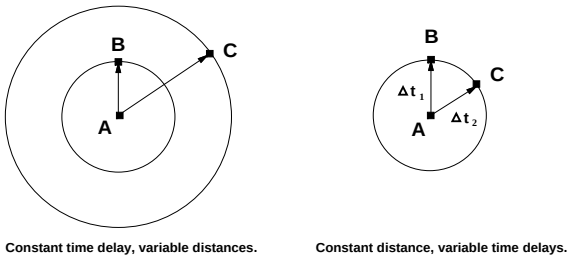

Figure 1: As the maximum pixel shift increases linearly, the search area increases quadratically. However with a constant shift distance and variable discrete time delays, search over time is linear.

Image Understanding Architecture simulator; they report an estimated 0.54 seconds per frame using a maximum possible displacement $\eta = 20$. Little and Kahn [14] calculate optical flow (for the purposes of tracking) within a limited radius (+/- 2 pixels vertically, +/- 3 pixels horizontally) using a 7x7 correlation window at 10 Hz on 128x120 images using the Datacube MaxVideo 200. Certainly, without the luxury of specialized hardware, other techniques must be used for practical operation with conventional serial computers due to the search-based nature of the optical flow algorithm itself, which is described next.

3 Time-Space Tradeoff

One limitation with the traditional correlation-based algorithm described in Section 2 is that its time complexity grows quadratically with the maximum possible displacement allowed for the pixel [15, 16]; see the left of Figure 1. Intuitively, as the speed of the object being tracked doubles, the time taken to search for its motion quadruples, because the area over which we have to search is equal to a circle centered at the pixel with a radius equal to the maximum speed we wish to detect.

However, note the simple relationship between velocity, distance and time:

$$velocity = \frac{\delta distance}{\delta time}.$$

Normally, in order to search for variable velocities, we keep the inter-frame delay δt constant and search over variable distances (pixel shifts)[2]:

$$\Delta v = \frac{\Delta d}{\delta t}, \quad d \leq \eta.$$

[2]The notation Δv is used to represent the fact that the process of searching for the correct velocity involves examining multiple discrete values of v, with the difference between any two particular velocities being the variable Δv.

However, we can easily see from Figure 1 that doing so results in an algorithm that is quadratic in the range of velocities present. Alternatively, we can keep the shift distance δd constant and search over variable time delays:

$$\Delta v = \frac{\delta d}{\Delta t}. \tag{2}$$

In this case, we generally prefer to keep δd as small as possible in order to avoid the quadratic increase in search area. Thus, in all examples δd is fixed to be a single pixel. (Note however, there is nothing preventing an algorithm based on both variable Δd and Δt). Since the frame rate is generally constant, we implement "variable time delays" by integral multiples of a single frame delay. Thus, we search for a fixed pixel shift distance $\delta d = 1$ pixel over variable integral frame delays of $\Delta t \in \{1, 2, 3, ...S\}$. S is the maximum time delay allowed and results in the slowest motion detectable, $1/S$ pixels per frame. For example, a $1/k$ pixels per frame motion is checked by searching for a 1-pixel motion between the current frame t and frame $t - k$. The fastest motion detectable is 1 pixel per frame using $S = 1$ (but see below for approaches to detecting faster motion). Thus our pixel-shift search space is fixed in the 2-D space of the current image, but has been extended linearly in time. As before, the chosen motion for a given pixel is that motion which yields the best match value of all possible shifts.

For example consider Figure 2. Here we are trying to calculate the optical flow for pixel (1,1) at the current frame, image T. For a time delay of 1 frame (at top) the optimal optical flow for pixel (1,1) from image $T - 1$ to image T is calculated to be a pixel shift of (1,-1). This is only a temporally local measurement however; it may not be the final chosen motion. At the bottom of Figure 2 the same search is performed, except using image $T - 2$ as the first image. In this case the calculated motion happens to be a pixel shift of (0,1) pixels over 2 frames, or equivalently (0,1/2) pixels per frame motion. If the maximum time delay $S = 2$, then the procedure would stop here, otherwise we would continue processing frames until finally the optical flow from image $T - S$ to image T was calculated as well. The best of all these motions is taken to be the final computed motion for that pixel.

This approach does assume that a motion of $1/S$ pixels per frame is continuous for at least S frames before it can be registered; failure of this assumption can lead to temporal aliasing and the *temporal aperture problem* [17, 18]. However, the temporal anti-aliasing heuristic described in [17, 18] imposed a latency of a single frame before producing results. Although this is

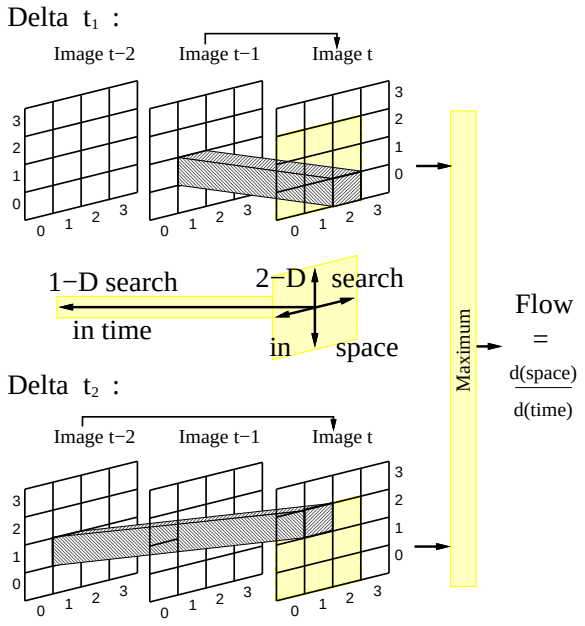

Figure 2: Optical flow is calculated for each individual time delay separately and then combined in a final winner-take-all stage. In this case, $S = 2$.

much less than techniques that smooth with a temporal Gaussian, any latency is undesirable when dealing with real-time robotics. Thus this technique is not employed in the examples used in this paper.

Another result of this approach is that slower-moving objects will take longer to be detected than faster objects. For example an object moving at 1 pixel per frame would be detectable in only one frame, but an object moving 1/5 pixels per frame would remain undetectable until after 5 frames of continuous motion. At this point however all motions within this velocity range would be detectable and would be correctly represented in the current frame without any latency.

Describing velocity in this way (correlating a signal a given distance apart at two separate times) can be traced at least as far back as Reichardt detectors [19]. Reichardt detectors, however, do not attempt to detect 2-dimensional motion and cannot compute precise velocity due to the confounding effect of contrast on the detector. Our algorithm does detect 2-dimensional motion by performing a spatial (as well as temporal) search and by taking the sum of absolute differences of the original and shifted patches rather than the multiplication of two point signals.

It should be stressed once again that although there are *performance* advantages in keeping the search radius small (i.e., with a minimum of one pixel), there is

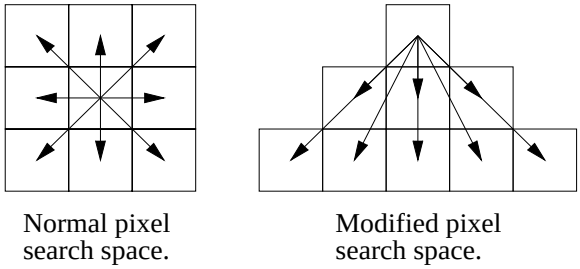

Figure 3:

nothing preventing an algorithm based on both variable Δd and Δt, in which case our algorithm can be viewed as being similar to that of [8], but extended in time. Each pixel shift is treated no differently than any other. Given $\Delta t = 1$, a search radius of two pixels consists of 25 pixel shifts instead of 9, and a search radius of three pixels consists of 49 pixel shifts. Adding one to a search radius of $\eta - 1$ increments the search cost by $8*\eta$ per Δt. To detect motion greater than one pixel per frame we would extend the search space in this manner only for $\Delta t = 1$; doing so for larger Δt would reduce the velocity of the detected motion and thus be counter-productive.

One criticism of this algorithm is that (except as noted in the previous paragraph) the maximum pixel shift detected is 1 pixel. There are two responses to this. First, as discussed in Section 6, block subsampling (which averages NxN blocks of pixels into a single value) is often used to reduce the size of the images processed. In these cases the shift of a single "pixel" at the new, coarser scale is equivalent to a shift of N pixels at the original scale; values of N as large as 8 have been used successfully [28]. Secondly, it can easily be shown that linear search in time combined with a high frame rate is more efficient than a quadratic search in space using a slower frame rate. For example, consider an object moving at 10 pixels per second. An algorithm detecting a maximum of 1 pixel per frame motion would need to run at 10 Hz to detect this object, and would be computing 10 1-pixel radius searches per second. Conversely, an algorithm which can detect up to 5 pixels per frame motion would only need to run at 2 Hz to detect the object, but would be performing 5^2 times the work per frame, for the equivalent of 50 1-pixel radius searches per second. Thus, the linear-time search is 5 times more efficient in this example, and has the bonus of a 5 times quicker update rate as well, since it runs at 10 Hz instead of 2 Hz.

Alternatively, one could simply optimize the pixel-shift search space for a particular type of motion. For

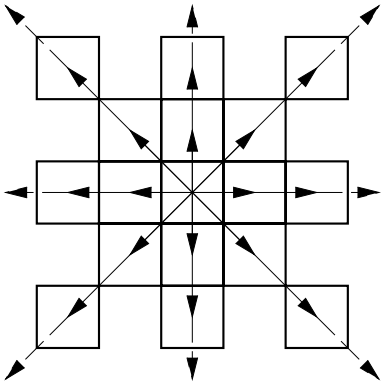

Figure 4: This alternate search space has linear-time complexity but gives bad results.

example, Figure 3 shows the normal pixel search space on the left and a modified search space on the right. The latter is optimized for downward motion and basically assumes that only such motion will occur. Alternate geometries are equally feasible, with computational time being a simple linear function of the number of pixel shifts examined. Thus it is not *easier* to compute flow that occurs within a limited radius, it is merely *faster*. This situation is different from a simple gradient technique, which is in general limited to nearby pixels, unless multi-resolution techniques are employed. Multi-resolution techniques may also be used in correlation-based optical flow such as in [20], and could be used in conjunction with our algorithm as well. Pyramid techniques can be used to good effect when large areas of the image move coherently, but are less effective when there are multiple velocities present [11], which is more likely to occur at coarser resolutions [17]; in any event hierarchical techniques are not inconsistent with the methods described in this paper. However, a fast optical flow algorithm can support faster frame rates where any given motion per frame would be less than if a slower frame rate were used, in which case it would not be necessary to track motion over several pixels in the first place.

It has been suggested that a modified search space such as Figure 4 could be used to retain the linear-time computational complexity shown in Figure 1 but still maintain the full spatial resolution of the output. In Figure 4 the search is performed along eight directions in space as before, but continued for more than one pixel. This will not work however; as the magnitude of the flow increases there will be "gaps" in the search space where the motion vector would be undetected (unless the image were composed of only very low spatial frequencies, an unreasonable assumption).

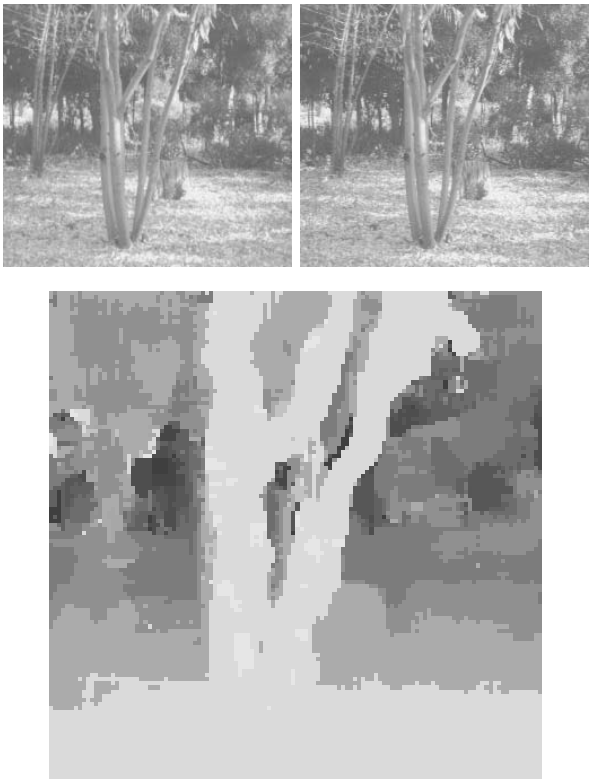

Figure 5: Two images from the SRI tree sequence along with a grey-level image indicating the magnitude of the resulting optical flow.

Figure 5 shows two images from the SRI tree sequence, subsampled to 116x128 pixels in size, along with a grey-level plot of the magnitude of the resulting flow obtained using our algorithm (in this example all flow results from the translation of the camera orthogonal to the line of sight). For all sequences in this paper, $S = 10$ and $\nu = 7$.

The time-space tradeoff discussed in this section reduces a quadratic search in space into a linear one in time, resulting in a very fast algorithm: optical flow can be computed on 64x64 images, calculating 10 Δt per frame (corresponding to 10 distinct velocity magnitudes), at up to 9 frames per second on an 80 MHz HyperSPARC computer. Since the algorithm's computational complexity is linear in the number of Δt detected, the algorithm could also calculate 18 Δt at 5 frames per second, or 3 Δt at about 30 frames per second.

4 Harmonic Search Intervals

We have seen that by performing searches in time instead of space we can, in theory, convert a quadratic time algorithm into a linear one. In this section we will

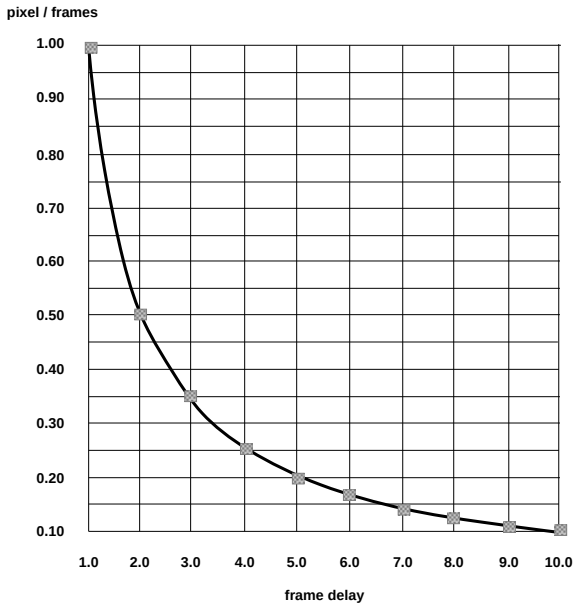

Figure 6: Only 1/(frame delay) velocity measurements can be detected in the current implementation.

more closely examine this point.

One disadvantage of the traditional algorithm is that it computes image velocities that are integral multiples of pixel shifts [8, 15]: {1,2,3,...,η} pixels per frame. Although the algorithm discussed in this paper does calculate sub-pixel motions, it still computes velocities that are basically a ratio of integers (with the numerator equal to one pixel in this implementation), not a truly real-valued measurement. Given $\delta d = 1$ and discrete frame delays $\Delta t = \{1, 2, 3, ..., S\}$, equation 2 yields $\{1/1, 1/2, 1/3, ..., 1/S\}$ pixels per frame equivalent motion (Figure 6). An immediate consequence of this is that sub-pixel motions are now detectable, correcting a deficiency in the traditional algorithm. Although the linear set of velocities may seem more intuitive than the harmonic series, in fact the latter is often much more suited to the types of motion found in real vision problems such as those dealing with perspective projection.

A simple example of such a problem is the calculation of depth from motion parallax. It is known that the apparent motion of an object in an image resulting from the motion of a camera is inversely proportional to the object's distance from the focus of projection, as shown in Figure 7 [21]. If the velocities in the image are small, we can approximate

$$\Delta\theta \approx \frac{v\Delta t \sin\theta}{d} \qquad (3)$$

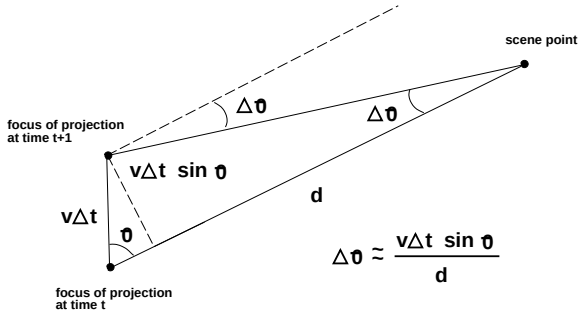

Figure 7: Change in visual angle is inversely related to depth. [21]

where θ is the visual angle[3].

An example of using motion parallax to determine relative depth is shown in Figure 5. Our optical flow algorithm, which computes velocities inversely proportional to the discrete frame delays $\Delta t = \{1, 2, 3, ...S\}$ is therefore very well suited to computing depth via motion parallax. It would be much more awkward to attempt to calculate motion parallax using a linear set of motions of $\{1, 2, 3, ..., \eta\}$ pixels per frame, although it could possibly be done with very high resolution images (with resultingly high computational cost). Although there is a sine factor in equation 3, note that sine "degrades gracefully" with small deviations from 90°, e.g., a 10° deviation from orthogonal motion to the line of sight is a factor of 0.985, and a 20° deviation (quite large for even a moderately calibrated mobile robot) still yields 0.94. Even a very large 30° deviation from the orthogonal yields only a 0.866 factor. To complete the argument, even if the deviation was a ridiculous 45° (i.e. the robot was heading as much parallel to the line of sight as orthogonal to it!) the factor is still only 0.707. Thus, we do not need anywhere near perfect orthogonal motion to make practical use of depth from motion parallax.

It should be noted that it is not necessary to limit the measured velocities to a strict harmonic sequence. Given that the motion to be detected lies within our slowest and fastest measurements (1/1 and $1/S$ pixels per frame), and in the absence of bad spatial aliasing, we need only make measurements as precisely as we need. For example, we could detect a range of veloc-

[3] Although visual angle is a measure used with spherical perspective projection, rather than planar perspective projection which is more appropriate for a machine vision system, both projection models are approximately equivalent near the line of sight where $\tan\theta \approx \theta$. Although the former has the advantage of being independent of any coordinate system, the latter more accurately characterizes a machine vision system. See also [22], Appendix.

ities $v = \{1/1, 1/2, 1/4, 1/8, .., 1/(2^{\lfloor log_2 S \rfloor})\}$ pixels per frame. In effect, this means that we have the option of detecting up to a quadratic range of velocities, again at only a linear-time cost. Of course, the resolution within these ranges is reduced by the same factor, but this may be a useful trade-off for a particular application. Conversely, it is generally not possible for the traditional algorithm to "skip" pixels, since the actual motion may be missed.

5 Applications

Our algorithm has been successfully used on many real and synthetic image sequences for a variety of real-time robotic vision tasks [16, 17, 23, 24, 25, 26]. Camus and Bülthoff [16] first introduced the linear-time space-time tradeoff and a method for dealing with the problem of temporal aliasing. Using a Sparc 2 this algorithm could reliably calculate velocities corresponding to 2 Δt's per frame at up to 8 frames per second. This low resolution was sufficient for simple qualitative tasks involving motion detection, however the technique used to deal with temporal aliasing was shown not to be extensible as the range of velocities increased [18].

Duchon and Warren [23] discusses *direct perception* and *ecological robotics* in the context of obstacle avoidance, and reports being able to use this optical flow algorithm to instantiate some fly-like control laws [27] in a small mobile robot. With the camera mounted on top of the robot, the image information from the camera is fed through the optical flow routine and the motion information is then used to directly control the robot. With a frame rate of 4 to 5 frames per second and the robot moving at 4 to 5 cm per second, the robot is able to successfully maneuver through an unmodified office environment with a theoretical maximum turning rate of 12° per frame.

Camus [17] uses the quantized, real-time optical flow algorithm to compute time-to-collision with a single flat surface. Although the individual motion measurements of the optical flow algorithm are of limited precision, they can be combined across space and time using both robust statistics and simple averaging to produce remarkably accurate time-to-contact measurements, most measurements being within one frame of the actual value as far away as 100 frames from contact. This accuracy was maintained up to 2 frames from contact with the surface. Later Camus [24] showed that similar accuracy could even be produced at video rates using our algorithm on standard computing hardware.

McCann [25] aims a camera down at the floor in front of a moving robot to detect obstacles. Flow is computed using our algorithm but only the horizontal component is used to detect obstacles, since the vertical component is generally maxed out across the visual field in this application. The 64x64 array of horizontal flow data is then subsampled to 32x32 and interpreted using a neural network. The network was trained on real data, using 150 examples of a person walking in front of the robot. The robot's direction would then be altered based on position of detected obstacles and the robot's current location in the corridor.

Figure 8: Three frames showing an approach to metal chairs, with a magnitude plot of the optical flow for the third frame; brighter points indicate faster motion.

Duchon, Warren and Kaelbling [26] extends the *ecological robotics* work of [23] to include specific action modes. The real-time optical flow algorithm, running at 4 Hz, is used to play the game of tag with a cardboard target, and even with a slow-moving person. The actions of pursuit, docking, and escape are implemented using only optical flow, without any intermediate representations such as object segmentation.

A primary application for our algorithm is real-time obstacle avoidance. Figures 8 and 9 show 3 frames (spaced 10 frames apart) of an approach to some metal chairs, along with plots showing the magnitude of the flow vectors (where brighter points indicate faster motion), and optical flow "needle" plots for the last frame of this sequence. The two peripheral chairs are clearly differentiated from the background motion in the magnitude plot; this information alone could be used for obstacle avoidance. Figure 10 is a more difficult sequence showing 3 frames of an approach to wooden

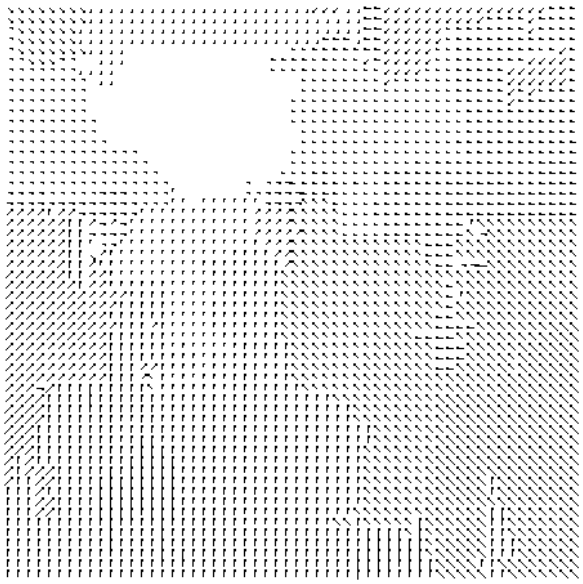

Figure 9: The optical flow "needle" plot for the third frame of Figure 8. The focus-of-expansion is clearly seen.

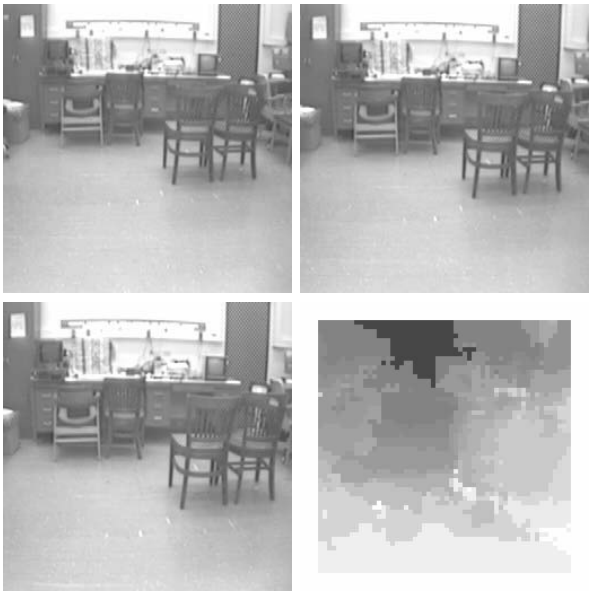

Figure 10: A more difficult sequence showing three frames of an approach to wooden chairs, with the magnitude plot. The chairs on the right show clearly despite their slow image motion.

chairs, whose edges are not as sharply defined against the background. The pair of chairs on the right are also clearly differentiated in the magnitude plot despite the relatively slow motion. In addition note that the optical flow on the floor is also very well defined despite the lack of texture and sharp edges. These images were taken at a rate of 5 frames per second, with the robot moving at 12 centimeters per second. The central 256x256 pixels (shown) were taken from a 512x512 image using a 115° field-of-view camera, and were subsampled to 64x64 pixels in size using a basic block subsampling technique. Subsampling by averaging each NxN block of pixels into a single 8-bit value can be done extremely quickly without special hardware; no other temporal or spatial smoothing was performed. The flow plots are generated and displayed in real-time simultaneously with the image flow computation; a special routine converts real-valued optical flow values into one of 161 possible 8x8 bitmaps. The resulting flow plots are 56x56 pixels in size, due to the $\eta + \lfloor \nu/2 \rfloor$ border. The real-time display typically slows down the frame rate by about 25 %. Example animations are available via the WWW at the URL listed in the author's address.

Current work is being done to use this real-time optical flow algorithm in place of the normal flow algorithm used for obstacle avoidance in [6]. The algorithm used in [6] did run in real-time on an Aspex PIPE but only gave normal flow at edges, resulting in sparse data especially in areas of low texture. In addition, the PIPE returned only a limited range of optical flow data, which made time-to-contact estimates very sensitive to noise near the FOE where the flows are small. Our algorithm has been shown to require very little texture (as shown in Figures 10 and 11) and can usefully detect very small flows given constant translational motion [17].

6 Quantitative Analysis

One disadvantage of the patch-matching approach is that the basic motion measurements are integer multiples of pixel-shifts. Although the algorithm discussed in this paper does calculate sub-pixel motions, it still computes velocities that are basically a ratio of integers, not a truly real-valued measurement. (Calculation of real-valued optical flow measurements using interpolation is described in [28].) It must be determined how seriously the quantization of motion measurements affects accuracy. This section describes quantitative results using the "diverging tree" sequence from [29] and [11]. Barron, *et al.* [11] notes that matching techniques typically perform worse on this sequence due to subpixel inaccuracy. Images which contain diverging motion have a non-uniform motion field everywhere and violate the translational model usually used by matching techniques. Despite this apparent weakness in matching

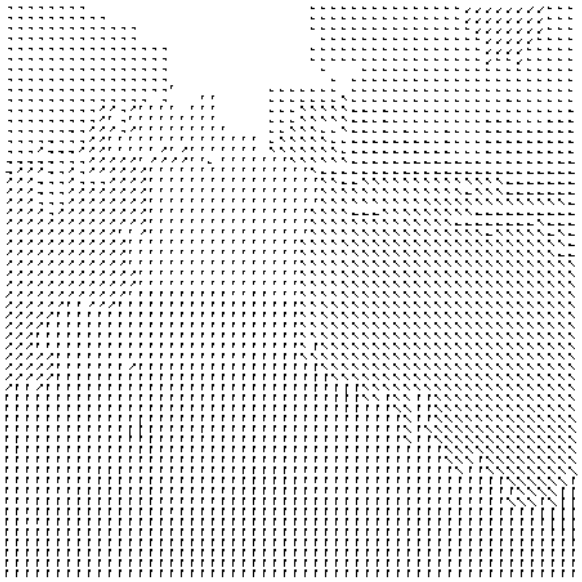

Figure 11: The optical flow plot for Figure 10. The flow on the floor shows clearly despite a lack of texture and sharp edges.

algorithms for diverging image sequences remarkable accuracy has already been shown for this real-time optical flow algorithm in *applications* such as calculating time-to-collision to sub-frame precision on diverging image sequences [17], even at video rates on an ordinary workstation [24]. The results reported here show that although this algorithm's accuracy is not as high as that of other algorithms, it is significantly faster than competing algorithms.

Bober and Kittler [30] describes a "robust hough" technique in multiparameter space for computing motion. Since an exhaustive search would be too computationally expensive, a gradient-based search in multiresolution image and parameter spaces is performed, with the initial coarser resolutions helping to avoid local minima. Using an 80 MHz HyperSPARC computer an execution time of 123 seconds is estimated for 100 % dense results [12]. A faster algorithm yielding much coarser results (similar to the sparse spatial sampling of [31]) would result in an algorithm of about a couple of seconds on an 80 MHz HyperSPARC. Accuracy results are not reported for this faster algorithm, but would depend on the effectivness of the affine model within each window.

Liu [32, 33] designs 3-dimensional Hermite polynomial differentiable filters and a general motion model that includes translation, expansion and rotation to compute motion. Accuracy can be sacrificed for increased speed by using simpler motion models (for example, using only a translation motion model) and by using Hermite polynomials of up to 2nd order rather than 3rd order.

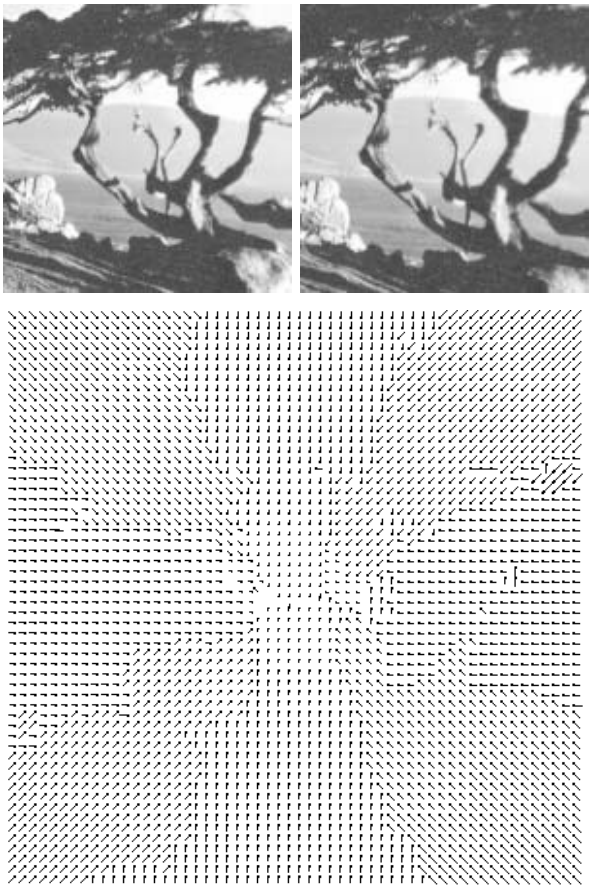

Figure 12: Two frames from the diverging tree sequence along with an optical flow needle plot (subsampled to 64x64 pixels in size for clarity) for the latter frame using $S = 10$.

The diverging tree sequence images are 150x150 pixels in size. Figure 12 shows two images from the diverging tree sequence, with the resulting optical flow (subsampled to 64x64 pixels in size for clarity). For this sequence $S = 10$ was used with the pixel patch size $\nu = 7$. Using the error measure in [11], the average error is 10.374° with standard deviation 7.943°. Increasing the value of S does not measurably decrease error since there are vanishingly fewer pixels close to the focus of expansion with such small motions. Density was 100 %, except for a border of 5 pixels[4]. A grey-level error map is shown in Figure 13, with lighter points indicating lower error. This form

[4]These images are apparently actually 149x149 pixels in size but are padded to give an even 150x150 pixels. For this reason the border was increased an extra pixel.

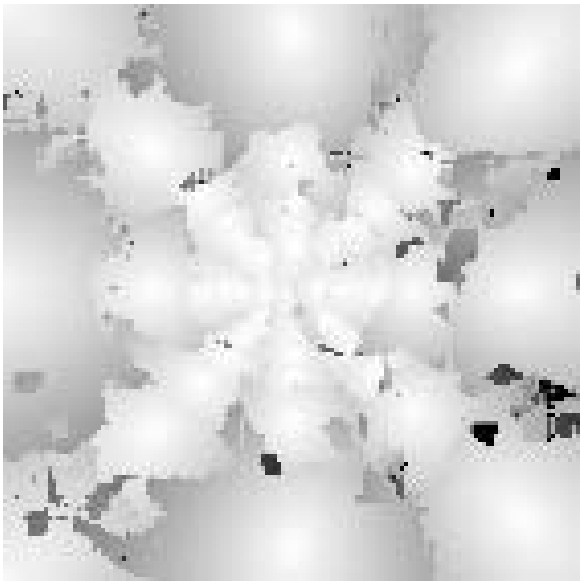

Figure 13: Grey-level error map for Figure 12 (lighter points indicate lower error). The individual velocity quantizations are clearly visible.

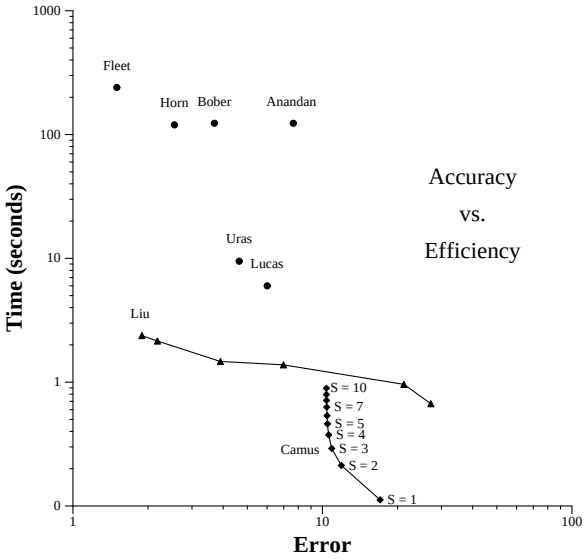

Figure 14: Accuracy-Efficiency graph showing several algorithms' timings on an 80 MHz HyperSPARC computer.

of error plot is especially useful in analyzing the performance of an optical flow algorithm on a sequence which contains only diverging images. This is because each image contains motion in all directions and in all magnitudes in a continuous range from no motion at the focus-of-expansion to the fastest motion present at the corners of each image. Since our algorithm quantizes both the direction and magnitude components of velocity, we can see the quantization error of each such individual "velocity quantization". As would be expected, velocity error is minimal at the center of each velocity quantization and highest at the boundaries between quantizations. Since a harmonic series of motions are detected, the precision of the measurements is less at the higher velocities; thus the worst error (other than random errors) is found at the borders of the quantizations corresponding to the fastest quantized motions detected in the image.

Figure 14 plots the accuracy of several authors' algorithms [20, 30, 34, 35, 4, 33, 36, 31] in terms of angular error (using the measure of [11]) versus efficiency (using seconds per frame) for the diverging tree sequence on an 80 MHz HyperSPARC computer (data on other authors' algorithms are courtesy [12]). As can be seen in the graph, our algorithm is not as accurate as the other authors' algorithms but it is many times faster and is one of only two (along with Liu's [33]) that may be considered for real-time performance on standard computing hardware. The strength of Liu's algorithm is accuracy, and real-time performance is achieved at a great increase in the average error, as shown by the general flatness of the Accuracy-Efficiency curve. The strength of the algorithm reported in this paper is speed, and greater accuracy is achieved by looking further back in time and thus detecting slower motions; in the case of the diverging tree sequence, there are diminishing returns in doing so as shown by the vertical asymptote in the AE curve.

To achieve "interactive-time" rates for robotic vision work, subsampling of the images is often needed. Liu [32] notes that subsampling an image is often detrimental unless low-pass filtering is performed prior to subsampling. Since such filtering is proportional to the original image size the computational advantage of subsampling is lost. The algorithm reported in this paper uses a simple form of block subsampling that averages NxN blocks of pixels into a single value. This subsampling can be performed on a standard workstation faster than frame rate (generally only being limited by the image data I/O bandwidth); more sophisticated techniques which may require special hardware need not be employed. This subsampling technique also has the benefit of averaging out random noise. However, when viewing such a subsampled image sequence it is clear that considerable aliasing is occurring. In fact the aliasing is so bad one wonders how the algorithm can work at all. Figure 15 provides an example of this problem. Here a dark square is translating across a subsampling block boundary

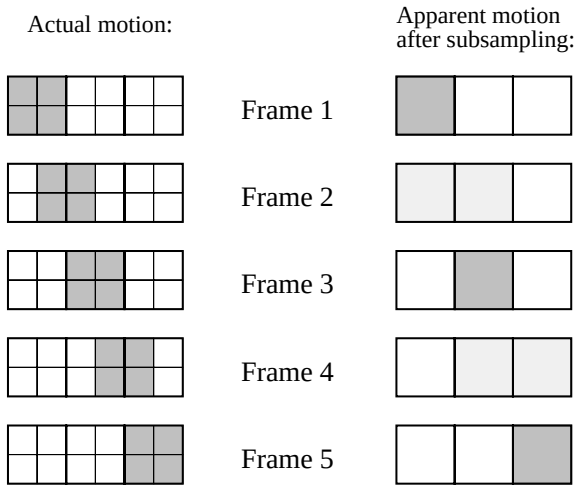

Figure 15: Aliasing due to subsampling.

which results in a flickering effect. For an algorithm measuring spatial derivatives, this could result in inaccurate measurements. Our linear-time correlation algorithm, however, does not measure these types of spatial derivatives. Instead, it performs pixel matching across space and time. In this example, a perfect pixel match can be made between frame 1 and frame 3, as well as between frame 3 and frame 5. Similarly, a perfect match may be made between frame 2 and frame 4 even though the high-contrast area is straddling a subsampling block border. Difficult matches such as that between frame 1 and frame 2 in this example occur only when a high-contrast area straddles a subsampling block border in one frame and does not do so in the other. Such pixels will usually be in the minority within a given patch and will affect the correlation match values somewhat but in general will not change the minimum sum-of-absolute-differences correlation match.

Although in general subsampling does not hurt the performance of this correlation-based algorithm (and helps in noisy images), there is one special case that should be noted. If a single, slanted, narrow band appears against a low-textured background such that the band presents two high-contrast edges within a single correlation patch, then we occasionally see a special instance of the aperture problem, as shown in Figure 16. Here a slanted line is translating slowly to the right, however after subsampling the motion appears to be a fast translation upward. Although this effect can occur with any slanted edge, the sum-of-absolute-differences correlation measure usually performs correctly (a sum-of-squared-difference measure, however, might not). The case of a narrow band is

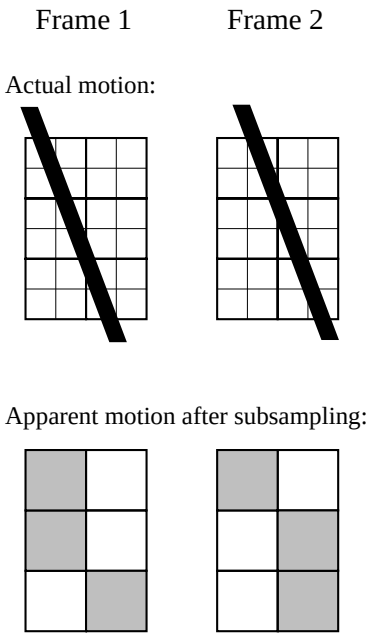

Figure 16: The effect of block-subsampling a narrow, high-contrast band.

special because it actually presents *two* high-contrast edges instead of just one, and is therefore more likely to cause a problem.

One characteristic of this algorithm is that it only utilizes past frames in detecting motion; this gives the advantage of lower latency but the disadvantage of *temporal occlusion*. Temporal occlusion occurs when an object occludes another object in some frame in the past, but disoccludes it in the present due to motion (but not the converse, as described below). The effect of temporal occlusion can be seen in Figure 17

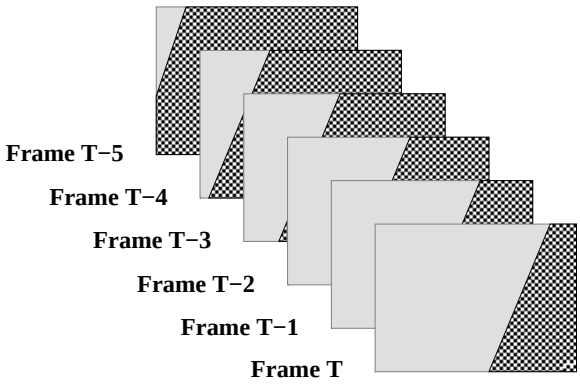

Figure 17: Temporal occlusion

([32]). In Frame T a slow-moving background is being disoccluded by a faster-moving foreground. A pixel towards the center of this frame should be given the motion of the background. However our algorithm detects slow motion by looking back several frames, and in this case, the faster-moving object is occluding the background. For example, the correlation match value for the motion of the slow-moving background between frames T and $T-5$ would be poor due to the occlusion, but the match value for the faster moving foreground between frames T and $T-1$ would not be as bad; therefore this point would incorrectly be given the motion of the foreground rather than the background. This results in the appearance of an extended trailing edge of the foreground object. Thus, fast-moving foreground objects often appear to be larger than their actual extent in the image. Although this effect is inelegant, it can actually be an advantage for obstacle avoidance, since foreground objects are by definition closer to the observer, and also have a faster image motion due to perspective projection; thus this effect tends to exaggerate those objects which are most likely to cause a collision. Note that the converse case of a slow-moving background object being *occluded* by a foreground object does not cause a problem since the motion of the given pixel in the current frame would be correctly assigned the motion of the foreground object. Future work will attempt to factor in the effect of temporal occlusion when calculating the correlation match values.

Algorithms such as [35] use symmetric temporal filtering, and require future frames as well as past frames. Liu [32] notes that at video rates (30 frames per second), incoming frames arrive each 33 ms which is a small amount compared to the 200 ms or so of computation required for a fast 5 frame p[214zer second optical flow algorithm[5]. Although this is true, the amount of motion present in these additional frames (at 33 ms intervals) would be much less than for those whose flow is actually computed (at 200 ms intervals). Given that individual pixels are usually quantized to 8 bits, the much smaller amount of motion may not be enough for accurate flow calculation; most image sequences authors use for testing flow algorithms are captured at a much slower frame rate where the motion per frame is larger. If it is true that the motion required for accurate temporal filtering is larger, then the true latency of such algorithms is worse than implied by [32]. Conversely however, the smaller amount of motion per frame may actually improve the results if aliasing is a problem; this does require sufficient image bandwidth to sustain video rates. Thus the severity of this temporal latency disadvantage is image sequence dependent.

[5]This assumes of course that such additional frames would be used only for temporal filtering and would not have their own optical flow computed, since only every sixth frame could have flow computed at a 5 Hz frame rate.

7 Real-Time Implementation

For computer vision to ever become practical, some consideration must be given to performance issues. The vast computational requirements of real-time computer vision make the actual implementation of such algorithms critical, even with modern hardware. One factor related to performance is an efficient representation and algorithm used to analyze the image(s). In the case of our linear-time optical flow algorithm, this has been addressed by the space-time tradeoff which converts a quadratic search in space into a linear search in time. The other important factor is the hardware implementation.

Although this algorithm is suitable for special-purpose image processors, the controversial view is taken that this particular algorithm is better suited to general-purpose computers because of their general availability and flexibility. A major advantage of our optical flow algorithm is that it can be implemented using only simple integer arithmetic such as adds, subtracts and integer compares, which can execute very quickly compared to floating point calculations. About two-thirds of the computational time of our algorithm is consumed during the patch-matching (equation 1) stage during which the best matching pixel shift is found. A key component of this stage is known as a *box filter*. This single operation is so important to real-time operation that it will be described in some detail here.

The first step of the patch-shift-match operation is the shifting stage [8]. In a massively parallel machine such as the Connection Machine, it might be desirable to implement this as an explicit image shift so that corresponding pixels are located in the same processing node. In the case of a serial processor however, this can be implemented at negligible cost by using normal addressing arithmetic on certain microprocessors. Once the base addresses of corresponding shifted rows are calculated, a single index register can be updated with each new pixel in many computer architectures.

After the shift stage (whether explicit or implicit) each pair of corresponding pixels is compared. This comparison could be either the sum of absolute differences (SAD) of each corresponding pair of pixels' intensity values, or the sum of their squared differences (SSD). Both sum-of-absolute differences and sum-of-squared differences could be implemented with no difference in computational time by using lookup tables

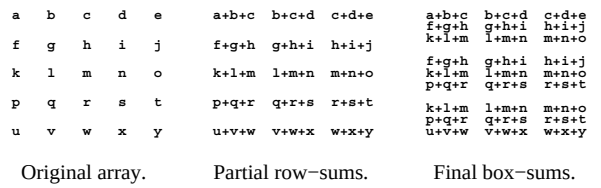

Figure 18: Demonstration of a 3x3 box filter.

for the absolute-value and squared-value functions. Anandan [20] chooses to use the SSD for easier mathematical analysis, since the derivatives of the SAD function are discontinuous at zero. However, for this algorithm the sum-of-absolute-differences has been used since it appears less sensitive to large deviations such as edges produced by shadows, and high-frequency noise; the discussion of robust measures in [17] reinforces this view as well. If a lookup table is not convenient (as it might not be in certain custom hardware), absolute value can be easily calculated on an N-bit 2's complement machine by the expression:

```
abs(x) = (x^(x>>(N-1)))-(x>>(N-1))
```

where '^' is the standard C operator for bitwise XOR and '>>' represents an arithmetic right shift. A dedicated hardware implementation of absolute value (such as in custom VLSI) would in fact be much simpler than implied by this expression since fully functional shifter and subtracter units are not necessary. The shift operation here is merely the replication of the high-order bit and the subtraction operation increments the expression by either one or zero only; these simpler operations would require much less hardware than fully general versions.

The next step is referred to as the "Excitation" stage in [8], or the local neighborhood summation stage, as demonstrated in Figure 18. Here each element of the absolute difference (or squared difference) array is replaced with the sum of all elements in a local neighborhood $\nu * \nu$ in size. In Figure 18, the neighborhood is 3x3 in size. Note that the final box-sums array is not defined on the very edges of the image, since that would require pixels not available in the image itself. A naive implementation would perform an explicit $\nu * \nu$ summation for each pixel location; such an algorithm would be quadratic in ν. A much better way is to perform a box filter, whose running time is independent of the neighborhood size ν. A box filter is performed in two passes, one along the rows of the absolute difference array and another along the columns. The first pass creates the partial row-wise summations, which are then added along the columns to yield the full neighborhood summations. Each pass is performed

```
/*
Select motion with the minimum match value.

 best_motion: current best matching motion
 best_match : the match for best_motion
 cand_motion: a candidate motion
 cand_match : the match for cand_motion
 mask       : a mask of all 0's or 1's

 This code assumes 2's complement integers
 and that there will not be an overflow.
 '>>' represents an arithmetic shift.
*/

mask = (best_match - cand_match) >> (N-1);
best_match  = ( best_match & mask)
              | ( cand_match &~ mask);
best_motion = (best_motion & mask)
              | (cand_motion &~ mask);
```

Figure 19: Direct updating of best motion without conditionals.

by taking the value for the previous location, adding one new absolute difference (corresponding to the new scope of the $\nu * \nu$ neighborhood), and subtracting an old absolute difference (corresponding to the neighborhood of the previous location). This takes advantage of the fact that the neighborhoods of adjacent locations overlap completely except for the most extreme elements. Therefore, a complete box summation consists of only a constant four simple computations, two along the element's row and two along the element's column (except for the very first element of each row or column which initializes the running summations), and this is independent of the size of ν.

The final "winner-take-all" stage is then done based on the match values. In practice on a serial machine it is convenient to compare the current best match value with each new one as it is calculated, using a traditional `IF-THEN` construct. In a massively parallel processor it may be undesirable to require a data-dependent conditional construct in the algorithm (e.g., [37]), since this means that some processors will be executing different instructions depending on the data being processed; this may be difficult or impossible with SIMD (single-instruction-multiple-data) hardware. In this case the updates can easily be performed directly using shift and mask operations, as

shown in Figure 19. Once again, the shift operation performed is merely the replication of the high-order bit and would not require a fully functional shifter. In dedicated hardware, parallelization of this expression could result in as few as four sequential steps per match comparison.

Because each iteration of the box filter reuses the computations of previous pixels, in general we compute the correlation match values of a given pixel shift (over space and time) for all pixels in the image (i.e., it is relatively inexpensive to compute a given pixel shift for all pixels in the image). Conversely, prior knowledge of the motions of certain pixels is of less benefit to this algorithm than it might be for other algorithms; a pixel that cannot take advantage of a previously computed running sum of a neighbor must calculate the initial running sum from scratch, and this operation is $O(\nu^2)$. Therefore, while pixel-search geometries such as in Figure 3 are possible and can be used to speed performance by reducing the search space, there is generally no benefit in prior knowledge of an *individual* pixel's likely motion. The box filter uses an extremely tight loop and incorporating any such knowledge would add more cycles than it would save. An exception would be if it were known that substantial areas of the image each had a uniform pixel-shift search geometry, in which case each large area could be processed as a separate image.

A remarkable fact is that the neighborhood summations can be speeded up significantly by calculating multiple sums in a single register. Normally, a single neighborhood summation would be accumulated into a single register. If we are summing absolute differences of unsigned 8-bit values (the usual representation for images), with a maximum neighborhood size $\nu = 15$, we are guaranteed that the entire sum of the $\nu * \nu$ absolute differences in the neighborhood patch will fit into 16 bits. Furthermore, we are also guaranteed than this sum will never be negative during any point of the summation. Thus, we can load two such values into a register at once, and add or subtract entire registers in a single cycle, effectively doubling the patch summation bandwidth for vertical passes. Similarly, a 64-bit architecture could process 4 such summations each cycle. On some 32-bit SPARC implementations, it is possible to load two registers (with 4 16-bit partial summations) simultaneously using a single assembly instruction. This optimization uses 12 lines of assembly code to implement the vertical pass loop as an inline function call and improves performance by a few percent (other computer architectures have their own specific advantages [17]). It appears that this optimization may only be performed on column-wise summations, where multiple adjacent values may be loaded simultaneously into a single register. For example, a block load of 2 16-bit summations, when stored in row-major order, can be effected by a single load of one 32-bit integer. If it were possible to quickly store the first-pass partial summations in column-major order, then this optimization could be performed a second time. However, attempting to do so appears to introduce too much overhead to be worthwhile.

A fully optimized implementation can calculate optical flow on 64x64 images, calculating 10 Δt per frame, at up to 9 frames a second on an 80 MHz HyperSPARC computer. Naturally, some care must be taken to insure that these optimizations are properly implemented. A proper understanding of pointer arithmetic is essential for optimal performance.

Using the technique of [15] (the same as described in Section 2) on a Datacube MaxVideo 200, [14] calculates optical flow within a limited radius of +/- 3 pixels horizontally and +/- 2 pixels vertically using a 7x7 correlation window at 10 Hz on 128x120 images. Since this implementation uses the same basic correlation measure as that described in this paper, it is instructive to compare the performance of the two. We will use single-pixel-shift units as a basic measure, i.e., the comparison of a single given pixel with a single hypothesized pixel shift. Recall that in the traditional algorithm [8, 15] only two frames are used, and a pixel is constrained to lie within a certain neighborhood, in their case 7x5. Motion is calculated everywhere except for a border region where motion calculation would require pixels not available in the image itself. This border region is the sum of the maximum pixel displacement in any direction plus half the correlation window; thus the "active" image area is (128-2(3+3))*(120-2(2+3)) = 116*110. Given a frame rate of 10 Hz, this is 7*5*116*110*10 = 4 466 000 single-pixel-shift units per second. (As has been noted, the correlation window size, which is 7x7 for both algorithms, does not affect computational complexity.) In our linear-time algorithm, the local neighborhood is +/- 1 pixel for a search neighborhood of 3x3. In addition, there are multiple time delays, which adds another factor. The active image area is (64-2(1+3))*(64-2(1+3)) = 56*56. When calculating 10 Δt per frame the frame rate is 9 frames per second on an 80 MHz HyperSPARC computer. This results in 3*3*10*56*56*9 = 2 540 160 single-pixel-shift units per second. Thus the workstation implementation is only 1.76 times slower than the implementation on the special-purpose image processor.[6] It is expected that

[6] Note that this only considers the raw amount of work done,

by the time of this printing, further increases in CPU performance will have closed this gap [17].

7.1 Parallel Implementations

This type of algorithm is attractive in that there are many places where parallelism can be exploited. The simplicity of the lowest-level calculations makes it suitable for virtually every form of parallelization possible, from multi-processors to massively parallel implementations. Of course, a major strength of the algorithm is that is does not need massive parallelism to run in real-time. Nevertheless, such a system could be fully utilized if available.

Parallelism exploited in software (i.e., explicitly running multiple parts of a program on multiple processors simultaneously) can include running the algorithm on multithreaded machines with a small number of processors, typically two or four. The simplest method is dividing the image into multiple, independent parts. For example, a 3-processor system could divide the image into top, center, and bottom thirds. Each of these separate parts must overlap slightly since the computed optical flow field for each part has an $\eta + \lfloor \nu/2 \rfloor$ border where flow is not calculated. This overlap makes this approach inefficient for implementations with more than about 4 processors. Experiments performed on a 3-processor Themis HyperSPARC 10MP resulted in a 50 % and a 90 % improvement for 2 and 3 processors respectively.

The previous method is applicable to many image processing algorithms. An alternative approach, specific to the optical flow algorithm described in this paper, is to divide the work in time rather than in space. In this case each zero-or-one pixel shift-and-match stage, one for each Δt, could be processed independently. In order to prevent a bottleneck, the winner-take-all stage would have to be merged into the excitation stage as well. Experiments performed on 2 and 3 processors yielded results no better than that of dividing the image into equal parts. However, with a larger number of processors this approach could have the advantage.

At a slightly finer level, the correlation step for each Δt could also be divided into its component multiple directions and computed individually. For the current implementation, this is eight directions of motion, plus zero motion. Future implementations could have greatly increased angular accuracy by interpolating grey level values and considering inter-pixel motions, i.e., motion to an interpolated "virtual" pixel lying in between existing pixels, or by using pixels created by subsampling a shifted input image (in cases where a subsampled image is used).

Hardware implementations could take parallelism to an even finer scale, and includes implementation on special-purpose image processors such as the Datacube [14]. Running the algorithm by processing individual *pixels* on separate processors (as in [8]) is not particularly recommended on typical modern parallel machines which use tens or hundreds of powerful microprocessors, since I/O and communication latencies are then likely to become bottlenecks. Instead, at the finest scale, each row of an image can be processed individually by a pipelined VLSI chip and fed into further processing modules, as has been shown by a partial implementation in CMOS VLSI [38] (see also [39]). For example, up to 256 rows/columns in all 9 directions (8 directions plus zero motion) for each of 10 Δt could in theory be parallelized, exploiting up to 23 040 separate processing units.

and not whether or not the work done is particularly efficient or useful. As noted in Section 3, the linear-time algorithm only grows linearly with the number of velocities detected rather than quadratically as with the traditional approach. In addition, Section 4 argues for the harmonic series of velocity measurements versus the linear set of measurements of the traditional algorithm. These are algorithmic issues however, and orthogonal to this discussion.

Acknowledgments

The "two-pass" implementation of the constant-time box filter was first pointed out to the author by Jim Little. The author is also grateful for helpful discussions with and suggestions from Martin Herman, David Coombs, Hongche Liu, and Tsai-Hong Hong at NIST, and Heinrich Bülthoff and Susanne Huber at the Max-Planck Institute at Tübingen. The translating tree sequence was taken at SRI. The diverging tree sequence was produced by David Fleet at Toronto; the correct flow was copied via ftp from the University of Western Ontario. The remaining images were taken by the author. This research was conducted while the author held a National Research Council Research Associateship at NIST.

References

[1] R. Nelson, J. Aloimonos, "Obstacle Avoidance Using Flow Field Divergence", *IEEE PAMI-11* no. 10, p.1102-1106, October 1987

[2] J. Weber, J. Malik, "Robust Computation of Optical Flow in a Multi-Scale Differential Framework", *Proceedings of the Fourth International Conference on Computer Vision*, p.12-20, 1993

[3] D. Patterson, J. Hennessy, *Computer Organization and Design: the Hardware/Software Interface*, Morgan Kaufmann, San Mateo CA, 1994

[4] B. Horn, P. Schunck, "Determining Optical Flow", *Artificial Intelligence* 17:185-203, August 1981

[5] L. Huang, Y. Aloimonos, "Relative Depth from Motion Using Normal Flow: An Active and Purposeful Solution", *Proceedings of the IEEE Workshop on Visual Motion*, p.196-204, 1991

[6] D. Coombs, M. Herman, T. Hong, M. Nashman, "Real-Time Obstacle Avoidance Using Central Flow Divergence and Peripheral Flow", *Proceedings of the Fifth International Conference on Computer Vision*, Cambridge, MA, June 1995.

[7] H. Nishihara, "Practical Real-Time Imaging Stereo Matcher", *Optical Engineering* 23(5):536-545, Sept./Oct. 1984

[8] H. Bülthoff, J. Little, T. Poggio, "A Parallel Algorithm for Real-time Computation of Optical Flow", *Nature* 337(6207):549-553, 9 Feb 1989

[9] A. Del Bimbo, P. Nesi, "Optical Flow Estimation on the Connection Machine 2", p.267-274, *Proceedings of the IEEE Workshop on Computer Architectures for Machine Perception*, New Orleans Louisiana, IEEE Computer Society Press, Los Alamitos CA, 1993

[10] J. Little, A. Verri, "Analysis of Differential and Matching Methods for Optical Flow", *Proceedings of the IEEE 1989 Workshop on Visual Motion*, 173-180, March 1989

[11] J. Barron, D. Fleet, S.S. Beauchemin, "Performance of Optical Flow Techniques", *International Journal of Computer Vision*, 12(1):43-77, 1994

[12] H. Liu, T. Hong, M. Herman, R. Chellapa, "Accuracy vs. Efficiency Trade-offs in Optical Flow Algorithms", *Proceedings of the Fourth European Conference on Computer Vision*, Cambridge England, April 1996

[13] R. Dutta, C. Weems, "Parallel Dense Depth from Motion on the Image Understanding Architecture", *Proceedings of the IEEE 1993 CVPR*, p.154-159, 1993

[14] J. Little, J. Kahn, "A Smart Buffer for Tracking Using Motion Data", p.257-266, *Proceedings of the IEEE Workshop on Computer Architectures for Machine Perception*, New Orleans Louisiana, IEEE Computer Society Press, Los Alamitos CA, 1993

[15] H. Bülthoff, J. Little, T. Poggio, "A Parallel Motion Algorithm Consistent with Psychophysics and Physiology", *Proceedings of the IEEE 1989 Workshop on Visual Motion*, 165-172, March 1989

[16] T. Camus, H. Bülthoff, "Space-Time Tradeoffs for Adaptive Real-Time Tracking", *Proceedings of the SPIE Mobile Robots VI Conference 1613*, William J. Wolfe, Wendall H. Chun ed., p.268-276, Nov. 1991

[17] T. Camus, *Real-Time Optical Flow*, PhD Thesis, Brown University Technical Report CS-94-36, 1994

[18] T. Camus, "Real-Time Optical Flow", SME Technical Paper MS94-176, MVA/SME Applied Machine Vision June 1994, Minneapolis Minnesota

[19] W. Reichardt, "Autokorrelationsauswertung als Funktionsprinzip des Zentralnervensystems", *Z. Naturforsch.* 12b:447-457, 1957

[20] P. Anandan, "A Computational Framework and an Algorithm for the Measurement of Visual Motion, *International Journal of Computer Vision*, 2:283-310, 1989

[21] V. Nalwa, *A Guided Tour of Computer Vision*, Addison-Wesley, Reading, Massachusetts, 1993

[22] J. Tresilian, "Empirical and Theoretical Issues in the Perception of Time to Contact", *Journal of Experimental Psychology: Human Perception and Performance*, 17:3 p.865-876, 1991

[23] A. Duchon, W. Warren, "Robot Navigation from a Gibsonian Viewpoint", *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, San Antonio, Texas. October 2-5, 1994. IEEE, Piscataway, NJ, pp 2272-2277.

[24] T. Camus, "Calculating Time-to-Contact Using Real-Time Quantized Optical Flow", National Institute of Standards and Technology NISTIR 5609, March 1995

[25] John McCann, *Neural Networks for Mobile Robot Navigation*, Masters Thesis, Brown University, February 1995

[26] A. Duchon, W. Warren, and L. Kaelbling, "Ecological Robotics: Controlling Behavior with Optical Flow", pp. 164-169, *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, Pittsburgh, PA July 22-25, 1995. Johanna D. Moore and Jill Fain Lehman, eds. Lawrence Erlbaum Associates, Mahwah, NJ.

[27] W. Warren Jr., "Action Modes and Laws of Control for the Visual Guidance of Action", in *Complex Movement Behavior: The motor-action controversy*, p.339-380, O. Meijer and K. Roth eds., Elsevier Science, B.V. (North-Holland), 1988

[28] T. Camus, D. Coombs, M Herman, T. Hong, "Real-Time Single-Workstation Obstacle Avoidance Using Only Wide-Field Flow Divergence", *Proceedings of the 13th International Conference on Pattern Recognition*, Vienna, Austria, August 1996.

[29] J. Barron, D. Fleet, S.S. Beauchemin, T. Burkitt, "Performance of Optical Flow Techniques", *Proceedings of the IEEE 1992 CVPR*, p.236-242, 1992.

[30] M. Bober, J. Kittler, "Robust Motion Analysis", *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Seattle, WA, 947-952, 1994.

[31] S. Uras, F. Girosi, A. Verri, V. Torre, "A Computational Approach to Motion Perception", *Biological Cybernetics*, 60:79-87, 1988

[32] H. Liu, *A General Motion Model and Spatio-Temporal Filters for 3-D Motion Interpretations*, Ph.D. Thesis, Technical Report CAR-TR-789 and CS-TR-3525, University of Maryland, 1995.

[33] H. Liu, T. Hong, M. Herman, R. Chellapa, "A General Motion Model and Spatio-Temporal Filters for Computing Optical Flow", to appear in *International Journal of Computer Vision*, 1996

[34] T. Camus, "Real-Time Quantized Optical Flow", *Proceedings of the IEEE Workshop on Computer Architectures for Machine Perception*, Como, Italy, September 18-20, IEEE Computer Society Press, Los Alamitos CA, 1995

[35] D. Fleet, A. Jepson, "Computation of Component Image Velocity from Local Phase Information", *Internation Journal of Computer Vision*, vol.5, no.1, p.77-104, 1990

[36] B. Lucas, T. Kanade, "An Iterative Image Registration Technique with an Application to Stereo Vision", *Proceedings of the DARPA Image Understanding Workshop*, 121-130, 1981

[37] C-C Lin, V. Prasanna, A. Khokhar, "Scalable Parallel Extraction of Linear Features on MP-2", p.352-361, *Proceedings of the IEEE Workshop on Computer Architectures for Machine Perception*, New Orleans Louisiana, IEEE Computer Society Press, Los Alamitos CA, 1993

[38] T. Camus, "Accelerating Optical Flow Computations in VLSI", unpublished report, Brown University January 1991

[39] C. Chakrabarti, J. Jaja, "VLSI Architectures for Template Matching and Block Matching", *Parallel Architectures and Algorithms for Image Understanding*, V.K.P. Kumar ed., p.3-27, Academic Press, 1991