Fast Image Motion Computation on an Embedded Computer

X. Lu and R. Manduchi
University of California, Santa Cruz
{xylu,manduchi}@soe.ucsc.edu

Abstract

Wireless, battery–powered camera networks are becoming of interest for surveillance and monitoring applications. The computational power of these platforms is often limited in order to reduce energy consumption. Among the visual tasks that the onboard processor may be required to perform, motion analysis is one of the most basic and relevant. Knowledge of the direction of motion and velocity of a moving body may be used to take actions such as sending an alarm or triggering other camera nodes in the network.

We present a fast algorithm for identifying moving areas in an image and computing the average velocity in such areas. The algorithm, which was implemented and tested on a Crossbow Stargate embedded platform, is comprised of three stages. First, local differential measurements are used to determine an initial labeling of image blocks. A total least squares approach is proposed, with fast implementation inspired by the work of Benedetti and Perona. Then, belief propagation is used to impose spatial coherence and resolve aperture effect inherent in textureless areas. Finally, the velocity of the resulting blobs is estimated via least squares regression. A detailed analysis of timing and power consumption characteristics of this algorithm is also presented.

1. Introduction

There is a growing interest in wireless networks of cameras for surveillance and monitoring. These are ideal systems for impromptu security installations or when the area to be monitored cannot be wired due to size, cost or other ecological or cosmetic reasons. Designing a wireless network poses a number of engineering challenges. In particular, if the system is battery–operated, power–aware strategies must be implemented to increase the network lifetime.

Energy is consumed in a camera node for three main tasks: sensing (i.e., image acquisition); processing; and communications (via a radio link). A number of trade–offs among these tasks is possible. For example, images may just be transmitted as they are acquired. No energy is spent for image processing, but the energy cost of communication can be substantial. Onboard processing may reduce communication cost by extracting only the relevant information. For example, if nothing has changed or nothing is moving in a monitored scene, there is probably nothing to report. However, image processing may be (and often is) a computationally intense task. Hence, efficient and fast algorithms should be employed in order to minimize the associated energy cost.

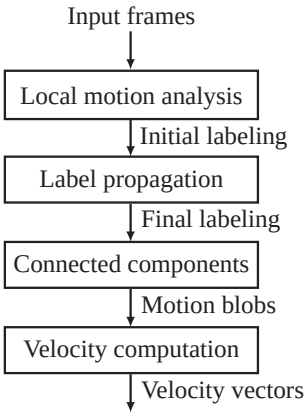


Figure 1. The general scheme of our algorithm.

Scene change detection is often performed using background subtraction. This is a relatively simple approach, which, however, fails in the case of unpredicted illumination changes or accidental camera motion. Another possibility is to detect and compute the image motion between two consecutive frames, taken at a short time interval. This approach requires that two images (rather than one) are acquired during a single duty cycle of the system, and assumes that interesting events are associated with moving bodies in the scene. Unlike background subtraction, however, this procedure is immune from illumination changes and camera motion, as long as these do not occur during the acquisition of a frame pair. In addition, motion information can be used to track a moving object across cameras.

In this paper we present an algorithm for fast motion computation, specifically designed for a processor using

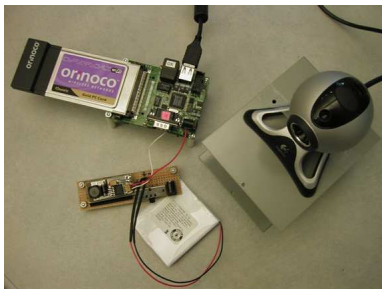


Figure 2. A Meerkats node. Visible are the Crossbow Stargate board with an Orinoco Wi-Fi card and Logitech webcam, the battery (white), and the DC–DC voltage regulator.

fixed point arithmetic. The general scheme of the algorithm is shown in Fig. 1. Image patches are first analyzed independently to derive an initial labeling. More specifically, each image patch is labeled as textured (moving or still), textured only along one direction (with or without a normal motion component), untextured, or outlier. This analysis is based on the computation of the rank of the structure matrix using a fast technique inspired by previous work of Benedetti and Perona [1]. Initial labeling is then propagated using Belief Propagation, generating the final labeling, by which each patch is labeled as moving, not moving, or outlier. Connected components of moving patches are then determined, and an average velocity vector is computed for each one of them. Only this last step requires floating point computation.

We have implemented this algorithm in the nodes of our wireless camera network, dubbed *Meerkats*. Meerkats nodes are based on the Crossbow Stargate board, which has an XScale PXA255 CPU running at 400 MHz, 32MB flash memory and 64MB SDRAM. No floating point unit is available, hence the requirement for fixed point arithmetic processing for efficiency. The Stargate board provides PCMCIA and Compact Flash connectors on the main board. It also has a daughter board with Ethernet, USB and serial connectors. As shown in Fig. 2, we equipped each Stargate with an Orinoco Gold 802.11b PCMCIA wireless card and a Logitech QuickCam Pro 4000 webcam connected through the USB, producing images at 320×240 resolution. We power the board with a customized 7.4 Volt, 1000mAh, 2 cell Li-Ion battery, and an external DC-DC switching regulator to reduce voltage to 5 Volts. The board runs the Linux OS (kernel 2.4.20).

This article is organized as follows. Sec. 2 introduces out local analysis algorithm to obtain the initial labeling. Label propagation is described in Sec. 3, where a number of experimental results are also discussed. A detail analysis of timing and energy consumption associated with the implementation of this algorithm in a Meerkats node is presented in Sec. 4. Sec. 5 has the conclusions.

2. Local Motion Analysis

The general assumption in motion computation is that the brightness I of a moving point in the image is conserved. This is formalized by the well–known optical flow equation:

$$\nabla I^T v + I_t = 0 \tag{1}$$

where $\nabla I = (I_x, I_y)^T$ and $v = (v_x, v_y)^t$. The velocity vector v cannot be computed at a single point due to the rank deficiency of system (1) (the so–called *aperture effect* problem). One possible strategy for conditioning this problem is to assume that all points within a small patch move by the same motion. This hypothesis is at the basis of the least squares approach of Tomasi and Kanade [9]:

$$v^{\text{LS}} = \arg\min_v \sum_p \left(\nabla I^T(p) v + I_t(p)\right)^2 \tag{2}$$

where the sum extends over the N pixels of the considered patch. We will represent (2) in matrix form for simplicity's sake, by stacking the values of $\nabla I^T(p)$ in the columns of the $N \times 2$ matrix A and the values of $I_t(p)$ in the column vector b. The least squares problem (2) can thus be re-written as:

$$v^{\text{LS}} = \arg\min_v \|Av + b\|^2 = \arg\min_v \left(v^T G v + 2v^T c\right) \tag{3}$$

where $G = A^T A$ and $c = A^T b$:

$$G = \begin{bmatrix} \sum_p I_x^2(p) & \sum_p I_x(p) I_y(p) \\ \sum_p I_x(p) I_y(p) & \sum_p I_y^2(p) \end{bmatrix} \tag{4}$$

As pointed out in [9], the rank of G determines whether the patch is well textured ($\text{rank}(G) = 2$) or aperture effects prevails ($\text{rank}(G) < 2$). Patches with $\text{rank}(G) = 1$ are characterized by 1-D texture, and only the component of the motion in the direction of the principal image gradient can be estimated.

Another possibility for velocity estimation in a well–textured patch is to use total least squares regression. In this case, we minimize the following form[1]:

$$r^{\text{TLS}} = \arg\min_r \sum_p \left([\nabla I^T(p) | I_t(p)] r\right)^2 , \ \|r\|^2 = 1 \tag{5}$$

where r is a 3-vector and $[\nabla I^T(p) | I_t(p)]$ is a $N \times 3$ matrix obtained by juxtaposing $\nabla I^T(p)$ and $I_t(p)$. Then, $v_x^{\text{TLS}} = r_1^{\text{TLS}} / r_3^{\text{TLS}}$ and $v_y^{\text{TLS}} = r_2^{\text{TLS}} / r_3^{\text{TLS}}$. Equation (5) can be rewritten in matrix form as:

$$r^{\text{TLS}} = \arg\min_r r^T H r \, , \ \|r\|^2 = 1 \tag{6}$$

[1] Before computing spatial/temporal derivatives, we smooth the images with a simple and fast two–pass separable block filter.

where H is the following symmetric positive semidefinite matrix:

$$H = \left[\begin{array}{cc|c} & & \sum_p I_x(p)I_t(p) \\ & G & \sum_p I_y(p)I_t(p) \\ \hline \sum_p I_x(p)I_t(p) & \sum_p I_x(p)I_t(p) & \sum_p I_t^2(p) \end{array}\right] \quad (7)$$

As well known, total least squares regression has the advantage that errors in the dependent and independent variables (or "data" and "observation" [4]) are given the same importance. Total least squares was first employed for motion flow computation in [11]. As pointed out by Haußecker and Jähne [5], the spectrum of H provides useful insight about the structure of the problem. For the optical flow equation (1) to be satisfied with the same value of v by all points in the patch, the spatio–temporal gradients of I must be contained in a plane containing the origin, and therefore $\text{rank}(H) \leq 2$. Thus, if $\text{rank}(H) = 3$, the patch behavior is not well described by (1). In this situation we will say that the patch is an *outlier*. Haußecker and Jähne maintain that the rank of H is equal to 2 in the case of a moving textured patch, 1 for a patch with 1-D texture (equivalent to the case $\text{rank}(G) = 1$ above), and 0 for a textureless patch. In fact, there are two more situations that were not considered in [5]. First, suppose that $\text{rank}(H) = 2$, which means that the gradients (I_x, I_y, I_t) live in a plane. In this case, (1) is satisfied unless the plane contains the axis I_t, in which case the spatial gradients (I_x, I_y) are confined to a line, corresponding to the case $\text{rank}(G) = 1$ above. The optical flow constraint is not satisfied in this case, and therefore this patch should be labeled as outlier. Likewise, the condition $\text{rank}(H) = 1$, which signifies that the spatio-temporal gradients live in a line, leads to two different possibilities. The optical flow equation is satisfied unless this line coincides with the I_t axis, corresponding to the case of a textureless patch ($\text{rank}(G) = 0$) that changes its brightness due, for example, to noise or illumination variations. Thus, this patch should be labeled as outlier. These additional two conditions must be taken into account for accurate motion analysis.

We formalize the above reasoning in the block scheme of Fig. 3(a), which exploits the observation that $\text{rank}(H)$ is equal to either $\text{rank}(G)$ or to $\text{rank}(G) + 1$. Based on this algorithm, a patch may be labeled into one of the following categories: Textured, satisfying (1); Aperture Effect (1–D, meaning $\text{rank}(G) = 1$); Aperture Effect (Flat, meaning $\text{rank}(G) = 0$); Outlier. The velocity vector for a Textured patch, or its projection onto the dominant spatial gradient for an Aperture Effect (1-D) patch, can be computed by determining the null space of H.

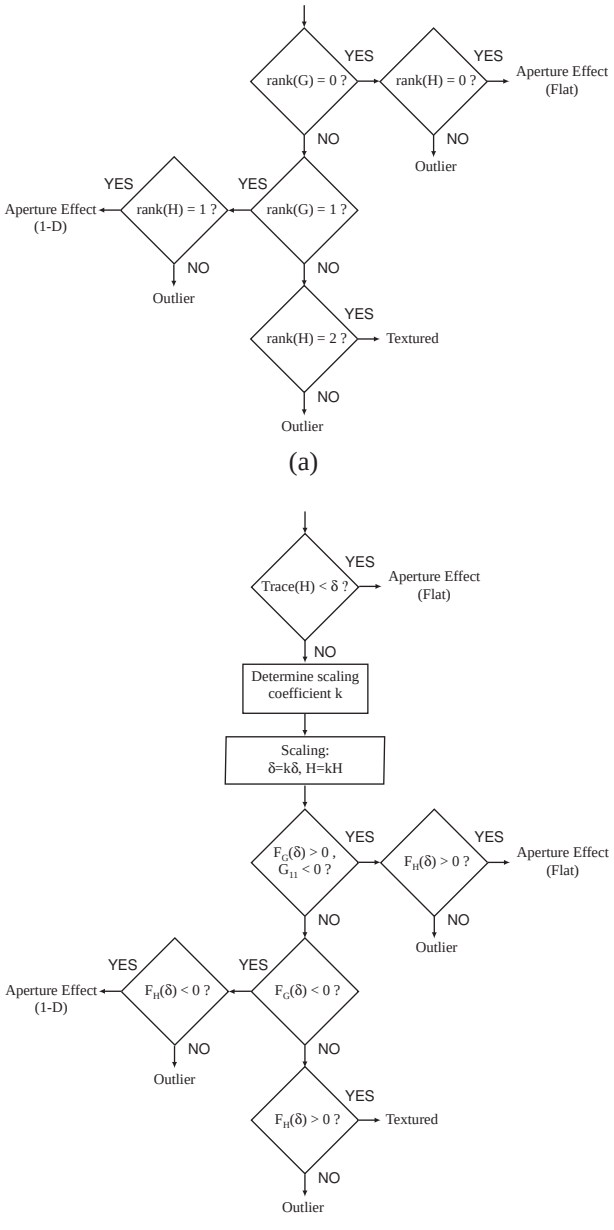


Figure 3. The block scheme of the algorithm to determine the initial labeling of a patch (a) and its implementation (b).

2.1. Fast Computation of Matrix Rank

Estimating the rank of a matrix with noisy data is usually performed using SVD (which in this case is equivalent to eigenvalues analysis). Let $\lambda_1 \leq \lambda_2 \leq \lambda_3$ be the eigenvalues of H. The "numerical rank" of H ($\widetilde{\text{rank}}(H)$) is the number of the eigenvalues that are larger than a threshold, δ [4].

Unfortunately, direct eigenvalue computation requires finding the roots of a polynomial, which is computationally demanding using fixed point arithmetic. Benedetti and Perona [1] proposed an elegant method to estimate the rank

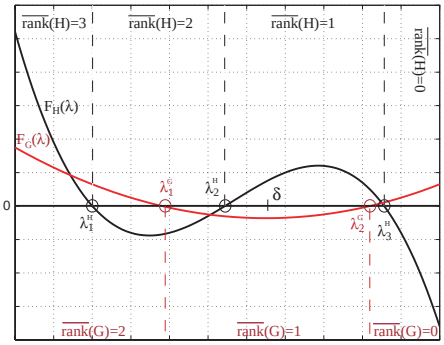


Figure 4. The characteristic polynomials $F_G(\lambda)$ (red) and $\bar{F}_H(\lambda)$ (black). The choice of the threshold δ determines the numerical rank of G and of H.

of G, by comparing its eigenvalues with a threshold without explicitly computing them. We extend this approach to our case in the following.

Consider the characteristic polynomials $F_G(\lambda) = \det(G - \lambda I_2)$ and $F_H(\lambda) = \det(H - \lambda I_3)$, where I_n is the $n \times n$ identity matrix. $F_G(\lambda)$ ($F_H(\lambda)$) is a quadratic (cubic) polynomial, with two (three) positive roots. In addition, $F_G(0) \geq 0$ and $F_H(0) \geq 0$. It can also be shown that the eigenvalues of G and H are interleaved. Fig. 4 shows an example of $F_G(\lambda)$ and $F_H(\lambda)$.

In [1] it is shown that:

- $\overline{\text{rank}}(G) = 0$ iff $F_G(\delta) > 0$ and $G_{11} < 0$
- $\overline{\text{rank}}(G) = 1$ iff $F_G(\delta) < 0$
- $\overline{\text{rank}}(G) = 2$ iff $F_G(\delta) > 0$ and $G_{11} > 0$

Let us recall that $\text{rank}(H)$ is equal to $\text{rank}(G)$ or $\text{rank}(G)+1$. A glance at Fig. 4 will convince the reader that this relationship holds also for numerical ranks. Hence, given $\overline{\text{rank}}(G)$, $\overline{\text{rank}}(H)$ is determined by the sign of $F_H(\delta)$. For example, in the case of Fig. 4, $\overline{\text{rank}}(G) = 1$, and thus $\overline{\text{rank}}(H) = 1$ because $F_H(\delta) > 0$. In our implementation, we chose $\delta = N \cdot 200$, where N is the number of pixels in each patch.

A quick check for patches with $\text{rank}(H) = 0$ can be done by comparing the trace of H with a threshold. The trace of H is equal to the sum of its (positive) eigenvalues, hence, if $\text{trace}(H) < \delta$, we are guaranteed that $\lambda_3 < \delta$. Patches that don't pass this check go through the test procedure described above. This strategy is very efficient in situations with many textureless patches (e.g., when the camera is facing a wall with solid color).

2.2. Avoiding Overflow

Since our implementation uses 32-bit fixed point arithmetic, it is important to correctly dimension and resize the quantities involved in the computation. We start with a 8 bit per pixel grey level and use windows of sizes 11×11. Hence, each entry in H uses up to $\lceil \log_2 121 \rceil + 17 = 24$ bits (no re-scaling is necessary here). We use the "integral image" method [10] to avoid duplicate operations when analyzing overlapping blocks. In order to avoid overflow, integral images are computed over strips with height of 64 pixels. If the strip width is N_w pixels, the accumulated values in the integral image can occupy up to $\lceil \log_2 64N_w \rceil + 17 \leq \lceil \log_2 N_w \rceil + 23$. Hence, as long as N_w is less than or equal to 512, the accumulated values in the strip fit into 32-bit words. We use images of size 240×320, with each strips formed by 64 full image rows. The case of patches straddling across two strips requires special handling which involves a few additional sums.

The entries of H may have to be rescaled before computing $F_G(\delta)$ and $F_H(\delta)$. It is easy to show that, in order to avoid overflow while computing $F_H(\delta)$, it is sufficient that each entries of H as well as δ use no more than 10 bits. This can be achieved by suitable bit-shifting on a patch-by-patch basis. For consistency, we use the same re-scaling for computing $F_G(\delta)$ and $F_H(\delta)$.

2.3. Velocity Thresholding

The next step is to determine whether each patch labeled as Textured or Aperture Effect (1-D) is moving or still. Direct velocity computation using the eigenvectors of H as in [5] requires floating point operations that are computationally intensive. Note, however, that at this stage we are not interested in finding the actual velocity of a patch, but only whether it is moving or not (i.e., whether its velocity is larger than a certain threshold). This can be achieved efficiently based on the least squares approach of (3). The least squares velocity estimate for a Textured block is:

$$\begin{pmatrix} v_x^{\text{LS}} \\ v_y^{\text{LS}} \end{pmatrix} = -G^{-1}c = \tag{8}$$

$$= \frac{-1}{H_{1,1}H_{2,2} - H_{1,2}^2} \cdot \begin{pmatrix} H_{2,2}H_{1,3} - H_{1,2}H_{2,3} \\ H_{1,1}H_{2,3} - H_{1,2}H_{1,3} \end{pmatrix}$$

Given a threshold γ, we can check whether both components of v^{LS} are smaller than it by performing the following tests:

$$-\gamma(H_{1,1}H_{2,2} - H_{1,2}^2) < (H_{2,2}H_{1,3} - H_{1,2}H_{2,3}) \tag{9}$$

$$-\gamma(H_{1,1}H_{2,2} - H_{1,2}^2) < (H_{1,1}H_{2,3} - H_{1,2}H_{1,3})$$

To avoid overflow, the factors in the expressions above should be rescaled by bit-shifting to a maximum size of 16 bits. We set $\gamma = 0.25$ pixels/frame in our experiments.

For what concerns patches marked as Aperture Effect (1-D), it is easy to see that the normal component of the velocity has magnitude equal to:

$$v_{\perp}^{\text{LS}} = \frac{|H_{1,3}|}{\sqrt{H_{1,1}^2 + H_{1,2}^2}} \tag{10}$$

Consequently, in this case we only need to perform the following test:

$$\gamma^2(H_{1,1}^2 + H_{1,2}^2) < H_{1,3}^2 \tag{11}$$

At the end of this local analysis stage, each image patch is labeled into one of six possible categories:

- T0: A textured patch (still);
- T1: A textured patch (moving);
- AE0: A patch with 1-D texture (generating aperture effect) with no normal motion;
- AE1: A patch with 1-D texture and normal motion;
- AEf: A textureless ("flat") patch;
- OL: A patch that does not satisfy the optical flow equation (1) ("outlier").

Note that we do not consider at this point (and in fact, never computed) the actual velocity vector at each patch.

Some examples of initial labeling are shown in Figs. 5-6. Local analysis was performed on 11×11 pixel overlapping patches centered on a subgrid of the original image (one patch every 4×4 pixels, totaling 78×58 patches). Note that most patches in the moving areas were labeled as AE0 or AE1, due to poor image resolution.

In order to appreciate the role of the two conditions on the rank of G mentioned in Sec. 2 and not considered in [5], we present an extreme case with a person abruptly "appearing" in the scene in Fig. 7. Our algorithm correctly interprets the corresponding image area as an outlier. However, if only the rank of H is considered as in Fig. 7(d) (neglecting the rank of G), the result is grossly incorrect, with most patches been labeled as AE1. A similarly incorrect result is obtained using the algorithm of Haußecker and Jähne [5] (Fig. 7(d)). Let us remark that the ability to robustly identify outlier patches is important for motion analysis. There are many situations (occlusions, abrupt change of brightness, moving object that are too close to the camera) in which the optical flow constraint (1) is not satisfied. Failure to detect these situations may lead to gross errors in the estimated motion.

3. Label Propagation

Different labels convey different levels of information and confidence based on local analysis. Patches of type T0 or T1 can be used as "anchor points" for motion information propagation, since they have enough texture and satisfy equation (1) to an acceptable degree. A patch of type AE1 conveys the same motion information as a T1 one (the block was seemingly moving). A patch of type AE0, however, is somewhat different from a T0 one in that there is a (small) chance that the former was actually moving, in a direction orthogonal to the dominant spatial gradient. A patch of type AEf conveys no information about its motion: it could be moving or not, but local analysis is not going to reveal it. Finally, a patch of type OL represents an "anomaly" with respect to our model.

Based on this labeling, we would like to create a new labeling into a reduced set of categories:

- M0: A still patch;
- M1: A moving patch;
- OL: An outlier patch;

We will do this by exploiting spatial coherence under the belief propagation (BP) framework. The basic concept is that a patch whose motion characteristics have been estimated with good confidence (e.g., T0) should propagate motion information to nearby patches with lower confidence (e.g., AEf). This is particularly important to "fill up" textureless areas.

Loopy belief propagation [12] has been used in recent years for stereo [7], image restoration [8], motion computation [3], and shape matching [2]. The goal is to estimate the unobservable labeling of a graph (in this case, the patch grid, with 4–connectivity) from site–based observations, hypothesizing an underlying generative model. The assumption is that labels vary smoothly across sites, except for a limited number of possibly abrupt variations. Borrowing the notation from [3], we define the energy of a given labeling $l(p)$ over the sites (nodes) p as:

$$E = \sum_{(p,q)} V(l(p), l(q)) + \sum_{p} D(l(p)|o(p)) \tag{12}$$

where (p, q) are neighboring sites. The function $V(l(p), l(q))$ is the *discontinuity cost*, that is the cost of assigning different labels to neighboring sites. The term $D(l(p)|o(p))$ (*data cost*) quintifies the cost of assigning label l to site p given that the observation was o. In our case, the observations are the original labels assigned by local analysis.

The BP algorithm seeks to find the labeling that minimizes energy (12) via an iterative mechanism of message exchange. At time t, a site p sends messages to all of its neighbors. Each message is in fact a vector of M values, where M is the number of labels (in our case, $M = 3$). Using the max–product formulation, the i–th message component from node p to node q is computed as follows:

$$m_i^t(p, q) = \min_j \left(V(l_j, l_i) + h_j^{t-1}(p, q) \right) \tag{13}$$

where:

$$h_j^{t-1}(p, q) = D(l_j|o(p)) + \sum_{s \in \mathcal{N}(p) \setminus q} m_j^{t-1}(p, q) \tag{14}$$

$\mathcal{N}(p) \setminus q$ represents the set of neighbors of p other than q. The *belief* at pixel p at time t is equal to :

$$b_p^t(j) = D(l_j|o(p)) + \sum_{s \in \mathcal{N}(p)} m_{s,p}^t(j) \qquad (15)$$

At convergence, the maximizer of $b_p^t(j)$ is the label index assigned to p.

Thus, at each each time t, $4M = 12$ messages must be computed by each site. We use the Potts model for the discontinuity cost, with $V(l_i, l_j) = d > 0$ if $l_i = l_j$, and 0 otherwise (d=3 in our implementation). As noted by Felzenszwalb and Huttenlocher [3], in this case the minimization in (13) can be expressed as:

$$m_{pq}^t(i) = \min\left(h_{pq}^{t-1}(i), \min_j(h_{pq}^{t-1}(j)) + d\right) \qquad (16)$$

This formulation reduces the number of comparisons involved in message formation.

In order to avoid possible overflow due to accumulation of messages in (14), we modify (16) by "normalizing" the messages from one node to another, i.e. by removing a constant so that the smallest message is equal to 0 (and consequently, the largest message is $\leq d$). In other words, the messages actually exchanged are:

$$\bar{m}_{pq}^t(i) = \min\left(h_{pq}^{t-1}(i) - \min_j(h_{pq}^{t-1}(j)), d\right) \qquad (17)$$

It is easy to see that the maximizer of the belief (15) does not change after message normalization. In order to speed up computation, we use the the multiscale implementation of BP proposed in [3]. Two iterations of BP per level are performed starting from the top level of a 5–level pyramid, except for the last level (comprising all image patches), in which case 5 iterations are performed.

In order to assign meaningful data costs, we should recall the meaning of the labels assigned in the previous stage. As a general rule, patches that were not originally labeled as outliers (OL) should have a very high cost of being transformed into outliers. Patches that were labeled as moving with a high degree of confidence (T1 and AE1) should be assigned a relatively high cost of switching to not moving (M0). The final assignment of a textureless patch (AEf) should, in principle, be completely dependent on the messages it receives from its neighbors. However, we noticed empirically that assigning a higher cost to the transition AEf →M1 than to AEf →M0 produces better results, partly because most of the image patches are expected to be still. Finally, an AE0 patch (which has no normal velocity) should be assigned a higher cost to be transformed into a moving (M1) than a still (M0) patch. Tab. 3 translates these considerations into data cost assignments over a small set of parameters ($\text{mid}_1 = 1$; $\text{mid}_2 = 8$; high $= 50$). These values have been assigned empirically, with trial-and-error adjustments, and work reasonably well in our experiments. Some results of final labeling are shown in Figs. 5-6. For each connected component of patches marked as M1, a velocity vector is estimated using the least squares regression of (3). Since only few connected components are usually present, we can afford to use floating point arithmetic for this computation. Although the results are for the most part satisfactory, there is some room for improvement. For example, moving surfaces that are too close to the camera are often labeled as outliers (e.g. Fig. 5(c.1-2)). Another problem is that sometime textureless moving areas are labeled as still, because of insufficient label propagation. This can be improved somewhat by reducing the cost $D(\text{M1}|\text{AEf})$, at the risk, however, of excessive motion propagating. For example, Fig. 7(f) shows the final labeling obtained by setting $D(\text{M1}|\text{AEf}) = 0$. Comparing with Fig. 5(c.2), one notices that a large portion of the still background has been labeled as moving. Finally, when a moving object occludes another object moving in a different direction, it may happen that a single connected components of M1 patches straddles across the two moving areas. In this case, the velocity vector regressed over this connected component is incorrect (see the topmost component in Fig. 6(c.2)).

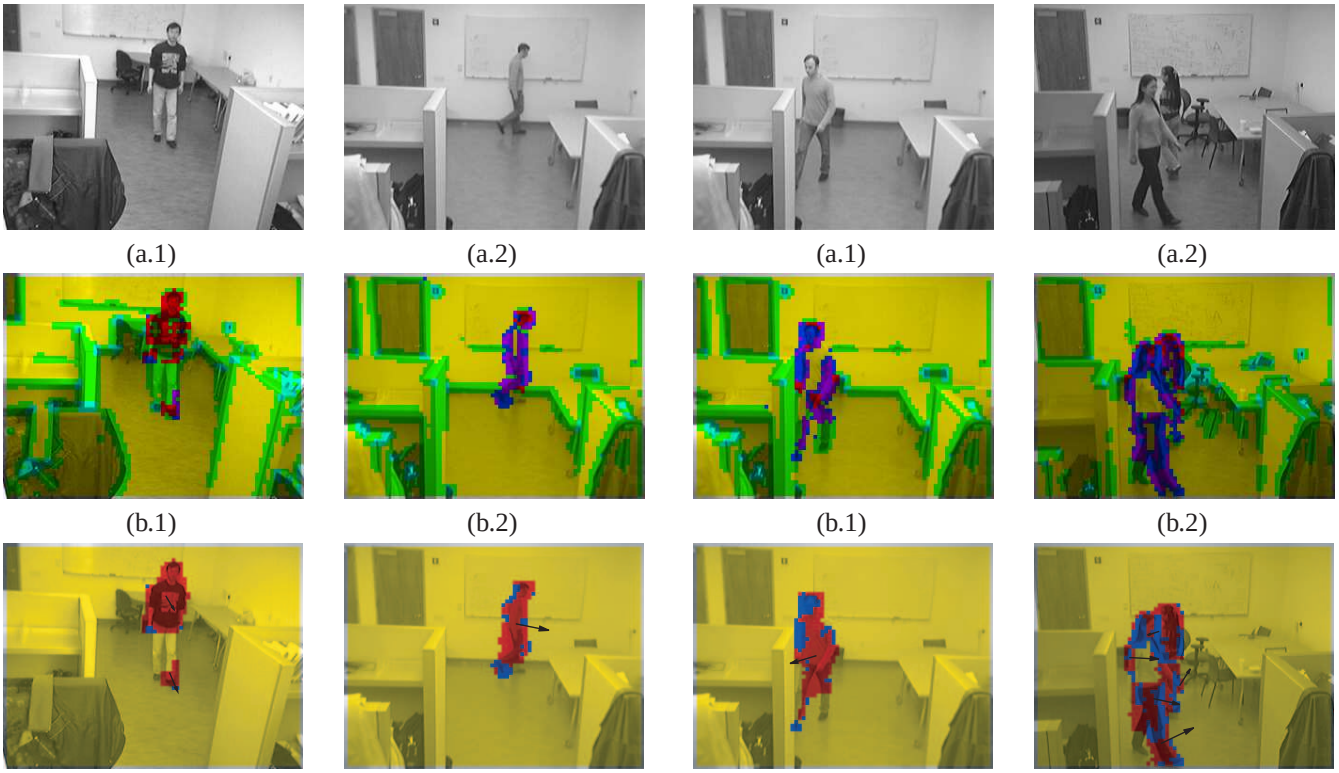
$D(l\|o)$	l=M0	l=M1	l=OL
o=T0	0	mid_2	high
o=T1	mid_2	0	high
o=AE0	0	mid_1	high
o=AE1	mid_2	0	high
o=AEf	0	mid_1	high
o=OL	mid_2	mid_2	0

Table 1. The matrix of data cost values $D(l|o)$. We have used the following parameters in our implementation: $\text{mid}_1 = 1$; $\text{mid}_2 = 8$; high $= 50$. The discontinuity cost d was set to 3.

4. Timing and Energy Considerations

An imaging duty cycle in a typical surveillance application of the Meerkats network includes opening the USB device (camera), taking the two images, closing the device, and processing the images[2]. Using the `time` command in Linux, we determined that the total time elapsed during a duty cycle (invoked as a script) is 1.2 s on average. For comparison, acquiring two images (without processing) requires 520 ms. The difference (680 ms) represents the time required for processing. Further analysis showed that initial labeling based on local analysis takes 70% of the processing time (with BP iterations using the remaining 30%).

[2] A full duty cycle would probably also include loading/removing the appropriate module in/from the kernel, thereby powering on/off the camera. These operations consume a considerable amount of energy [6].



(a.1) (a.2) (b.1) (b.2) (c.1) (c.2)

Figure 5. Motion computation experiments. (a.1),(b.1): First image in the pair. (a.2),(b.2): Initial labeling based on local analysis. Yellow: AEf; Green: AE0; Magenta: AE1; Cyan: T0; Red: T1; Blue: OL. (a.3),(b.3): Final labeling. Yellow: M0; Red: M1; Blue: OL. For each connected component of blocks labeled as M1, the velocity vector (multiplied by a constant factor) is shown with an arrow.

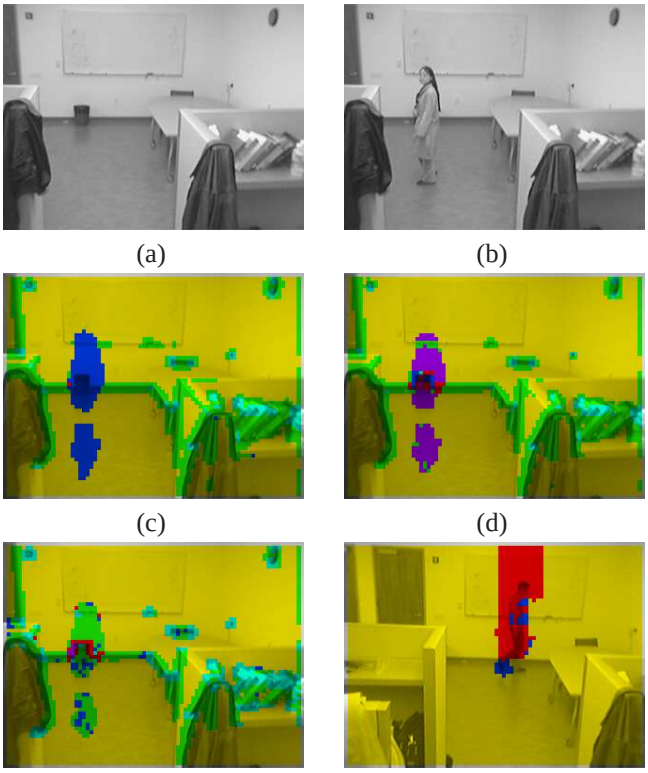
(a.1) (a.2) (b.1) (b.2) (c.1) (c.2)

Figure 6. See caption of Fig. 5.

These numbers alone, however, are not sufficient to characterize the energy cost of an image acquisition/processing duty cycle. Ideally, the CPU would be in idle state until the beginning of the duty cycle, and then switch back to idle as soon as the cycle is over. Note, however, that it is not possible to control this state switch from the application (the control is operated by the OS). In addition, other OS tasks may take place that increase the effective processing time. In order to get a better understanding of the energy issues involved in a duty cycle, we performed direct measurements of the current drawn by the board during as a function of time. For this purpose, we used an HP 34001A digital multimeter providing a reading rate of 60 Hz. During this analysis, the board was powered at 5.4 Volts by an HP E3631A power supply. Fig. 8(a) shows the time profile of current drawn during a duty cycle comprising only acquisition of two images (no processing). Note that the current drawn during idle state (before and after the duty cycle) is about 345 mA. The duration of the "effective" duty cycle (during which time the current drawn is considerably higher than during idle state) is 0.7 s. The incremental charge drawn (i.e. the integral over time of the additional current drawn during the duty cycle with respect to idle state current) is 0.06 C (Coulomb). When motion analysis using our algorithm is performed (Fig. 8(b)), the duration of the effective duty cycle is 1.61 s and the incremental charge is 0.22 C. We thus maintain that our processing has an effective duration of 1.61-0.7=0.91 s, and requires an additional charge of 0.22-0.06=0.16 C (corresponding to an energy of $0.16 \cdot 5.4$=0.86 J). This quantitative analysis is important for the determination of the energy consumption over a certain period of time (and therefore node lifetime for a given battery charge) under fixed duty cycle policy.

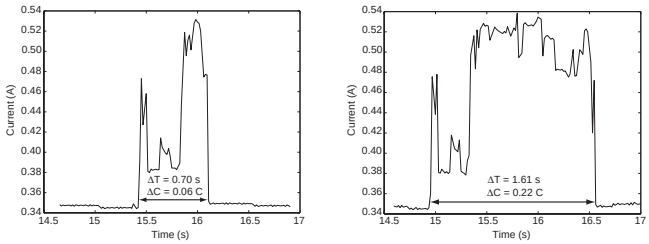
5. Conclusions

We have presented an algorithm for fast velocity computation of moving areas in a scene, and discussed its implementation using fixed point arithmetic in an ARM–based embedded computer. This algorithm proceeds by first labeling individual image patches according to their spatial-temporal structure, and then propagating motion information using Belief Propagation. Floating point operations are only required at the end for estimating the velocity of a very small number of moving image blobs. In addition to algorithm description and experimental tests, we have provided



(a) (b) (c) (d) (e) (f)

Figure 7. (a),(b): Pair of images for motion computation. (c) Initial labeling using our algorithm (see caption of Fig. 5 for color codes). The area where a person "appeared" is correctly labeled as outlier (OL). (d) Initial labeling using only $\overline{\text{rank}}(H)$ (neglecting $\overline{\text{rank}}(G)$). (e) Initial labeling using the method of Haußecker and Jähne [5]. (f) Final labeling for the case of Fig. 5, with $D(\text{M1}|\text{AEf}) = 0$.



(a) (b)

Figure 8. The current drawn during a duty cycle of a Meerkats node. (a): A duty cycle comprising only acquisition of two frames. (b) A duty cycle comprising acquisition of two frames and motion computation. ΔT represents the duration of the interval with higher current drawn than in the idle state. ΔC represents the additional (with respect to the idle state) charge (in Coulomb) drawn during this period.

a detailed timing and energy consumption analysis. This algorithm is currently used in our Meerkats camera network for wide area surveillance.

Two main improvements to the current algorithm are currently being investigated. The first one regards adding the option of multiscale (pyramidal) motion analysis. This would allow for estimation of velocity of bodies at different distances from the camera. Our current system fails when bodies are too close to the camera, due to large image displacement between the two frames (and therefore incorrect time derivative computation). The second improvement regards the segmentation into moving "blobs" for final velocity computation. Fig. 6(c.3) shows the shortcomings of the current method. As pointed out earlier, the topmost blob straddles across two differently moving areas. At the same time, several blobs have been identified for the same moving body, which complicates scene understanding. Further research will be needed to devise a more robust and effective mechanism for splitting and merging moving blobs in a more meaningful way.

References

[1] A. Benedetti and P. Perona. Real-time 2-d feature detection on a reconfigurable computer. In *CVPR98*, pages 586–593, 1998. 2, 3, 4

[2] J. Coughlan and S. Ferreira. Finding deformable shapes using loopy belief propagation. In *ECCV02*, page III: 453 ff., 2002. 5

[3] P. Felzenszwalb and D. Huttenlocher. Efficient belief propagation for early vision. In *CVPR04*, pages I: 261–268, 2004. 5, 6

[4] G. Golub and C. V. Loan. *Matrix Computations*. Johns Hopkins, 1989. 3

[5] H. Haußecker and B. Jähne. A tensor approach for precise computation of dense displacement vector fields. In *DAGM-Symposium*, 1997. 3, 4, 5, 8

[6] C. Margi, V. Petkov, K. Obraczka, and R. Manduchi. Characterizing energy consumption in a visual sensor network testbed. In *2nd Int. IEEE/Create-Net Conf. on Testbeds and Research Infrastructures for the Development of Networks and Communities*, 2006. 6

[7] J. Sun, N. Zheng, and H. Shum. Stereo matching using belief propagation. *PAMI*, 25(7):787–800, July 2003. 5

[8] K. Taaka, J. Inoue, and D. Titteringron. Loopy belief propagation and probabilistic image processing. In *Workshop on Neural Networks for Signal Processing*, 2003. 5

[9] C. Tomasi and T. Kanade. Detection and tracking of point features. Technical Report CMU-CS-91-132, Carnegie Mellon University, 1991. 2

[10] P. Viola and M. Jones. Robust real-time face detection. In *ICCV01*, page II: 747, 2001. 4

[11] J. Weber and J. Malik. Robust computation of optical-flow in a multiscale differential framework. *IJCV*, 14(1):67–81, January 1995. 3

[12] J. Yedidia, W. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. Technical Report TR-2001-22, MERL, 2002. 5