MASTER THESIS

Efficient Implementation and Evaluation of Methods for the Estimation of Motion in Image Sequences

UiS Universitetet i Stavanger, Norway
HSR Hochschule für Technik Rapperswil, Switzerland

Robert Hegner	robert.hegner@hsr.ch

Advisors:

Assoc. Prof. Dr. Ivar Austvoll	ivar.austvoll@uis.no
Assoc. Prof. Dr. Tom Ryen	tom.ryen@uis.no
Prof. Dr. Guido Schuster	guido.schuster@hsr.ch

Stavanger, June 13, 2010

This report was typeset using LaTeX. The figures were created with Matlab or MS Visio.

Abstract

Introduction

Optical flow estimation (the estimation of the apparent motion of objects in an image sequence) is used in many applications like video compression, object detection and tracking, robot navigation, and so on.

This project was focussed on one specific optical flow estimation algorithm, which uses directional filters and an AM-FM demodulation algorithm for the estimation of the velocities.

Goals

The main goals of this project were

1. implementing the algorithm in CUDA (parallel computing architecture developed by NVIDIA) to make use of the huge parallel computing power of modern GPUs (graphic processing units).
2. extending the algorithm to a multiresolution scheme to allow the estimation of higher speeds (Pyramid Algorithm).
3. integrating the algorithms into an (existing) Matlab GUI which allows to compare the new algorithm with other optical flow estimation algorithms.

Results

The speedup of the CUDA implementation (running on a GeForce GTX 260 with 216 parallel cores) compared to an existing Matlab implementation (running on an Intel Core 2 Quad 2.4GHz) is several thousand (depending on the dimensions of the image sequence).

For most of the image sequences used for evaluation, the accuracy of the Pyramid Algorithm is better than or comparable to the accuracy of the OpenCV implementation of the famous Lucas-Kanade algorithm.

Due to the large spatial support of the directional filters, the algorithm has some problems handling motion discontinuities, particularly in the border regions of an image sequence.

The estimation of high speeds was achieved by computing a rough estimate of the (high) speeds in a downsampled image sequence. The motion in the full-resolution image sequence can then be compensated before estimating the speeds on the full-resolution image. This new estimates of the compensated speeds can finally be used to refine the first rough estimates. This procedure can be applied recursively over several levels (Pyramid Algorithm).

One problem of the pyramid implementation of the algorithm is that unreliable estimates from downsampled levels can have a negative impact on the estimations on the full-resolution level.

Outlook

For both the Basic Algorithm and the Pyramid Algorithm it could be worthwhile to have a closer look at the boundary problem. For the Pyramid Algorithm, a postprocessing step for the rough estimates should be considered. There is also some potential to reduce the execution time of the CUDA Pyramid Algorithm.

Acknowledgments

First of all I want to thank my advisors from the University of Stavanger (UiS), Assoc. Prof. Dr. Ivar Austvoll and Assoc. Prof. Dr. Tom Ryen for introducing me to another interesting topic in signal processing and for supervising and supporting me during this project. Thanks also to Tuan Williams for providing all the hard- and software I needed and for the support with the IT infrastructure.

Many thanks also to the people who made this semester abroad possible. First of all to Assoc. Prof. Dr. Tom Ryen (UiS) and Prof. Dr. Guido Schuster (HSR), but also to Bente Dale (UiS) who handled the administrative issues.

Finally I want to thank all my friends from the White-Boxes for the great time we spent together in Norway. I hope we will meet again sometime.

Contents

List of Tables

List of Figures

1 Original Project Description

1. Project title: **Efficient Implementation and Evaluation of methods for estimation of motion (optical flow) in image sequences (video)**.
2. Subject: *Computer Vision, Image and Video Analysis*
3. Problem: See 1.2
4. Implementation: See 1.3
5. Advisors: See 1.4

1.1 Introduction

Motion is one of the basic phenomena in vision. Humans and animals use motion to avoid dangers, find food etc. Motion is therefore important also in many artificial vision systems. Some applications are surveillance, object tracking and reconstruction, robot navigation and video compression (coding), e.g. in MPEG 1-2 and 4. The apparent motion of objects in an image is called optical flow. There is usually a difference between this apparent motion and the real motion which is the projection of the object's 3D motion on the 2D image plane. The optical flow (OF) is represented by a field of motion vectors, one for each pixel in the image. The OF, as defined by the Brightness Constraint Equation, is an ill-posed problem. A simple model of the imaging system is a pinhole camera. If we assume that the illumination is given by a point source at infinity and a single moving object is considered, and that the reflectance of the object is Lambertian and that there is no photometric distortion, then the difference between the real velocity in the image plane and the OF can be shown to be small when the spatial gradient is large, i.e. when there is an edge or structure in the image.

1.2 Problem

Given an image sequence (video), how can we describe the motion of objects and background? Many methods have been suggested for estimation of the velocity field (optical flow) [13, 15, 14, 7, 9, 10, 8]. From a practical point of view there are two main problems connected to the choice between methods, accuracy of the flow vectors and computation time. The most accurate methods are in general more complex and therefore more computational intensive. For real time applications this is a serious drawback. In this project we want to evaluate a few chosen methods with respect to accuracy, density of flow vectors for a given certainty measure and the efficiency of the implementation. We are particularly interested in a comparison between differential methods and phase-based methods using directional filters. We also want to study the importance and necessity of a multiresolution scheme (image pyramid, scale space), and how the estimated velocity vectors can be propagated from one level to the next in order to improve the final result.

1.3 Implementation

As a reference we want to use the Lucas & Kanade method. Implementation of this method is available in the Open Computer Vision (OpenCV) library [25, 31, 32]. The OpenCV integrates into applications written in C or C++. A real time implementation of this algorithm is discussed in [16]. The work will be done with matlab as interface. In the bachelor work of Fabian Braun and Marc Länzlinger 2009 a GUI is developed in Matlab and the optical flow algorithm runs on the graphic card using CUDA. In this master project we want to use this framework and extend their work. Our algorithm based on phase and the use of directional filters [7, 8] should be implemented. Different solutions for efficient computation should be considered. An evaluation methodology for optical flow is presented in [11] where also a data base of test sequences is presented. We want to evaluate our method compared with other methods with respect to accuracy and computation time (complexity). Methods that are of interest to compare is a method based on warping of Brox et al. [14], the PDE-based method [15] of Bruhn et al., and a method using point trajectories of Sand and Teller [17].

1.4 Advisors

HSR: Prof. Guido Schuster

at UiS Assoc. prof. Ivar Austvoll

at UiS Assoc. prof. Tom Ryen

2 Introduction

2.1 Optical Flow Estimation

Introduction The estimation of motion in an image sequence (video) is needed in many applications like object detection and tracking, video compression, or robot navigation.

Real Motion The projection of the motion of objects in a 3D scenery to the 2D image plane is called *real motion*. It is usually not possible to estimate this real motion, since the *apparent motion* differs from the real motion. This can have several reasons:

- Objects can be partially hidden by other objects or suddenly (dis)appear (*occlusion problem*).
- The *aperture problem* is illustrated in figure 2.1. It shows an image sequence containing a striped square, moving along direction v_1, v_2 or v_3. The goal is to estimate the motion for one specific point (■) of the image sequence. When looking at the local neighbourhood (circle) of this point, the apparent motion is always in the direction of v_2 (orthogonal to the structure of the square), no matter in which of the three directions the square is moving.

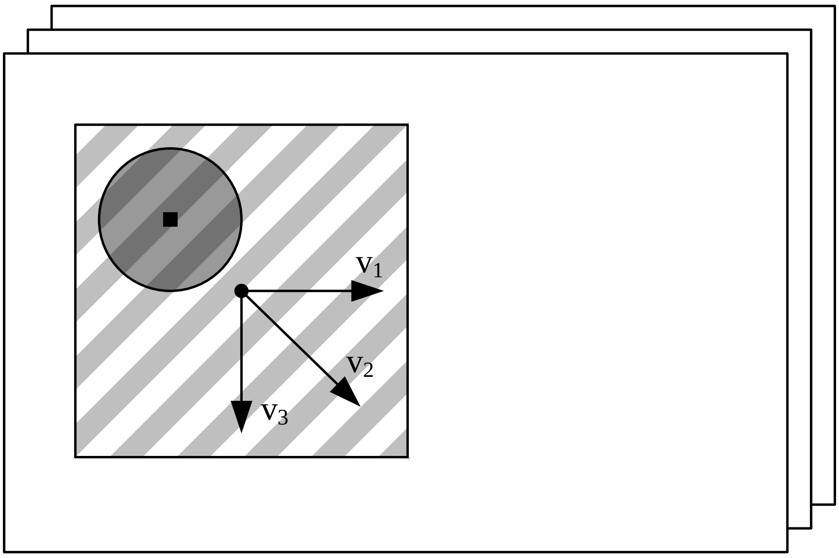


Figure 2.1 Illustration of the aperture problem.

- If the square object in figure 2.1 had no texture at all, no motion could be detected (unless the border is inside the current neighbourhood).
- On the other hand, in a scenery with no motion of the objects, a moving camera or moving light sources can suggest motion of the objects.

Optical Flow The optical flow of an image sequence is a vector field describing the velocity (speed and direction) of the apparent motion of every point in the image (dense motion field).

Algorithms In the past decades, many optical flow estimation algorithms have been developed. The ones proposed by Horn & Schunck and Lucas & Kanade are amog the most famous ones. A good overview and comparison of different optical flow estimation algorithms can be found in [11, 12, 18].

Implemented Algorithm This project is focussed on one specific optical flow estimation algorithm developed by Ivar Austvoll and Espen Kristoffersen [6].

2.2 Tasks of this Project

Main Tasks The practical part of this project consisted of the following tasks:

- An (existing) Matlab implementation of the algorithm described in this report was integrated into the Matlab GUI, which was developed during the previous project.
- The algorithm was efficiently implemented in C for running on the graphic card (GPU) using CUDA (see section 2.3) and integrated into the Matlab GUI.
- The algorithm was improved by applying a pyramid scheme. This allows to estimate higher speeds in the image sequence.
- The speedup of the CUDA implementations comparing to the Matlab implementation were determined.
- The accuracy of the implementated algorithms was compared for different parameters and compared with well known optical flow estimation algorithms.

Further Tasks Furthermore, the existing C and Matlab code from the previous project was modified:

- The C code (CUDA implementations and OpenCV wrappers) had to be modified in order to support also Win64 systems (besides Win32).
- The functionality of the Matlab GUI was extended by adding new filter types and new post-processing options.

2.3 CUDA Basics

Introduction

CUDA is a parallel computing architecture developed by NVIDIA which allows to utilize the computation power of modern NVIDIA GPUs (graphic processing units). Modern GPUs with dozens or even hundreds of parallel processing units are perfectly suited for many computational intensive applications.

CUDA basically provides some extensions to the standard C language which allow kernel code (code that runs on a GPU) to be written in C. Additionally, an API to manage devices, threads, memory, etc. is provided.

This section describes some basic CUDA concepts which are needed to understand the description of the implementation in chapter 5. Further information can be found in the CUDA documentation [21, 22, 23].

2.3.1 Thread Hierarchy

Threads

CUDA kernels are executed in threads. A common scheme in image processing applications is to implement a kernel in such a way that it processes just one pixel. Therefore, $M \cdot N$ threads are executed for an image with dimensions $(M \times N)$.

Blocks and Grid

Threads are grouped into thread blocks. Thread blocks can be one-, two- or three-dimensional, depending on the application. In image processing, two-dimensional blocks are the obvious choice. Choosing a block size of say (16×16), leads to $(\lceil M/16 \rceil \times \lceil N/16 \rceil)$ blocks. These blocks constitue a grid (figure 2.2).

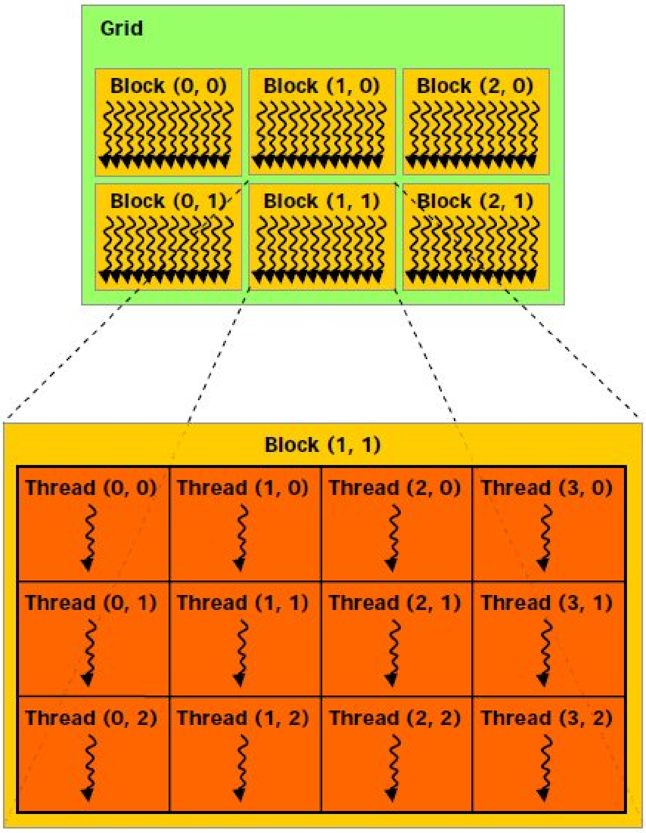


Figure 2.2 Grid of thread blocks [21].

Note that the grid dimension is determined by the size of the data being processed and is not limited by hardware ressources. This is due to the highly scalable architecture of the CUDA GPUs (see below).

CUDA provides the two built-in variables `threadIdx` and `blockIdx` which contain the thread index within a block and the block index within the grid. Two additional built-in variables (`blockDim` and `gridDim`) can be used to find out the block and grid dimensions of the current configuration.

In a two-dimensional setup, the coordinate of the pixel being computed by the current thread can be computed as shown in listing 2.1.

Listing 2.1 Coordinates of the current thread in a two-dimensional setup.

```
unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;
unsigned int idy = blockIdx.y*blockDim.y + threadIdx.y;
```

Multiprocessors

Current GPUs have up to 30 streaming multiprocessors (SMs) which consist of eight scalar processor (SP) cores, two special function units for transcendentals, a multithreaded instruction unit, and on-chip shared memory [21]. A multiprocessor executes one or more thread blocks at a time. The GPU is free to allocate thread blocks to multiprocessors in any order. Therefore, dependencies between thread blocks must be avoided.

This architecture has the advantage that the execution can be scaled automatically depending on the number of available multiprocessors (figure 2.3).

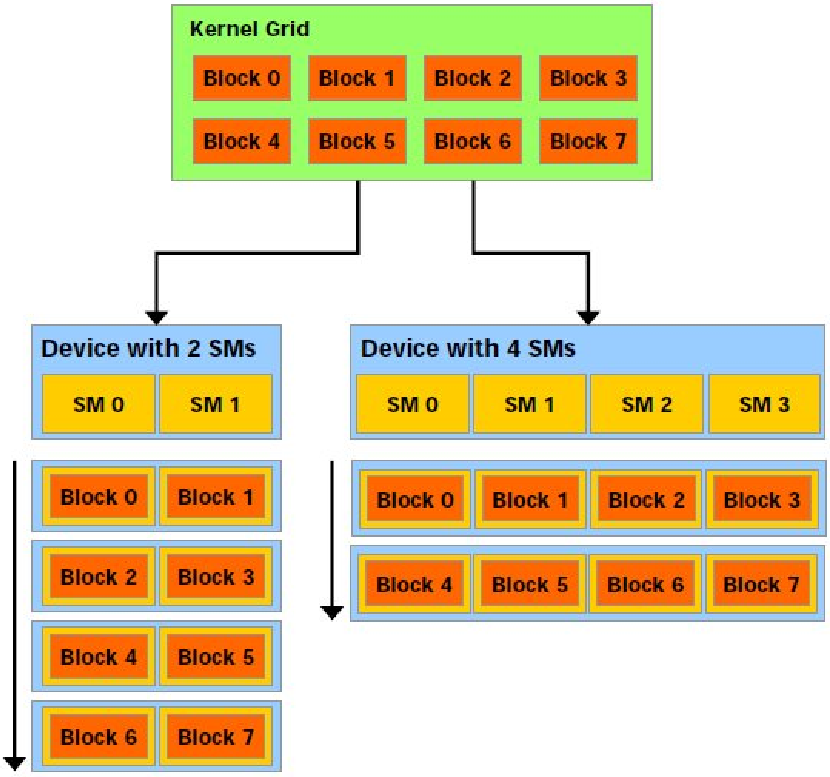


Figure 2.3 A device with more SMs will automatically execute a kernel grid in less time than a device with fewer SMs [21].

2.3.2 Memory Organization

Host Memory and Device Memory

In CUDA terminology, the main memory of the computer (RAM) is called host memory whereas memory on the GPU is called device memory. Transfers between host and device memory are very slow (compared to transfers on the GPU) and should therefore be avoided. A good strategy is to copy the unprocessed data to the device memory in one big transaction, keep all intermediate results in the device memory, and copy the results back to host memory in the end.

There are different types of device memory:

Global Memory

Most of the GPU memory is global memory which can be accessed by all threads (and the host). Access to global GPU memory is generally slow (compared to shared memory or registers). However, under certain conditions, several memory accesses are performed in one transaction (coalescing), which makes it much more efficient. The typical situation where accesses can be coalesced is when all threads in a half-warp are accessing subsequent words (4, 8 or 16 bytes) in global memory at the same time. In this situation, only one transaction is needed [21].

For devices with CUDA compute capability 1.2 or higher, the conditions for coalesced memory access are less strict than for devices with compute capability 1.0 and 1.1. Details can be found in [21, 22].

Local Memory

Every thread has its own local memory. It is as slow as global memory, because it is not physically located on the chip. Local memory is not explicitly used by the programmer. The compiler places automatic variables into local memory when there are not enough registers available [22].

Registers

Access to registers is very fast. However, the number of registers is limited and the scope of a register is only one thread.

Shared Memory

Shared memory is much faster than global memory (about 100× lower latency) and can be as fast as accessing registers [21, 22]. All threads in a thread block have access to the same shared memory.

Shared memory is divided into 16 banks. A bank conflict occurs when more than one thread of a half-warp wants to access memory in the same bank at the same time. In this case, the memory accesses are serialized. The Parallel Nsight Analysis Tools can be used to detect bank conflicts (by monitoring the *Warp Serialization* counter) and therefore help optimizing the memory access.

Constant Memory

Constant memory can be read from all threads but can only be written by the host. Since the constant memory is cached, access can be very fast.

Texture Memory

Texture memory is also a read-only memory which can be accessed by all threads. To use this kind of memory, a properly aligned global memory area is bound to a texture. Data can then be read by texture fetches. Texture memory is cached (optimized for spatial locality in two dimensions).

Textures also offer some interesting addressing features:

- A texture can be addressed by floating point values and return interpolated values (nearest neighbour or linear interpolation).
- Boundary cases (out of range addessing) can be handled automatically (clamping or wraping).
- A normalized addressing mode is available which allows to access a texture with addresses in the range $[0, 1]$, independent of the actual texture dimensions.

- When a texture is bound to a memory area storing integer values, these values can automatically be converted to floating point values in the range $[0,1]$ or $[-1,1]$.

3 Basic Algorithm

Introduction This chapter describes the Basic Algorithm (no multiresolution scheme and no scale-space filtering) proposed in [6] from an implementation point of view. Section 3.1 explains the basic idea behind this algorithm by summarizing the relevant information from [4, 6, 7].

The remaining sections contain supplementary implementation related information that is not covered by [4, 6, 7].

3.1 Description of the Algorithm

Overview The basic idea of this algorithm is to decompose the image sequence using a set of directional filters. The output of a directional filter reveals the structure of an image for a given direction.

For each direction, the magnitude (and the sign) of the velocity is computed. These directional velocities are called *component velocities*. With two or more component velocities (of different directions), the x- and y-components of the resulting optical flow vectors can be computed using a linear system of equations.

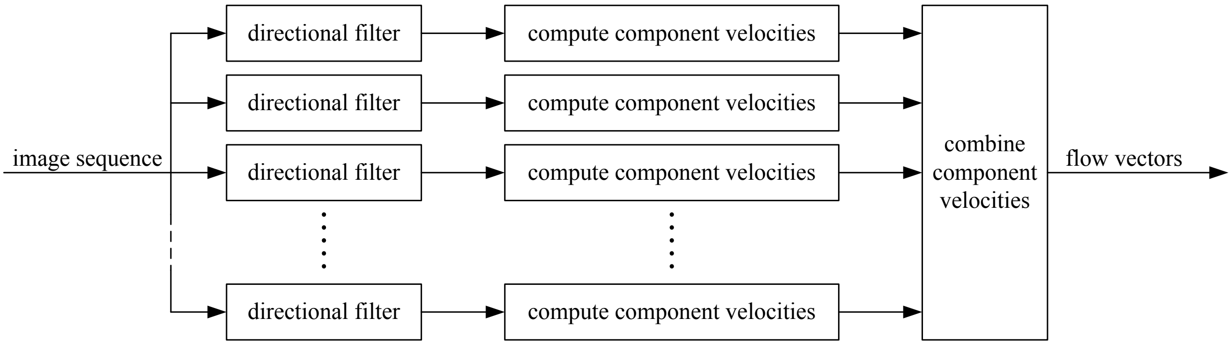


image sequence
directional filter
directional filter
directional filter
directional filter
compute component velocities
compute component velocities
compute component velocities
compute component velocities
combine component velocities
flow vectors

Figure 3.1 Overview of the Basic Algorithm.

ST-Slices When looking at the image sequence as a three-dimensional volume, the component velocity for a given direction can be determined by slicing the cuboid in the time direction along the respective direction and examining the structure of this so called ST-slice.

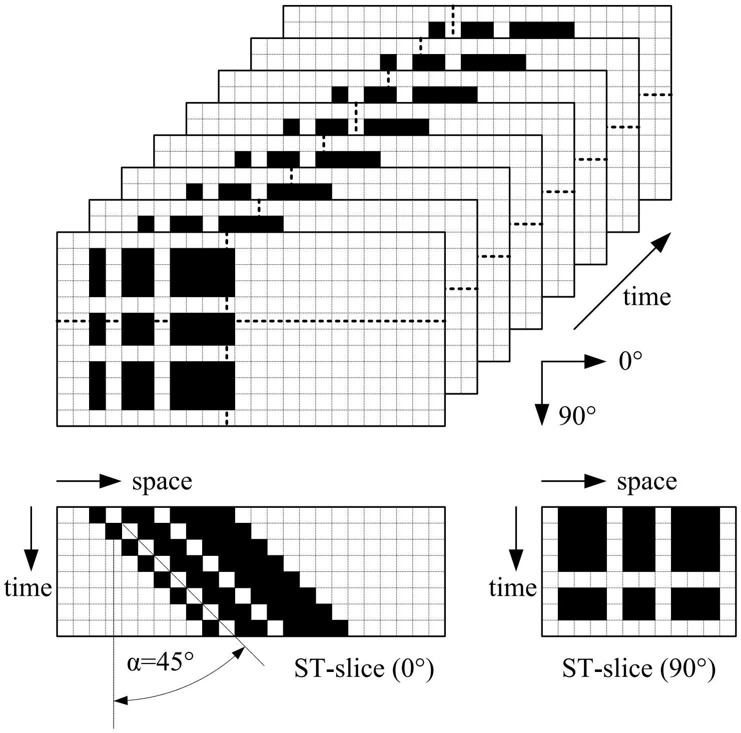


Figure 3.2 Illustration of ST-slices.

Figure 3.2 shows two ST-slices for the directions 0° and 90°. The image sequence contains a simple translatory motion with $1\, pixel/frame$ along the 0° direction. The two slices were taken along the bold dashed lines in the image sequence. The space coordinate of an ST-slice always corresponds to the direction of the directional filter. It can easily be seen that the speed in a given direction is directly related to the angle α at which features move along the time axis of the ST-slice. In the first case (0° ST-slice), the angle is 45°, which corresponds to a speed of $\tan(45°) = 1\, pixel/frame$. In the second case (90° ST-slice), the angle is 0° and therefore the speed in this direction is 0.

Figure 3.2 is somewhat simplyfied; the output of the directional filter and therefore the content of the ST-slices is complex valued and the phase information is used for the component velocity computation. More specifically, the translation of the phase fronts along the time axis of the ST-slice corresponds to the component velocity.

Directional Filters

The directional filters used in this project are two-dimensional complex filters. The filters are separable into a longitudinal (along the filter direction) and a transversal (orthogonal to the filter direction) part.

The transversal part is a real lowpass filter with a narrow bandwidth to ensure that most of the energy is in the direction of the filter.

The longitudinal part is a complex bandpass filter with a wide bandwidth to capture as much energy for the given direction as possible. The longitudinal part also acts as a Hilbert transform. This means, that the spectrum of the output is zero for negative frequencies. This type of signal is called an *analytic signal* [30].

Signal Model

The fact that the output of the directional filter is approximately an analytic signal, lets us model a ST-slice in the form:

$$z(\boldsymbol{s}) = a(\boldsymbol{s}) \cdot e^{j\phi(\boldsymbol{s})} \tag{3.1}$$

where $\boldsymbol{s}$ is a vector consisting of the coordinate s along the filter direction and the time coordinate t.

The signal representation in (3.1) consists of an AM-function $a(\boldsymbol{s})$ and a phase function $\phi(\boldsymbol{s})$.

As mentioned above, the velocities are obtained by examining the movement of the phase fronts in the ST-slices. Therefore, the essential information is contained in the phase gradient $\nabla\phi(\boldsymbol{s})$, which is also called the *instantaneous frequency* (FM-function).

A good explanation of the Hilbert transform and it's relation to analytic signals and the instantaneous frequency can be found in [5].

Demodulation

The algorithm proposed in [6] uses a discrete 2D AM-FM demodulation algorithm to estimate the instantaneous frequency directly from the real- and imaginary-part of the directional filter output. The demodulation is not described here. The formulas for the computation are reproduced in section 3.3.

In the following, the two components of the estimated instantaneous frequency, $\nabla_s\hat{\phi}(\boldsymbol{s})$ (along the filter direction) and $\nabla_t\hat{\phi}(\boldsymbol{s})$ (along the time coordinate), are treated separately.

Structural Tensor

To compute the direction of the movement of the structure in an ST-slice (angle α in figure 3.2) from the instantaneous frequency, a tensor formulation is used. The tensor is in this case simply a symmetric 2×2 matrix which represents the structure within the ST-slice [1, 2].

The component velocity can then be found by looking at the eigenvalues and eigenvectors of this matrix. The eigenvector $\boldsymbol{e}_1$ associated with the smallest eigenvalue points in the direction of the movement. Therefore, the component velocity is:

$$v_c = \tan(\alpha) = \frac{\boldsymbol{e}_1(1)}{\boldsymbol{e}_1(2)} \tag{3.2}$$

To get a more robust estimation of the component velocity, the matrix is averaged over a local ST-slice (rectangular region around the current pixel). The equations for building the tensor and for computing the component velocity are reproduced in section 3.4.

Flow Vectors

The equations to calculate a flow vector out of the component velocities using a weighted least squares approach are reproduced in section 3.5.

3.2 Directional Filtering

Introduction

Instead of using a set of directional filters with different orientations, rotated versions of the image sequence are filtered by the same directional filter with orientation 0°. [6]

3.2.1 Rotating

Introduction

When rotating an image counterclockwise by an angle ϑ, the new coordinates (x', y') of a point (x, y) can be computed using the following transformation [29]:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\vartheta & -\sin\vartheta \\ \sin\vartheta & \cos\vartheta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{3.3}$$

Coordinate System

Note that equation (3.3) is a counterclockwise rotation in a Cartesian coordinate system (positive x-axis to the right, positive y-axis to the top). In image processing, where the y-axis usually points downwards, it corresponds to a clockwise rotation. However, when passing data from Matlab to a *.mex* file, the x- and y-axes are interchanged, which results in another change of the rotation direction (see also section 5.3). The equations in this section, particularly the sign of the rotation angle, are therefore in accordance with the actual CUDA implementation.

New Image Size

To determine the size of the rotated image, the new coordinates $P'_1 = (x'_1, y'_1)$, $P'_2 = (x'_2, y'_2)$, $P'_3 = (x'_3, y'_3)$ and $P'_4 = (x'_4, y'_4)$ of the corners $P_1 = (0, 0)$, $P_2 = (w-1, 0)$, $P_3 = (0, h-1)$ and $P_4 = (w-1, h-1)$ are computed (where w and h are the dimensions of the original image).

Then the new image dimensions w' and h' are:

$$min_x = \min\{x'_1, x'_2, x'_3, x'_4\} \tag{3.4}$$
$$max_x = \max\{x'_1, x'_2, x'_3, x'_4\} \tag{3.5}$$
$$min_y = \min\{y'_1, y'_2, y'_3, y'_4\} \tag{3.6}$$
$$max_y = \max\{y'_1, y'_2, y'_3, y'_4\} \tag{3.7}$$
$$w' = \lceil max_x - min_x + 1 \rceil \tag{3.8}$$
$$h' = \lceil max_y - min_y + 1 \rceil \tag{3.9}$$

Rotating

The rotation process itself goes the other way round: for every destination pixel (x', y'), the corresponding pixel in the source image (x, y) is found. This implies to use the inverse of the transformation in (3.3) which is easy to find since the rotation matrix is orthogonal:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos\vartheta & \sin\vartheta \\ -\sin\vartheta & \cos\vartheta \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} \tag{3.10}$$

The transformation in (3.10) rotates around the point $(0, 0)$ and leads therefore to negative indexes for the destination image. To prevent this, an offset of min_x (3.4) and min_y (3.6) is added to the destination coordinates:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos\vartheta & \sin\vartheta \\ -\sin\vartheta & \cos\vartheta \end{bmatrix} \begin{bmatrix} x' + min_x \\ y' + min_y \end{bmatrix} \tag{3.11}$$

Padding

When the padding of the image should be done implicitly during rotation (see section 5.4.3), the transformation looks like this:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos\vartheta & \sin\vartheta \\ -\sin\vartheta & \cos\vartheta \end{bmatrix} \begin{bmatrix} x' + min_x \\ y' + min_y \end{bmatrix} - \begin{bmatrix} pads \\ pads \end{bmatrix} \tag{3.12}$$

In this case the dimension of the source image is assumed to be extended by $2 \cdot pads$ in both directions.

Inverse Rotation

For the most general case (3.12), the inverse transformation (for rotating the image back to its original position) can be found as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\vartheta & -\sin\vartheta \\ \sin\vartheta & \cos\vartheta \end{bmatrix} \begin{bmatrix} x + pads \\ y + pads \end{bmatrix} - \begin{bmatrix} min_x \\ min_y \end{bmatrix} \tag{3.13}$$

Interpolation

In general, the transformed coordinates in the rotating or back-rotating process are not integers and therefore some interpolation is needed. The simplest case is the nearest neighbour interpolation, where the transformed coordinates are rounded to the nearest integer. Better results can be achieved when linear or higher order interpolation is used.

When implementing the rotation on the GPU with texture memory, linear interpolation can be performed in hardware (see section 5.4.3).

3.2.2 Filtering

Introduction

The complex 2D directional filters introduced in [6] are separable. This means that the convolutions along the two dimensions (longitudinal and transversal direction) can be computed consecutively, which is more efficient than applying one single 2D convolution [20].

Filters

The longitudinal filter is a complex filter of length 15 and the transversal filter is a real filter of length 55. Therefore it is advisable to first apply the transversal filter to the rotated image and then compute the real part of the filter response using the real part of the longitudinal filter and the imaginary part of the filter response using the imaginary part of the longitudinal filter.

Coordinate System

Figure 3.3 shows the new coordinate system which is used for rotated values in the following sections. The s-axis always points towards the filter direction.

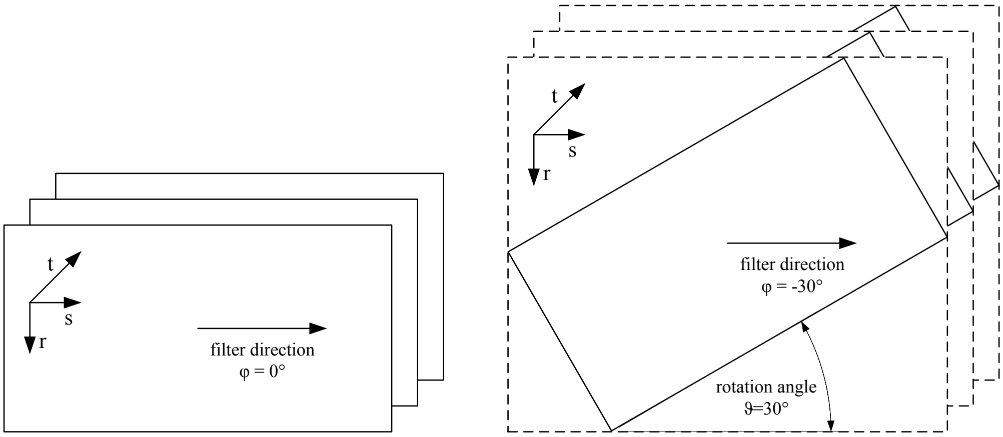


Figure 3.3 Coordinate system for the output of the directional filters ($\varphi = 0°$ and $\varphi = -30°$).

3.3 Computation of Instantaneous Frequencies

Introduction

In [6], the following equations for estimating the instantaneous frequency components $\nabla_s\hat{\phi}(r,s,t)$ and $\nabla_t\hat{\phi}(r,s,t)$ are derived:

$$\left|\nabla_s\hat{\phi}(r,s,t)\right| = \arccos \operatorname{Re}\left\{\frac{z(r,s+1,t)+z(r,s-1,t)}{2\cdot z(r,s,t)}\right\} \tag{3.14}$$

$$\left|\nabla_t\hat{\phi}(r,s,t)\right| = \arccos \operatorname{Re}\left\{\frac{z(r,s,t+1)+z(r,s,t-1)}{2\cdot z(r,s,t)}\right\} \tag{3.15}$$

$$\operatorname{sgn}\nabla_s\hat{\phi}(r,s,t) = \operatorname{sgn}\arcsin \operatorname{Re}\left\{\frac{z(r,s+1,t)-z(r,s-1,t)}{2j\cdot z(r,s,t)}\right\} \tag{3.16}$$

$$\operatorname{sgn}\nabla_t\hat{\phi}(r,s,t) = \operatorname{sgn}\arcsin \operatorname{Re}\left\{\frac{z(r,s,t+1)-z(r,s,t-1)}{2j\cdot z(r,s,t)}\right\} \tag{3.17}$$

$$\nabla_s\hat{\phi}(r,s,t) = \operatorname{sgn}\nabla_s\hat{\phi}(r,s,t)\cdot\left|\nabla_s\hat{\phi}(r,s,t)\right| \tag{3.18}$$

$$\nabla_t\hat{\phi}(r,s,t) = \operatorname{sgn}\nabla_t\hat{\phi}(r,s,t)\cdot\left|\nabla_t\hat{\phi}(r,s,t)\right| \tag{3.19}$$

Where $z(r,s,t)$ is the complex valued filter response of the directional filters.

Analytical Solution for Re **Operator**

The equations (3.14) to (3.17) include expressions of the form

$$\frac{c_1+c_2}{c_3} \quad \text{and} \quad \frac{c_1-c_2}{jc_3} \tag{3.20}$$

where $c_i = a_i + jb_i$ are complex numbers. To get the real part of these expressions in the CUDA implementation, they can be rewritten:

$$\begin{aligned}\frac{c_1+c_2}{c_3} &= \frac{(a_1+jb_1)+(a_2+jb_2)}{a_3+jb_3} = \frac{(a_1+a_2)+j(b_1+b_2)}{a_3+jb_3}\\ &= \frac{(a_1+a_2)+j(b_1+b_2)}{a_3+jb_3}\cdot\frac{a_3-jb_3}{a_3-jb_3}\\ &= \underbrace{\frac{a_3(a_1+a_2)+b_3(b_1+b_2)}{a_3^2+b_3^2}}_{\operatorname{Re}\{\cdot\}} + j\underbrace{\frac{a_3(b_1+b_2)-b_3(a_1+a_2)}{a_3^2+b_3^2}}_{\operatorname{Im}\{\cdot\}}\end{aligned} \tag{3.21}$$

and

$$\frac{c_1-c_2}{jc_3} = \ldots = \underbrace{\frac{a_3(b_1-b_2)-b_3(a_1-a_2)}{a_3^2+b_3^2}}_{\operatorname{Re}\{\cdot\}} + j\underbrace{\frac{-a_3(a_1-a_2)-b_3(b_1-b_2)}{a_3^2+b_3^2}}_{\operatorname{Im}\{\cdot\}} \tag{3.22}$$

Applying (3.21) and (3.22) to equations (3.14) to (3.17):

$$\left|\nabla_s\hat{\phi}(r,s,t)\right| = \arccos\frac{\operatorname{Re}\{z(r,s,t)\}\Big(\operatorname{Re}\{z(r,s+1,t)\}+\operatorname{Re}\{z(r,s-1,t)\}\Big)+\operatorname{Im}\{z(r,s,t)\}\Big(\operatorname{Im}\{z(r,s+1,t)\}+\operatorname{Im}\{z(r,s-1,t)\}\Big)}{2\cdot\Big(\operatorname{Re}\{z(r,s,t)\}^2+\operatorname{Im}\{z(r,s,t)\}^2\Big)} \tag{3.23}$$

$$\left|\nabla_t\hat{\phi}(r,s,t)\right| = \arccos\frac{\operatorname{Re}\{z(r,s,t)\}\Big(\operatorname{Re}\{z(r,s,t+1)\}+\operatorname{Re}\{z(r,s,t-1)\}\Big)+\operatorname{Im}\{z(r,s,t)\}\Big(\operatorname{Im}\{z(r,s,t+1)\}+\operatorname{Im}\{z(r,s,t-1)\}\Big)}{2\cdot\Big(\operatorname{Re}\{z(r,s,t)\}^2+\operatorname{Im}\{z(r,s,t)\}^2\Big)} \tag{3.24}$$

$$\operatorname{sgn}\nabla_s\hat{\phi}(r,s,t) = \operatorname{sgn}\arcsin\frac{\operatorname{Re}\{z(r,s,t)\}\Big(\operatorname{Im}\{z(r,s+1,t)\}-\operatorname{Im}\{z(r,s-1,t)\}\Big)-\operatorname{Im}\{z(r,s,t)\}\Big(\operatorname{Re}\{z(r,s+1,t)\}-\operatorname{Re}\{z(r,s-1,t)\}\Big)}{2\cdot\Big(\operatorname{Re}\{z(r,s,t)\}^2+\operatorname{Im}\{z(r,s,t)\}^2\Big)} \tag{3.25}$$

$$\operatorname{sgn}\nabla_t\hat{\phi}(r,s,t) = \operatorname{sgn}\arcsin\frac{\operatorname{Re}\{z(r,s,t)\}\Big(\operatorname{Im}\{z(r,s,t+1)\}-\operatorname{Im}\{z(r,s,t-1)\}\Big)-\operatorname{Im}\{z(r,s,t)\}\Big(\operatorname{Re}\{z(r,s,t+1)\}-\operatorname{Re}\{z(r,s,t-1)\}\Big)}{2\cdot\Big(\operatorname{Re}\{z(r,s,t)\}^2+\operatorname{Im}\{z(r,s,t)\}^2\Big)} \tag{3.26}$$

Sign Terms

The expressions for the signs can be simplyfied, since $\operatorname{sgn}\arcsin(x) = \operatorname{sgn}(x)$ and since the denominator of (3.25) and (3.26) is always positive:

$$\operatorname{sgn}\nabla_s\hat{\phi}(r,s,t) = \operatorname{sgn}\Big(\operatorname{Re}\{z(r,s,t)\}\Big(\operatorname{Im}\{z(r,s+1,t)\} - \operatorname{Im}\{z(r,s-1,t)\}\Big) - \operatorname{Im}\{z(r,s,t)\}\Big(\operatorname{Re}\{z(r,s+1,t)\} - \operatorname{Re}\{z(r,s-1,t)\}\Big)\Big) \tag{3.27}$$

$$\operatorname{sgn}\nabla_t\hat{\phi}(r,s,t) = \operatorname{sgn}\Big(\operatorname{Re}\{z(r,s,t)\}\Big(\operatorname{Im}\{z(r,s,t+1)\} - \operatorname{Im}\{z(r,s,t-1)\}\Big) - \operatorname{Im}\{z(r,s,t)\}\Big(\operatorname{Re}\{z(r,s,t+1)\} - \operatorname{Re}\{z(r,s,t-1)\}\Big)\Big) \tag{3.28}$$

Confidence Measure

One assumption in the derivation of this algorithm is that the amplitude of the AM-FM modelled signal is varying sufficiently slow. This assumption can be checked by looking at the gradient of the amplitude part. From this, a confidence measure $\nabla\hat{\phi}_c(r,s,t)$ can be constructed which is close to one when the gradient is small and close to zero when the amplitude part is varying heavily [7]:

$$\nabla_s\hat{\phi}_c(r,s,t) = \arcsin\operatorname{Im}\left\{\frac{z(r,s+1,t) + z(r,s-1,t)}{2\cdot z(r,s,t)}\right\} \tag{3.29}$$

$$\nabla_t\hat{\phi}_c(r,s,t) = \arcsin\operatorname{Im}\left\{\frac{z(r,s,t+1) + z(r,s,t-1)}{2\cdot z(r,s,t)}\right\} \tag{3.30}$$

$$\nabla\hat{\phi}_c(r,s,t) = \sqrt{\left(1 - \frac{2}{\pi}\cdot\left|\nabla_s\hat{\phi}_c(r,s,t)\right|\right)\cdot\left(1 - \frac{2}{\pi}\cdot\left|\nabla_t\hat{\phi}_c(r,s,t)\right|\right)} \tag{3.31}$$

The first two expressions can again be rewritten using (3.21):

$$\nabla_s\hat{\phi}_c(r,s,t) = \arcsin\frac{\operatorname{Re}\{z(r,s,t)\}\Big(\operatorname{Im}\{z(r,s+1,t)\} + \operatorname{Im}\{z(r,s-1,t)\}\Big) - \operatorname{Im}\{z(r,s,t)\}\Big(\operatorname{Re}\{z(r,s+1,t)\} + \operatorname{Re}\{z(r,s-1,t)\}\Big)}{2\cdot\Big(\operatorname{Re}\{z(r,s,t)\}^2 + \operatorname{Re}\{z(r,s,t)\}^2\Big)} \tag{3.32}$$

$$\nabla_t\hat{\phi}_c(r,s,t) = \arcsin\frac{\operatorname{Re}\{z(r,s,t)\}\Big(\operatorname{Im}\{z(r,s,t+1)\} + \operatorname{Im}\{z(r,s,t-1)\}\Big) - \operatorname{Im}\{z(r,s,t)\}\Big(\operatorname{Re}\{z(r,s,t+1)\} + \operatorname{Re}\{z(r,s,t-1)\}\Big)}{2\cdot\Big(\operatorname{Re}\{z(r,s,t)\}^2 + \operatorname{Re}\{z(r,s,t)\}^2\Big)} \tag{3.33}$$

3.4 Computation of Component Velocities

Introduction

To compute the component velocities $v_c(r,s,t)$ from the instantaneous frequencies $\nabla_s\hat{\phi}(r,s,t)$ and $\nabla_t\hat{\phi}(r,s,t)$, an eigenvector analysis of the following 2×2 matrix is proposed in [6]:

$$\boldsymbol{T}(r,s,t) = \begin{bmatrix} T_{ss}(r,s,t) & T_{st}(r,s,t) \\ T_{st}(r,s,t) & T_{tt}(r,s,t) \end{bmatrix} \tag{3.34}$$

$$T_{ss}(r,s,t) = \left\langle \nabla_s\hat{\phi}_i \cdot \nabla_s\hat{\phi}_i \right\rangle_{i\in\Omega} \tag{3.35}$$

$$T_{st}(r,s,t) = \left\langle \nabla_s\hat{\phi}_i \cdot \nabla_t\hat{\phi}_i \right\rangle_{i\in\Omega} \tag{3.36}$$

$$T_{tt}(r,s,t) = \left\langle \nabla_t\hat{\phi}_i \cdot \nabla_t\hat{\phi}_i \right\rangle_{i\in\Omega} \tag{3.37}$$

Where $\langle\cdot\rangle$ is the mean operator and $\nabla_s\hat{\phi}_i$ and $\nabla_t\hat{\phi}_i$ are instantaneous frequencies in the spatial (along the filter direction s) and temporal neighbourhood. However, only neighbours whose confidence values $\nabla\hat{\phi}_c(r,s',t')$ are above a certain threshold, are considered.

From (3.34), the component velocity $v_c(r,s,t)$ and a confidence measure $c_v(r,s,t)$ can be computed as follows [6]:

$$v_c(r,s,t) = \frac{\boldsymbol{e}_1(1)}{\boldsymbol{e}_1(2)} \tag{3.38}$$

$$c_v(r,s,t) = \frac{\lambda_2 - \lambda_1}{\lambda_2 + \lambda_1} \tag{3.39}$$

λ_1 and λ_2 are the eigenvalues of $\boldsymbol{T}(r,s,t)$ (where $0 \leq \lambda_1 \leq \lambda_2$) and $\boldsymbol{e}_1$ is the eigenvector that corresponds to λ_1.

Matlab Implementation In the existing Matlab implementation of the algorithm, the velocity and the confidence measure are computed as follows:

Listing 3.1 Computation of the component velocity and the confidence measure using Matlab.

```
1  T = [t1'*t1  t1'*t2; t2'*t1  t2'*t2];
2  T = T/norm(T,'fro');
3  [ev,lamb] = eig(T);
4  vel = ev(1,1)/ev(2,1);
5  conf = (lamb(2,2)-lamb(1,1))/(lamb(2,2)+lamb(1,1));
```

However, for the CUDA implementation a more explicit expression is needed.

CUDA Implementation For a 2 × 2 matrix

$$\boldsymbol{T} = \begin{bmatrix} T_{ss} & T_{st} \\ T_{st} & T_{tt} \end{bmatrix} \tag{3.40}$$

an analytic solution for the eigenvalues and eigenvectors can easily be found:

$$\lambda_{1,2} = \frac{T_{ss} + T_{tt} \mp \sqrt{T_{ss}^2 + 4 \cdot T_{st}^2 - 2 \cdot T_{ss} \cdot T_{tt} + T_{tt}^2}}{2} \tag{3.41}$$

$$\boldsymbol{e}_{1,2} = \begin{bmatrix} \dfrac{T_{ss} - T_{tt} \mp \sqrt{T_{ss}^2 + 4 \cdot T_{st}^2 - 2 \cdot T_{ss} \cdot T_{tt} + T_{tt}^2}}{2 \cdot T_{st}} \\ 1 \end{bmatrix} \tag{3.42}$$

Equations (3.38) and (3.39) can now be rewritten:

$$v_c(r,s,t) = \frac{T_{ss}(r,s,t) - T_{tt}(r,s,t) - \sqrt{T_{ss}^2(r,s,t) + 4 \cdot T_{st}^2(r,s,t) - 2 \cdot T_{ss}(r,s,t) \cdot T_{tt}(r,s,t) + T_{tt}^2(r,s,t)}}{2 \cdot T_{st}(r,s,t)} \tag{3.43}$$

$$c_v(r,s,t) = \frac{\sqrt{T_{ss}^2(r,s,t) + 4 \cdot T_{st}^2(r,s,t) - 2 \cdot T_{ss}(r,s,t) \cdot T_{tt}(r,s,t) + T_{tt}^2(r,s,t)}}{T_{ss}(r,s,t) + T_{tt}(r,s,t)} \tag{3.44}$$

Normalization Multiplying the matrix $\boldsymbol{T}$ by a constant factor (see listing 3.1, line 2) has no influence on the eigenvectors. The eigenvalues are scaled by the same factor, but this factor would be cancelled in (3.39). Therefore, no normalization of the matrix is done in the CUDA implementation.

Confidence Measure

From $c_v(r,s,t)$ and the confidence values $\nabla\hat{\phi}_c(r,s,t)$ from section 3.3, a final confidence measure $c(r,s,t)$ for the component velocity $v_c(r,s,t)$ can be computed:

$$c(r,s,t) = \sqrt{c_v(r,s,t) \cdot \sqrt[N]{\prod_{i\in\Omega} \nabla\hat{\phi}_{c\,i}(r,s',t')}} \tag{3.45}$$

$\nabla\hat{\phi}_{c\,i}(r,s',t')$ are the confidence measures in the spatial (along the filter direction s) and temporal neighbourhood which are above a certain threshold (the same as for $\nabla_s\hat{\phi}_i$ and $\nabla_t\hat{\phi}_i$). N is the number of valid $\nabla\hat{\phi}_{c\,i}(r,s',t')$ values.

3.5 Combining the Component Velocities

Introduction

The last step in computing the flow vector field is to combine the component velocities $v_{ck}(r,s,t)$ to the final velocity vector with the components $v_x(x,y,t)$ and $v_y(x,y,t)$. The subscript k is the index of the direction (which was omitted in the sections 3.2 to 3.4 for the sake of clarity).

The component velocities $v_{ck}(r,s,t)$ and the confident values $c_k(r,s,t)$ must be rotated to the 0°direction first (see also section 3.2.1). These back-rotated values are denoted with $v_{ck}(x,y,t)$ and $c_k(x,y,t)$. The x, y und t indices are omitted in the following.

The following (in general overdetermined) system of equations is used to compute $v_x(x,y,t)$ and $v_y(x,y,t)$: [6]

$$\begin{bmatrix} \cos(\varphi_0) & \sin(\varphi_0) \\ \cos(\varphi_1) & \sin(\varphi_1) \\ \vdots & \vdots \\ \cos(\varphi_{N-1}) & \sin(\varphi_{N-1}) \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} v_{c0} \\ v_{c1} \\ \vdots \\ v_{c(N-1)} \end{bmatrix} - \varepsilon \tag{3.46}$$

The rows of the matrix on the left-hand side are unit vectors pointing to the same direction as the according directional filter (note that $\varphi_i = -\vartheta_i$, see figure 3.3). Only component velocities whose certiainty measure is above a certain threshold are used. N is the number of valid component velocities for a certain pixel.

This overdetermined system of equations is solved using a weighted least squares approach where the confidence measures c_k are used as weights.

Solution

The equations in (3.46) can be written as:

$$\cos(\varphi_k) \cdot v_x + \sin(\varphi_k) \cdot v_y = v_{ck} - \varepsilon_k \tag{3.47}$$

The error measure Q is the sum of the weighted squared errors ε_k:

$$Q = \sum_k c_k \cdot \varepsilon_k^2 = \sum_k c_k \cdot \left(v_{ck} - \cos(\varphi_k) \cdot v_x - \sin(\varphi_k) \cdot v_y\right)^2 \tag{3.48}$$

To minimize Q, the partial derivatives with respect to v_x and v_y are taken:

$$\frac{\partial Q}{\partial v_x} = -2 \sum_k c_k \cdot \left(v_{ck} - \cos(\varphi_k) \cdot v_x - \sin(\varphi_k) \cdot v_y\right) \cdot \cos(\varphi_k) \tag{3.49}$$

$$\frac{\partial Q}{\partial v_y} = -2 \sum_k c_k \cdot \left(v_{ck} - \cos(\varphi_k) \cdot v_x - \sin(\varphi_k) \cdot v_y\right) \cdot \sin(\varphi_k) \tag{3.50}$$

Setting the derivatives to zero and reordering gives the following system of equation to compute v_x and v_y:

$$\begin{bmatrix} \sum_k c_k \cdot \cos(\varphi_k)^2 & \sum_k c_k \cdot \cos(\varphi_k) \cdot \sin(\varphi_k) \\ \sum_k c_k \cdot \cos(\varphi_k) \cdot \sin(\varphi_k) & \sum_k c_k \cdot \sin(\varphi_k)^2 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} \sum_k c_k \cdot v_{ck} \cdot \cos(\varphi_k) \\ \sum_k c_k \cdot v_{ck} \cdot \sin(\varphi_k) \end{bmatrix} \tag{3.51}$$

Confidence Measure The confidence value for the resulting velocity vector is the arithmetic mean of the valid confidence values of the component velocities.

Special Cases When the confidence value of only one direction k is above the threshold, the x and y components of this component velocity are returned:

$$v_x = \cos(\varphi_k) \cdot v_{ck} \tag{3.52}$$

$$v_y = \sin(\varphi_k) \cdot v_{ck} \tag{3.53}$$

When no valid component velocity is available, v_x and v_y as well as the confidence value for this velocity vector is set to zero.

4 Pyramid Algorithm

4.1 Introduction

Maximum Speed

The range of speeds that can be estimated using the method described in chapter 3 is limited due to aliasing. The maximum speed that can be detected reliably depends on the spatial frequencies occuring in the image sequence. This relation is illustrated in figure 4.1.

Figure 4.1 shows the ST-slice of an image sequence with a uniform and constant translatory motion along the filter direction. In this case, the trajectories of the features in the image sequence are straight lines. The angle α of these trajectories (71.6°) is directly related with the speed of the motion in the image sequence:

$$v = \tan(71.6^\circ) = 3\,pixels/frame \tag{4.1}$$

The spatial frequency in this first example of figure 4.1 is relatively small (with a "wavelength" of $\lambda = 7\,pixels$), so no aliasing occurs in this case.

In the second example of figure 4.1, the image sequence has higher spatial frequencies ($\lambda = 2\,pixels$). Due to aliasing, the (wrong) speed belonging to the dashed trajectories is estimated. Another case where the speed is too high for a given spatial frequency is shown in the third example. Instead of $v = 5\,pixels/frame$ (solid trajectories), a speed of $v = -2\,pixels/frame$ (dashed trajectories) is estimated.

A more formal discussion of the relationship between the spatial frequency and the maximum speed that can be estimated can be found in [4].

Smoothing Filter

In order to avoid aliasing, the image sequence is pre-smoothed with a spatial filter (see also section 4.2, scale-space filters) to suppress high spatial frequencies. In this project, filters with a relative cutoff frequency of 0.5 are used, which allows to estimate speeds of up to $2\,pixels/frame$.

High Speeds

To estimate higher speeds, one could use a smoothing filter with a lower cutoff frequency. But this would reduce the accuarcy of the estimates since more details of the images get lost. The second problem with estimating high speeds is that the accuracy gets worse for high speeds (when the angles of the trajectories in figure 4.1 approach 90°), because of the non-linear relation between the angle α and the speed [4].

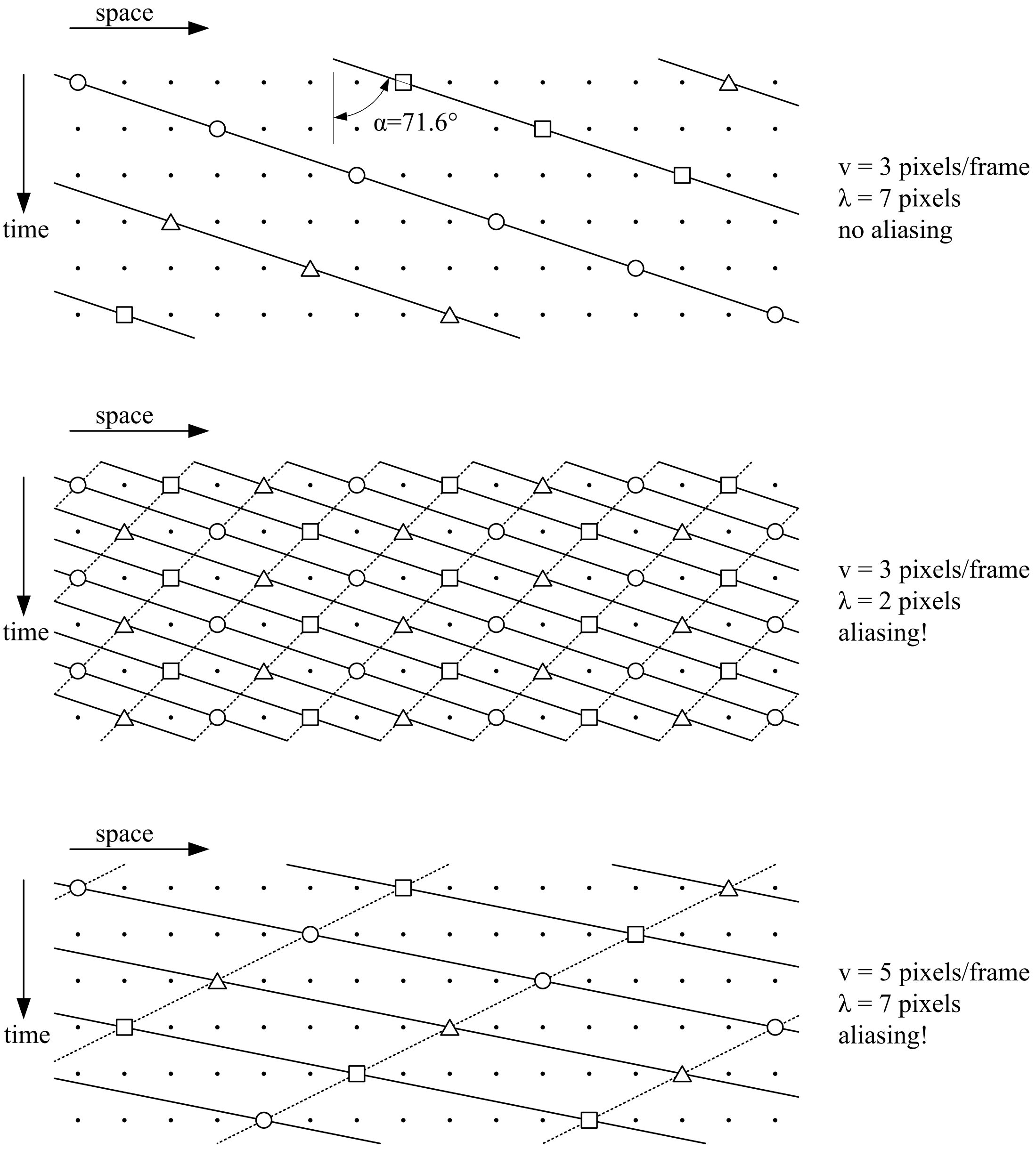


Figure 4.1 Illustration of the aliasing problem in a ST-slice. The circles, squares and triangles represent features of the image moving along the direction of the ST-slice.

4.2 Multiresolution

Introduction

A better approach to estimate high speeds is to use a multiresolution scheme where a downsampled image sequence is used to compute a coarse estimate of the speeds and the full resolution image sequence is used to refine this estimation. A speed of 1 $pixel/frame$ in the downsampled image sequence corresponds to a speed of K $pixels/frame$ in the original image sequence, when the image sequence was downsampled by factor K in both spatial dimensions.

This idea can be applied repeatedly, resulting in a pyramid scheme. The downsampling factor from one level to the next is always 2 in this project (see figure 4.2). Note that there is no downsampling in the temporal dimension (no change in the number of frames).

When going up to level M in the pyramid (and assuming that speeds of up to 2 $pixels/frame$ can be estimated on each frame), the maximum speed that can be handled by the pyramid scheme is 2^{M+1} $pixels/frame$.

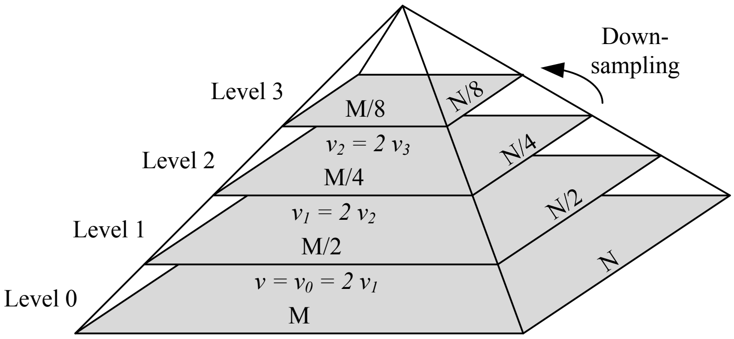


Figure 4.2 Pyramid with 4 levels.

Scale-Space Filter

Figure 4.3 shows the recursive smoothing and downsampling of the original image to get the higher level representations. The scale-space filter H_{sc} used for smoothing is a filter out of the class found by Pauwels et al. [19]. In their paper they show that the set of all possible linear and rotation-invariant filters can be reduced considerably when two additional conditions (recursivity and scale invariance) are demanded. With these conditions, a family of filters is found which is characterized by only one parameter. The two-dimensional Gaussian filter is one special case of this filter class.

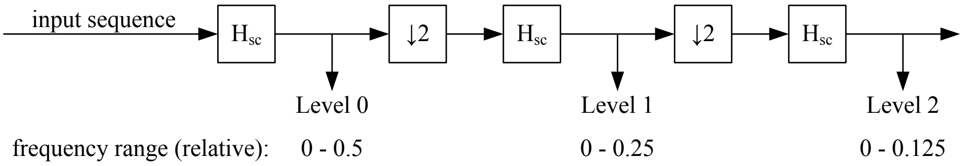


Figure 4.3 Scale-space filters and downsampling.

Note that the images on the different scales are used after scale-space filtering but before downsampling, leading to an oversampled representation. This redundancy is necessary to avoid aliasing when estimating speeds of up to 2 $pixels/frame$ per level (see section 4.1). This also means that even on the lowest level (level 0), spatial frequencies higher than 0.5 are suppressed. However, in practical image sequences, this frequency range is considered to contain no useful information [4].

4.3 Integration of Pyramid Levels

Basic Assumption

When looking at the results of one specific level m in the pyramid, it is not possible to determine if aliasing has occured or not. This can only be seen by checking the speeds on the next higher level $m+1$. However, one can only be sure that the speeds of level $m+1$ are reliable when also checking level $m+2, \ldots$

This makes it impossible to automatically find a good number of levels at runtime. Therefore, the user has to choose the maximum level according to his a priori knowledge of the image sequence at hand.

The basic assumption for this chapter is that the user always chooses a maximum level M which is high enough to handle also the highest velocities in the image sequence without aliasing.

Compensation

The key concept to estimate high velocities with a pyramid algorithm is compensation. Figure 4.4 shows a local ST-slice with a radius of 3 in the spatial dimension and a radius of 2 in the temporal direction to compute the speed of one output pixel (filled circle).

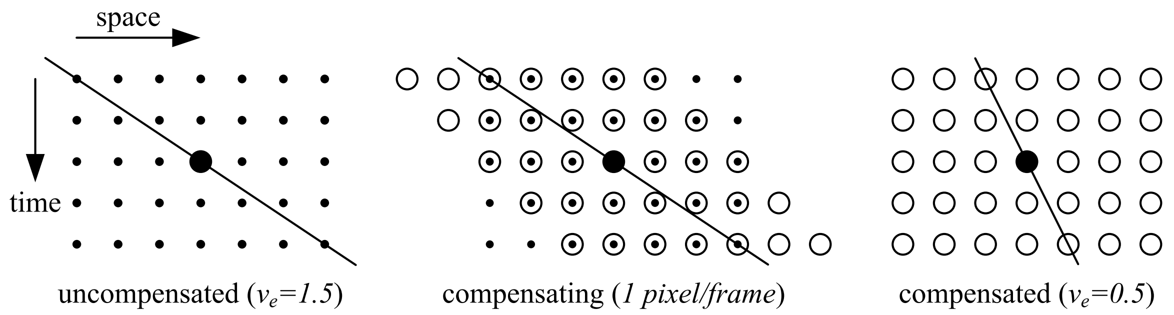


Figure 4.4 Compensation of high speeds.

In the uncompensated ST-slice (left), a speed of 1.5 $pixels/frame$ is estimated. By shearing the local ST-slice it is easily possible to compensate the estimated speed by an integer value (figure 4.4, center): Instead of looking at a rectangular area of the ST-slice (dots), a rhomboid-shaped area of the ST-slice is used to compute the component velocity (circles). Then, the compensated estimation is always in the range $[-0.5, 0.5]$ $pixels/frame$, which makes the estimation more reliable (figure 4.4, right).

Example

The numerical example in table 4.1 shows how the rounded component velocity of a level m can be used for compensation on the next lower level $m-1$. The goal is to estimate a velocity of 7.4 $pixels/frame$ with a pyramid implementation consisting of four levels.

m	$\mathbf{v_u(m)}$	$\mathbf{\Delta v(m)}$	$\mathbf{v_c(m)}$	$\mathbf{r_{0.5}\{v_c(m)\}}$
0	**7.400**	$8 \cdot r_{0.5}\{v_c(3)\} + 4 \cdot r_{0.5}\{v_c(2)\} + 2 \cdot r_{0.5}\{v_c(1)\} = 7$	0.400	
1	3.700	$4 \cdot r_{0.5}\{v_c(3)\} + 2 \cdot r_{0.5}\{v_c(2)\} = 4$	−0.300	−0.5
2	1.850	$2 \cdot r_{0.5}\{v_c(3)\} = 2$	−0.150	0.0
3	0.925		0.925	1.0

Table 4.1 Compensation of high speeds.

The $v_u(m)$ column shows the uncompensated velocity for each level. This is a theoretical value which can not directly be computed (unless the velocity on a particular level is less than $2\,pixels/frame$).

The next column, $\Delta v(m)$, is the compensation which is applied on level m. On the highest level, no compensation is carried out (the basic assumption is that the uncompensated velocity of the highest level is in a reasonable range).

The $v_c(m)$ column contains the velocities that are actually estimated (after compensation) on level m. In the last column, the compensated velocity estimations are rounded to multiples of 0.5 (the $r_y\{x\}$ function rounds the value x to the nearest multiple of y).

First, $v_c(3) = 0.925$ is estimated. This value is rounded to 1 which corresponds to a compensation of $2\,pixels/frame$ on level 2. With this compensation, a compensated velocity of $-0.15\,pixels/frame$ is estimated on level 2. This value is rounded to 0, which means that (on level 1) no additional compensation is contributed from level 2. However, the rounded estimate from level 3 has to be multiplied by 2 again, leading to a compensation of $4\,pixels/frame$ on level 1. Finally, on level 0, the compensation is twice the compensation used on level 1, plus twice the rounded estimate from level 1.

The estimated velocity on level 0 ($v_c(0) = 0.4$) plus the compensation used on level 0 ($\Delta v(0) = 7$) results in the sought velocity of $v = 7.4\,pixels/frame$.

Recursive Formulation

This process can be formulated recursively (see also figure 4.5):

$$\Delta v(m) = \begin{cases} 0 & m \geq M \\ 2 \cdot \left[\Delta v(m+1) + r_{0.5}\{v_c(m+1)\}\right] & m < M \end{cases} \tag{4.2}$$

$$v = \Delta v(0) + v_c(0) \tag{4.3}$$

The $r_{0.5}\{x\}$ function can be rewritten using the truncating function $\mathrm{Int}\{x\}$, which allows a direct implementation in C (using `float` to `int` conversion):

$$r_{0.5}\{x\} = 0.5 \cdot r_{1.0}\{2 \cdot x\} = \begin{cases} 0.5 \cdot \mathrm{Int}\{2 \cdot x + 0.5\} & x \geq 0 \\ 0.5 \cdot \mathrm{Int}\{2 \cdot x - 0.5\} & x < 0 \end{cases} \tag{4.4}$$

The recursion equation (4.2) can then be rewritten:

$$\Delta v(m) = \begin{cases} 0 & m \geq M \\ 2 \cdot \Delta v(m+1) + \mathrm{Int}\{2 \cdot v_c(m+1) + 0.5\} & m < M,\, v_c(m+1) \geq 0 \\ 2 \cdot \Delta v(m+1) + \mathrm{Int}\{2 \cdot v_c(m+1) - 0.5\} & m < M,\, v_c(m+1) < 0 \end{cases} \tag{4.5}$$

In the case where the confidence $c(m+1)$ of the estimated velocity $v_c(m+1)$ is below a threshold T, the velocity is not used. This leads to the final recursion formula:

$$\Delta v(m) = \begin{cases} 0 & m \geq M \\ 2 \cdot \Delta v(m+1) & m < M,\, c(m+1) \leq T \\ 2 \cdot \Delta v(m+1) + \mathrm{Int}\{2 \cdot v_c(m+1) + 0.5\} & m < M,\, c(m+1) > T,\, v_c(m+1) \geq 0 \\ 2 \cdot \Delta v(m+1) + \mathrm{Int}\{2 \cdot v_c(m+1) - 0.5\} & m < M,\, c(m+1) > T,\, v_c(m+1) < 0 \end{cases} \tag{4.6}$$

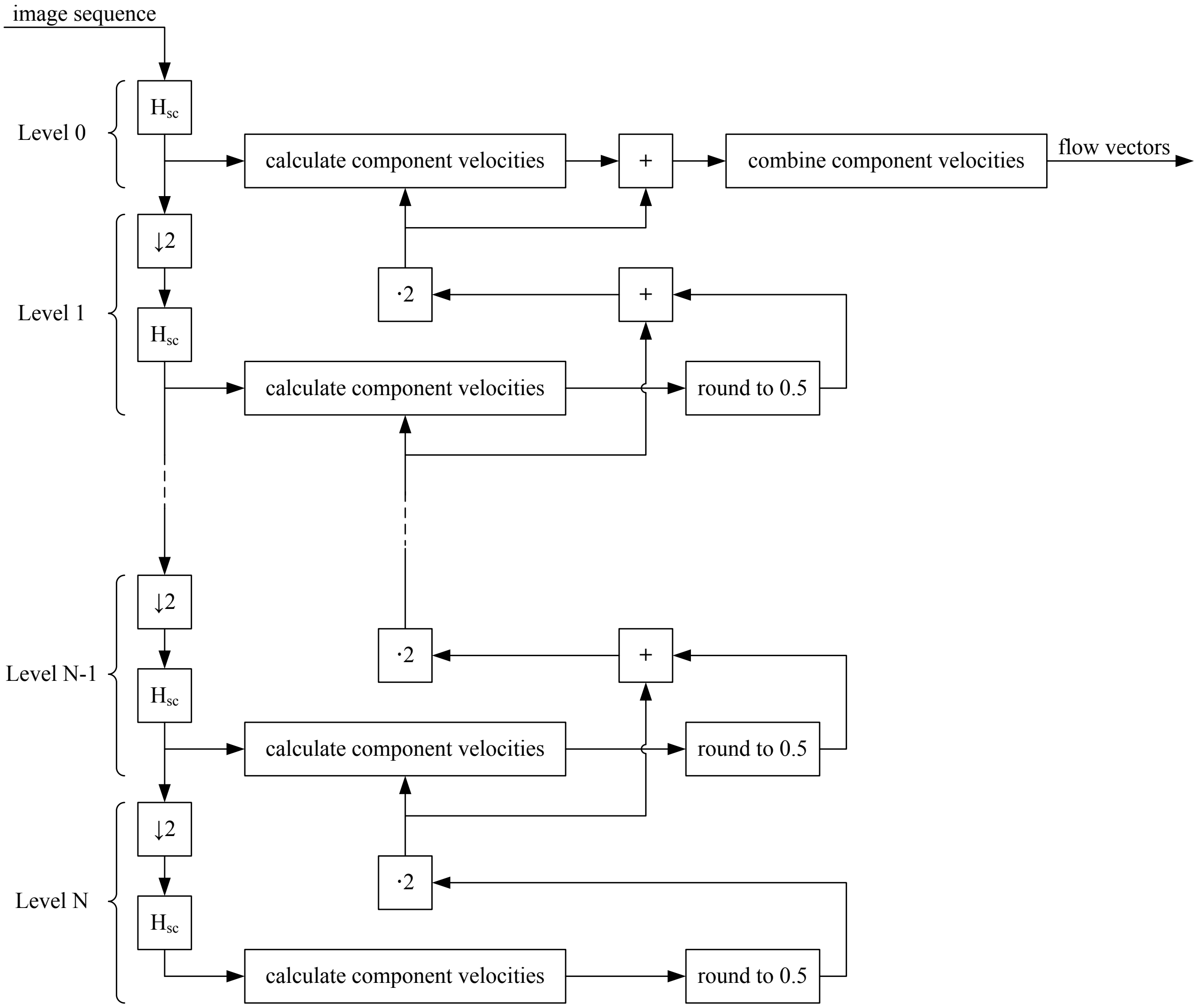


image sequence
Level 0
Hsc
calculate component velocities
+
combine component velocities
flow vectors
Level 1
↓2
·2
+
Hsc
calculate component velocities
round to 0.5
Level N-1
↓2
·2
+
Hsc
calculate component velocities
round to 0.5
Level N
↓2
·2
Hsc
calculate component velocities
round to 0.5

Figure 4.5 Program flow.

5 CUDA Implementation

5.1 Development Environment

Introduction

In the previous project, the CUDA C code was developed with the CUDA Toolkit 2.3 which did not allow to debug code that runs on a GPU, i.e. setting breakpoints in kernel functions was not possible.

At the beginning of this project, first beta versions of NVIDIAs new GPU development environment Parallel Nsight (codename Nexus) became available[1].

Configurations

Parallel Nsight is designed for remote debugging with a host system (where Visual Studio is installed) and a target system (with a CUDA compatible GPU). This setup did not work for debugging CUDA C code that is called from Matlab (*.mex* files).

It is also possible to use one single system as a host and target. In this case, two graphic cards are needed; a CUDA capable one for the debugging and another one for the screen. This configuration is not (yet) officially supported by NVIDIA, but it allows to debug CUDA C code in a *.mex* file with Parallel Nsight.

The next section describes the setup of this second configuration. Table 5.1 shows the software versions used during this project for the CUDA development. The CUDA Toolkit 2.3 was still used to compile the existing *.mex* files of the previous project [3]. The new *.mex* files were built using the CUDA Toolkit 3.0, which comes with Parallel Nsight.

C Code Debugging vs. Kernel Debugging

It is not possible to debug kernel code and the surrounding C code at the same time. When starting a project with the standard Visual Studio debugger, breakpoints in kernel code are ignored. When using Parallel Nsight as a debugger (use *Start CUDA Debugging*), breakpoints outside of the kernels are ignored.

Kernels Using Textures

With CUDA-GDB (which is NVIDIA's debugging environment for Linux systems), it is not possible to debug kernels which use texture memory [24]. It seems that Parallel Nsight has the same limitation, as it was not possible to debug such kernels during this project.

OpenCV

Information about building and running the OpenCV [25] algorithms can be found in appendix D.

[1] `http://www.nvidia.com/object/cuda_home_new.html`

Software	Version	Architecture
MathWorks Matlab	R2009b	Win64
Microsoft Visual Studio	2008 SP1	Win32
NVIDIA Display Driver	195.62	Win64
NVIDIA Parallel Nsight	1.0.10083.2	Win64
NVIDIA CUDA Toolkit	2.3 / 3.0	Win64
NVIDIA CUDA SDK	2.3	-

Table 5.1 Software versions used during this project.

5.1.1 System Setup

Requirements

Parallel Nsight requires Windows Vista SP1 or Windows 7 (32-bit or 64-bit in either case) on both the host and the target system.

For local debugging, two graphic cards are needed (and a motherboard that can carry them as well as a strong enough power supply). At least one of them must support hardware debugging, which is only available on devices with a CUDA compute capability of 1.1 or higher. For a more detailed description of the requirements, the readme of Parallel Nsight should be consulted before buying a new graphic card.

The second graphic card in the system which is used to drive the display(s) must not be CUDA capable and could in theory also be a non-NVIDIA model. However, tests with an ATI graphic card were not successful because it didn't show up in the NVIDIA control panel and it was therefore not possible to setup headless debugging.

Used Hardware

The following hardware was successfully used for local debugging:

NVIDIA GeForce GTX 260 Graphic Card This GPU has CUDA compute capability 1.3 and was used for debugging.

NVIDIA GeForce 8800 GTX Graphic Card This GPU has CUDA compute capability 1.0 and can therefore not be used for hardware debugging of CUDA kernels. This graphic card was used to drive the displays.

NVIDIA nForce 680i SLI Mainboard This mainboard has two PCI Express v1.0 x16 slots to hold two graphic cards. The GTX 260 GPU is built for PCI Express v2.0 but is backwards compatible to a PCI Express v1.0 mainboard.

ANTEC EarthWatts 750W Power Supply This is a SLI-ready 750W power supply. SLI-ready means that it is capable (regarding power and connectors) to drive two NVIDIA graphic cards at the same time. However, the two graphic cards in this setup are not running in SLI mode.

Besides this CUDA Workstation, a notebook was also used for development. Since the notebook has only one GPU, debugging of kernels was not possible with this system. However, other features of Parallel Nsight (Analysis Tools) were still available. Table 5.2 summarizes the hardware configuration of the two systems. Table 5.3 compares the specifications of the GPUs used during this project.

	CUDA Workstation	Notebook
Processor	Intel Core 2 Quad Q6600	Intel Mobile Core 2 Duo T9600
Number of Cores	4	2
Frequency	2400 *MHz*	2800 *MHz*
FSB Frequency	1066 *MHz*	1066 *MHz*
RAM	$2 \times 1024\,MB$	$2 \times 2048\,MB$
RAM Type	DDR2-SDRAM PC2-6400	DDR3-SDRAM PC3-8500
Mainboard	NVIDIA nForce 680i SLI	ASUSTeK N61Vn
Chipset	NVIDIA nForce 680i SLI SPP	Intel PM45
PCIe Version	1.0	1.0
PCIe Link Speed	$2.5\,GB/s$	$2.5\,GB/s$
GPU 1	NVIDIA GeForce GTX 260	NVIDIA GeForce GT 240M
GPU 2	NVIDIA GeForce 8800 GTX	-
Operating System	Windows 7 Enterprise	Windows 7 Ultimate
Architecture	64-bit	64-bit

Table 5.2 Development systems.

	GeForce GTX 260	GeForce 8800 GTX	GeForce GT 240M
CUDA Compute Capability	1.3	1.0	1.2
Number of Multiprocessors	27	16	6
Number of Cores	216	128	48
Global Memory	896 *MB*	768 *MB*	1024 *MB*
Constant Memory	64 *kB*	64 *kB*	64 *kB*
Shared Memory per Block	16 *kB*	16 *kB*	16 *kB*
Registers per Block	16384	8192	16384
Warp Size	32	32	32
Texture Alignment	256 *bytes*	256 *bytes*	256 *bytes*
Max. Threads per Block	512	512	512
Max. Block Dimension	$512 \times 512 \times 64$	$512 \times 512 \times 64$	$512 \times 512 \times 64$
Max. Grid Dimension	65535×65535	65535×65535	65535×65535
Clock Rate	1242 *MHz*	1350 *MHz*	1210 *MHz*
Memory Clock	1015 *MHz*	900 *MHz*	790 *MHz*
Memory Interface	448 *bit*	384 *bit*	128 *bit*

Table 5.3 GPU comparison. The two first GPUs belong to the CUDA Workstation, the latter is the Notebook's GPU.

Software To compile, run and debug the code of this project, the following software has to be installed on a single machine:

1. Matlab
2. Visual Studio

   Refer to the Parallel Nsight readme for more information about the supported development environments. For Parallel Nsight 1.0, only Visual Studio 2008 SP1 is supported.
3. NVIDIA Display Driver for the CUDA capable GPU

   Refer to the Parallel Nsight readme for information about the supported driver version. For the beta version of Parallel Nsight, the driver version has to match exactly and can not be newer than the one stated in the readme file.
4. NVIDIA CUDA Toolkit 2.3
5. NVIDIA CUDA SDK
6. NVIDIA Parallel Nsight Monitor
7. NVIDIA Parallel Nsight Host

Environment Variables To build the code of this project, some environment variables have to be set (see table 5.4). They are used in the custom makefile of this project. A reboot of the computer might be necessary before the new environment variables are recognized by Visual Studio.

Variable	Example
`MATLAB_DIR`	*C:\Program Files\MATLAB\R2009b*
`VC_BIN_DIR`	*C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin*
`CUDA_ROOT`	*C:\Program Files\NVIDIA Nexus 1.0\CUDA Toolkit\v3.0\Win64\CUDA*
`NVSDKCUDA_ROOT`	*C:\ProgramData\NVIDIA Corporation\NVIDIA GPU Computing SDK\C*

Table 5.4 Required environment variables.

Local Debugging Setup Some special setup steps are needed for local debugging. They are described in the Parallel Nsight User Guide. For the version used during this project, the following two steps were needed:

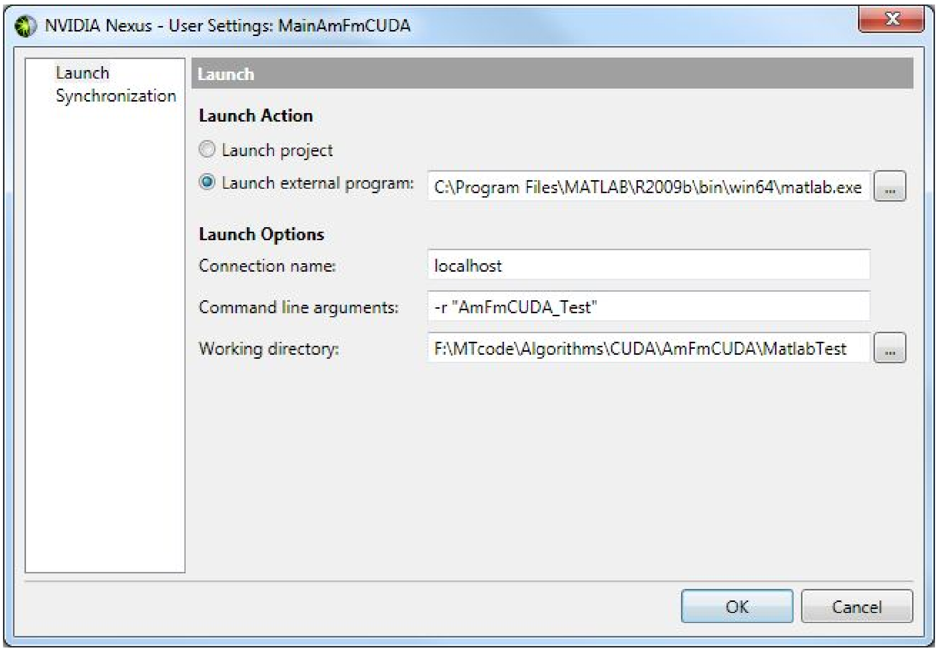
- Disable Direct3D acceleration for WPF using a *.reg* file that ships with Parallel Nsight.
- Setup headless debugging for the Parallel Nsight capable GPU.

Parallel Nsight Setup Some of the Parallel Nsight settings are global and some are project specific.

The global settings for the Monitor can be modified by right-clicking on the Parallel Nsight Monitor icon in the taskbar and selecting *Options*. Here, the *WDDM TDR* and the *Enable secure server* options were disabled.

The global settings for the Host can be modified from Visual Studio by selecting the *Nexus→Options...* menu. Here, the *Enable secure connection* option was disabled.

The project specific settings can be accessed by right-clicking on the project in the Visual Studio Solution Explorer and selecting *Nexus User Properties*. Figure 5.1 shows the settings used for this project.



Launch Action

○ Launch project

● Launch external program: C:\Program Files\MATLAB\R2009b\bin\win64\matlab.exe

Launch Options

Connection name: localhost

Command line arguments: -r "AmFmCUDA_Test"

Working directory: F:\MTcode\Algorithms\CUDA\AmFmCUDA\MatlabTest

Figure 5.1 Nexus user properties.

Launch external program: It is important to not enter *...\bin\matlab.exe* here, because this is just a small starter application which returns immediately to the calling process (and therefore stops the debugging immediately).

Connection name: Use *localhost* for local debugging.

Command line arguments: This value is passed to Matlab as a command line argument. The `-r` option specifies an *.m* file to run after the startup of Matlab. Note that no path information or file extension should be supplied here.

Working directory: This setting defines the current folder of Matlab after startup. It should be set to the directory where the *.m* file specified above is located.

Analysis Tools

The settings for the Analysis Tools (menu *Nexus*→*New Analysis Activity*) are similar to the ones described above.

Note that the Analysis Tools can also be used in a local setup when only one CUDA capable GPU is available.

Debugging C Code

To debug standard C code (without Parallel Nsight) in a *.mex* file, the project properties *Command*, *Command Arguments* and *Working Directory* must be set according to the explanations above.

5.1.2 Troubleshooting

No CUDA-GPU found

The Matlab GUI shows this message also when it was not able to successfully call the *cuGpuDeviceInfo .mex* file. See next point for possible reasons.

Invalid MEX file

The message *Invalid MEX-file . . . : The specified module could not be found* can have two reasons (at least. . .):

- The *.mex* file can not be found in the search paths. Note that Matlab looks for a *.mexw32* or a *.mexw64* file, depending on the Matlab version (32- or 64-bit).
- One or more dependencies of the *.mex* file can not be found. The missing dependency is most probably the CUDA runtime DLL. For the *.mex* files compiled with CUDA Toolkit 2.3 (existing *.mex* files from the previous project), this is *cudart.dll*. When the CUDA Toolkit 2.3 is properly installed, this file should be found because its location is added to the Path environment variable. For new *.mex* files built with the CUDA Toolkit 3.0, the CUDA runtime is a file like *cudart64_30_8.dll*. This file should be in the same directory as the *.mex* file. To identify missing dependencies, the freeware *Dependency Walker*[2] can be very useful.

5.2 Build Process

5.2.1 Win64 Compatibility

Introduction

In order to be able to run the software on Win32 as well as on Win64 systems, some modifications to the code and the build setup were necessary.

5.2.1.1 Data Types

Win32 vs. Win64

Table 5.5 shows the size of different data types on Win32 and Win64 [26].

Fortunately, the float datatypes (which are used to store the image data) remain the same and are also compatible with Matlab (C and Matlab use the IEEE 754 standard).

Problem

The MEX interface of Matlab uses the `mwSize` and `mwIndex` data types for dimensions and indexes. While it is not a problem to cast between `mwSize` (or `mwIndex`) and `unsigned int` on Win32, it can be dangerous on Win64; especially when a casted value is passed to a function by reference.

Modification of the Existing Code

The existing Lucas-Kanade implementation from the previous work [3] was modified to avoid any casts from `mwSize` or `mwIndex` to `unsigned int` or `int`. Therefore, the data types of many local variables and function parameters have been changed to `mwSize`.

`WIN64_CAST_WARNING`

The signatures of the CUDA kernels in the existing Lucas-Kanade implementation use the `int` data type for dimensions (of the image sequence, for example). These signatures have not been changed. Therefore, when calling these kernels, a cast from `mwSize` to `int` must be performed. This reduces the available data range; even on Win32 (because the destination is a signed type). Here the assumption is that all dimensions and filter lengths are in a reasonable range. However, all these casts were marked with a `WIN64_CAST_WARNING` comment.

[2] http://dependencywalker.com/

Data Type	Win32	Win64
char	8	8
short	16	16
int, long	32	32
long long	64	64
Pointers (ptrdiff_t, . . .)	32	64
size_t	32	64
time_t	32	64
float	32	32
double	64[1)]	64
Matlab single	32	32
Matlab double	64	64
MEX type mwSize	32[2)]	64
MEX type mwIndex	32[2)]	64

[1)] Depends on compiler and compiler settings. In our configuration it is 64 bits.

[2)] Might change to 64 bits in the future [27]. In our configuration it is 32 bits.

Table 5.5 Size of data types (in bits) on Win32 and Win64 systems.

MWSIZE_WARNING

The functions of the CUDA API use the size_t type for dimensions. Casting from mwSize is therefore not a problem. However, should in future Matlab versions the mwSize data type be extended to 64 bits on Win32 systems [27], this would lead to a cast from a 64-bit integer to a 32-bit integer. This is not a problem because (again) we assume reasonable dimensions. However, the assignments of a mwSize value to a size_t value were marked with a MWSIZE_WARNING comment.

New Code

The implementation of the new algorithm uses unsigned int for dimensions. Only at the very beginning, the dimensions of the Matlab data are casted from mwSize to unsigned int. These casts are marked with a MWSIZE_WARNING comment.

In calculations which can result in a negative index, a cast from unsigned int to int is done. These casts are marked with a SIGNED_CAST_WARNING comment.

5.2.2 Makefile

Introduction

To build a CUDA project, the *.cu* files (which contain the source code of the CUDA kernels) must be compiled by invoking the CUDA Compiler Driver (NVCC). In the previous work [3], a custom makefile was used to realize this. The makefile of this project is based on the existing one, but it was modified to support compilation on Win32 systems as well as on Win64 systems.

Input Parameters

Depending on the desired build configuration and target system, a combination of the following "input parameters" can be used. They affect several compiler and linker command line parameters and select the appropriate versions of the libraries to link.

`dbg=1` Causes the compiler to generate debug information.

`emu=1` Passes the `--device-emulation` option to the NVCC compiler. This allows to run CUDA kernels on a system where no CUDA capable GPU is available. This feature was not used during this project.

`x64=1` This is used when building the code on a Win64 system.

Example To rebuild the debug configuration on a Win64 system, the build command line in the project properties of Visual Studio looks like: `nmake rebuild dbg=1 x64=1`

NVCC vs. CL With the current makefile, all *.cpp* and *.cu* files are passed to the CUDA Compiler Driver (NVCC). NVCC extracts and processes all GPU kernel code. NVCC then calls the Microsoft C/C++ Compiler (CL) to process the rest of the code. The `--compiler-options` argument of NVCC can be used to specify the arguments that should be passed to CL.

Shortcomings The current version of the makefile has some shortcomings:

- The dependencies in the build rules are not complete. Preferably, the list of dependencies would be generated automatically. With this current makefile it is recommended to always use the rebuild command (instead of doing an incerementental build).
- A separate build rule is needed for every source file. It would be nicer to have a generic rule for all *.cu* and *.cpp* files.
- Cross-compiling is not supported. It is not possible to build the *.mexw32* file on a Win64 machine and vice versa.

5.3 General Information about the Implementations

Introduction

The entrance point of the *.mex* file is the `mexFunction` function in the file *MainAmFmCUDA_MexInterface.cpp*. It has a defined signature [28] which allows Matlab to exchange parameters and return values with the *.mex* file.

`mexFunction`

`mexFunction` is also the main function of the CUDA implementation. It performs the following actions:

- Checking input parameters and throwing exceptions when parameters with a wrong data type, dimension, or value are passed.
- Copying the original image sequence to the global GPU memory and copying the results back to the host memory.
- Allocating and releasing memory on the GPU.
- Handling the data flow and the program logic by instantiating the worker objects and calling their functions.

Worker Objects

All CUDA kernels are encapsulated in classes which have (at least) a function to initialize itself for a given set of parameters and a function to actually perform the work (calling the CUDA kernel).

Note that it was not possible to declare the CUDA texture references as private class members. Therefore, all objects of a class share the same texture reference.

Block Dimensions and Image Dimensions

The optimum number of threads per block depends on many factors [21, 22]. Small block sizes yield to many blocks per multiprocessor which is in general good for the utilization of the hardware (reduced number of idle multiprocessors). Large block sizes on the other hand can for example reduce redundant memory reads from global memory (when using shared memory; see section 5.4.4).

Since the number of threads per block should always be a multiple of the warp size, the minimum block size is 32 (see table 5.3). The maximum block size is limited by the number of registers and the amount of shared memory needed.

In this project, all kernels use a block size of 256 (16×16). To avoid diverging branches within a block, all image dimensions are extended to a multiple of 16 in both directions. This also makes the kernel code more readable, since no boundary checks are needed.

Extending the image dimensions to a multiple of 16 has also the advantage that every image in a sequence is nicely aligned in memory, as long as the first image is aligned (which is ensured by `cudaMalloc`) [23]. Textures, for example, can only be bound to 256 *byte* aligned addresses in global memory with the GPUs used during this project (see table 5.3).

Memory Organization

C and Matlab use different schemes to store a 2D array in the (linear) memory. Matlab uses column-major order, C uses row-major order. This means, that the role of the x- and y- coordinates have to be exchanged when passing data from Matlab to a *.mex* file [3].

5.4 Basic Algorithm

Introduction This section describes the CUDA implementation of the algorithm explained in chapter 3.

5.4.1 Using the MEX File

Introduction The *.mex* file can be called from Matlab like any other Matlab function (provided that it is in the working directory or in a search path together with the CUDA runtime DLL). It has the following signature. Note that the data types have to match exactly.

```
[Flow Cert Time] = AmFmCUDA(ImgSeq, Pads, NumDir, AngleOffset, RadSpat,
   RadTemp, CompVelThr, OptFlowThr)
```

ImgSeq A three-dimensional ($h \times w \times l$) array of `single` values (`float` in C) representing the source image sequence. The length l of the sequence has to be at least $(2 \cdot \text{RadTemp} + 3)$.

Pads Number of pixels (`uint32`) to add on each side of the original images (padding). The default value used in the Matlab GUI is 15. Note that the effective padding on the right and the bottom side can be extended by up to 15 pixels since the (padded) image dimensions are extended to a multiple of 16 (see section 5.3).

NumDir and AngleOffset These arguments are used to determine the angles φ_k of the directional filters:

$$\varphi_k = \frac{180^\circ}{\text{NumDir}} \cdot k + \text{AngleOffset} \qquad k = 0 \ldots \text{NumDir} - 1 \tag{5.1}$$

`NumDir` can also be 0. This special case is described below. The `AngleOffset` argument is particularly useful when only one direction is computed (i.e. when `NumDir` is 1 or 0).

`NumDir` has data type `uint32` and is typically 4, `AngleOffset` has data type `double` and is typically 0.

RadSpat and RadTemp These two arguments (`uint32`) define the dimensions of the neighbourhood in which the instantaneous frequencies are averaged to compute the component velocities (local ST-slice, see section 3.4). `RadSpat` is the radius along the directional filter direction and `RadTemp` is the temporal radius. Typical values for `RadSpat` and `RadTemp` are 3 and 1, respectively.

Note that $(2 \cdot \text{RadTemp} + 1)$ frames of instantaneous frequencies are needed to compute one frame of component velocities. The computation of one frame of instantaneous frequencies also has a temporal radius of 1 (to build the forward and backward differences). Therefore, $(2 \cdot \text{RadTemp} + 3)$ input frames are needed for one output frame.

CompVelThr This is the threshold for computing the component velocities mentioned in section 3.4. Its data type is `single` and a typical value is 0.8.

OptFlowThr This is the threshold for computing the optical flow vectors mentioned in section 3.5. Its data type is `single` and a typical value is 0.8.

This parameter is ignored when `NumDir=0` (see below).

`Flow` This output parameter returns the the flow vectors. It is a complex array of `single` values with dimensions ($h \times w \times (l - 2 \cdot \text{RadTemp} - 2)$). The real and imaginary parts of the values are the horizontal and vertical components of the flow vectors, respectively.

`Cert` This output parameter returns the confidence measure for the flow vectors. It is an array of `single` values with dimensions ($h \times w \times (l - 2 \cdot \text{RadTemp} - 2)$). The values are in the range between 0 ('not reliable' or 'error during computation') and 1 ('highly reliable').

`Time` This is an optional output parameter of type `single` (`float` in C). It returns the total execution time of the mex (without calling overhead of Matlab) in milliseconds. The time is measured with CUDA events and has therefore a good resolution.

Special Case Calling the *.mex* file with `NumDir=0` calculates the component velocities for one direction (which can be specified by the `AngleOffset` parameter). However, the last step of the algorithm (section 3.5) is not applied. Instead, `Flow` (which is not a complex array in this case) returns the component velocities and `Cert` returns the certainties of the component velocities for that particular direction.

5.4.2 Unoptimized Implementation

Introduction Figure 5.2 shows the program flow of the first unoptimized implementation of the Basic Algorithm. Basically there is one kernel for each step described in chapter 3, as well as a kernel for padding the image sequence at the beginning.

Padding The image sequence received from Matlab is copied to a block of memory on the GPU which is big enough to hold the padded and extended image sequence. With `cudaMemcpy3D`, the (linear) GPU memory is viewed as a three-dimensional cuboid and it is possible to copy the data with an offset within this cuboid [23]. Figure 5.3 illustrates the three-dimensional memory block after copying the original data with an offset of ($pads, pads, 0$). The gray area depicts uninitialized memory.

The padding kernel then fills the uninitialized memory with the corresponding border values of the images.

Rotating The rotating and back-rotating kernels basically implement equations (3.12) and (3.13) without taking advantage of the texture memory. Padding is not done implicitly during rotation, therefore $pads = 0$ in equations (3.12) and (3.13).

Directional Filtering The directional filtering is split into two kernels, one for the longitudinal and one for the transversal dimension. This unoptimized implementation follows a straightforward approach without using shared memory or texture memory. Because the filters are quite long, this leads to a huge amount of (slow) global memory accesses. The filter coefficients are stored in fast constant memory before the first kernel call.

Instantaneous Frequencies This kernel implements the algorithm described in section 3.3. This kernel also reads filter response values from the previous and the next frame. This is easily possible since the filter responses of all frames are computed before the instantaneous frequency kernel is executed for the first time.

Component Velocities The component velocity kernel computes the component velocities and its certainties as described in section 3.4. This kernel also reads instantaneous frequency values from previous and successive frames.

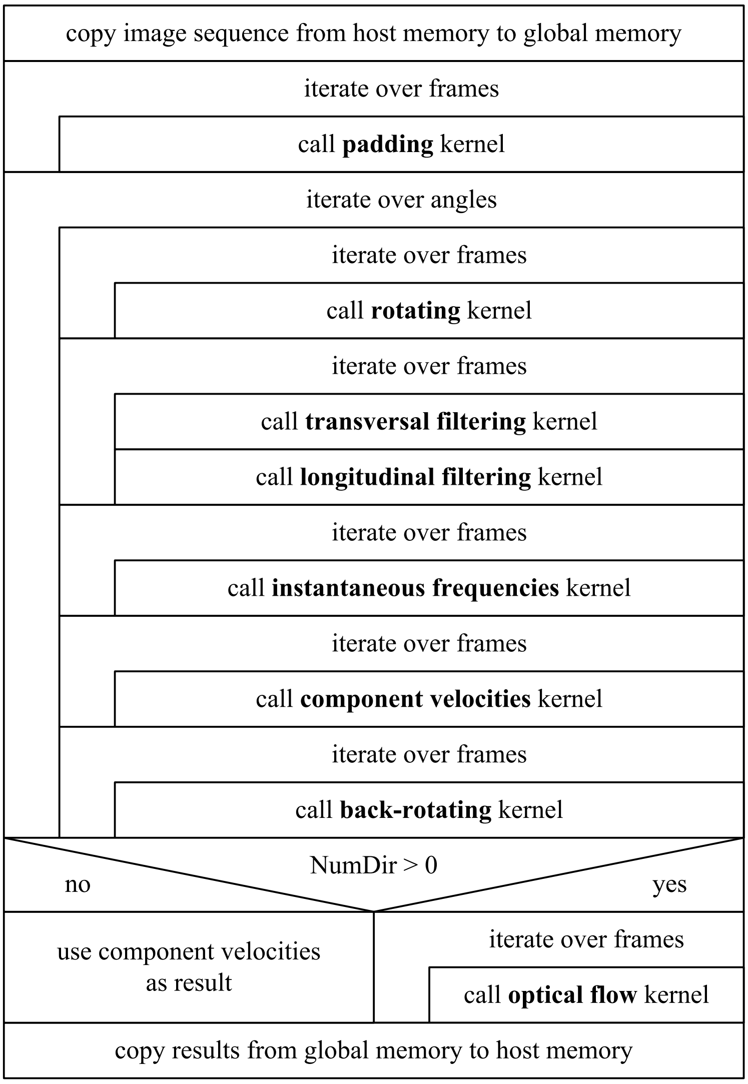

copy image sequence from host memory to global memory
iterate over frames
call **padding** kernel
iterate over angles
iterate over frames
call **rotating** kernel
iterate over frames
call **transversal filtering** kernel
call **longitudinal filtering** kernel
iterate over frames
call **instantaneous frequencies** kernel
iterate over frames
call **component velocities** kernel
iterate over frames
call **back-rotating** kernel
NumDir > 0
no
yes
use component velocities as result
iterate over frames
call **optical flow** kernel
copy results from global memory to host memory

Figure 5.2 Structogram: Basic Algorithm - unoptimized.

Optical Flow

This kernel combines the component velocities of all directions to the resulting optical flow vectors. The $\cos(\varphi_k)$, $\sin(\varphi_k)$, $\cos^2(\varphi_k)$, $\sin^2(\varphi_k)$ and $\cos(\varphi_k)\sin(\varphi_k)$ terms are precomputed and stored in the fast constant memory of the GPU.

The component values and its certainties are stored in a separate buffer for every direction. But passing an array of pointers (`float**`) to a kernel does not work since the first dereferenciation would try to access host memory which is not possible out of a CUDA kernel. Therefore, the start addresses of all buffers are stored in the constant memory of the GPU before the kernel is called. This is done only once. To access a specific frame within these buffers, an offset argument is passed to the kernel on every call.

Program Flow

This unoptimized implementation stores all intermediate results in the GPU memory and the memory is allocated for every direction separately. This results in a huge memory consumption and a large number of memory allocation operations, which is time consuming.

Optimizations

In the next sections, the kernels itself as well as the program structure are optimized step by step.

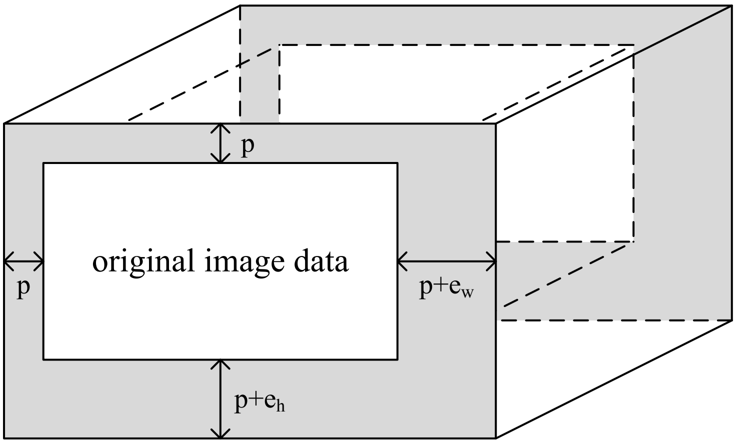


Figure 5.3 GPU memory for padded (with p pixels) and extended (width and height with e_w and e_h pixels, respectively) image sequence.

Performance Measures To measure the improvements of the optimization steps, the execution times of the kernels and the total execution time of the *.mex* file were recorded.

The execution times for the kernels were taken from the Parallel Nsight Analysis Report. The numbers in the following tables are the mean values of five runs.

The total execution time of the *.mex* file was measured using the CUDA event framework. Two events were captured at the beginnig and at the end of the `mexFunction` (see section 5.4.1, third output argument). The numbers in the following tables are the mean values of ten runs.

The Analysis Tools seem to add some overhead when executing the *.mex* file. The total execution time returned from the CUDA event framework is 10 to 20 milliseconds longer when the execution is started by the Analysis Tools. The total execution times in the following tables were therefore measured without the Analysis Tools.

Setup The *.mex* files were compiled and benchmarked with the January 2010 Beta of Parallel Nsight (version 1.0.10013) in Release configuration and executed on the CUDA Workstation (see table 5.2). The algorithm was applied to the Rubik Cube sequence (see appendix C).

The algorithm was called with the default parameters (15 pixels padding, 4 directions, no angle offset, spatial radius 3, temporal radius 1, thresholds 0.8).

Unoptimized Timing Table 5.6 shows the execution times for the unoptimized algorithm.

Kernel	%	Time [ms]	Calls
Total	100.0	212.25	1
Total Kernels	41.5	88.04	570
Transversal Filtering	16.2	34.42	84
Component Velocities	10.7	22.78	68
Longitudinal Filtering	6.6	13.91	84
Instantaneous Frequencies	3.8	8.13	76
Back-Rotating	2.2	4.60	136
Rotating	1.4	2.95	84
Optical Flow	0.4	0.89	17
Padding	0.2	0.36	21

Table 5.6 Timing: unoptimized (SW rev. 74).

5.4.3 Texture Memory for Image Rotation

Introduction

The unoptimized implementation is not very elegant:

- Reading the source image from global memory is not coalesced (whereas the writing of the rotated image is perfectly fine since the algorithm iterates linearly over the output pixels).
- In general, the transformed destination coordinates do not result in integer source coordinates. The unoptimized version rounds the coordinates to the nearest neighbours.
- In general, the rotated image is bigger than the source image. Therefore, access violations must be prevented using boundary checks when iterating over the destination image. The boundary checks lead to diverging branches.

Texture Memory

All these drawbacks can be avoided using texture memory (see section 2.3.2). This new implementation avoids uncoalesced global memory reads and uses the texture cache instead (since there is good 2D spatial locality when reading the source image). Furthermore, the sophisticated addressing modes allow an implicit bi-linear interpolation as well as implicit padding of the source image.

Performance

As can be seen from table 5.7, the rotating and back-rotating kernels perform much better and the padding kernel could be omitted. However, the over all speedup is not very impressive, since these three kernels only needed a small fraction of the total execution time. It also must be mentioned that binding the texture to a new global memory area for every frame adds some overhead on the host side.

Kernel	%	Time [ms]	Calls
Total	100.0	210.98	1
Total Kernels	40.2	84.88	549
Transversal Filtering	16.3	34.42	84
Component Velocities	11.0	23.28	68
Longitudinal Filtering	6.6	13.90	84
Instantaneous Frequencies	3.9	8.15	76
Back-Rotating	1.1	**2.41**	136
Rotating	0.9	**1.97**	84
Optical Flow	0.3	0.73	17

Table 5.7 Timing: texture memory for image rotation (SW rev. 76).

5.4.4 Optimization of Directional Filtering

Introduction

The goal for optimizing the directional filtering is to reduce the numer of (redundant) read accesses to global memory. Two approaches have been tested: one using shared memory and one using texture memory.

Shared Memory

The idea of the shared memory approach is to first load an area from global memory to shared memory and then in a second step to compute the convolution with the values stored in shared memory.

A little modification of the filter lenghts (49 instead of 55 for the transversal filter and 17 instead of 15 for the longitudinal filter) allows a scheme where every thread of the transversal kernel loads exactly 4 values into shared memory and every thread of the longitudinal kernel loads exactly 2 values into shared memory. After loading the values into shared memory, every thread computes one output value. The additional amount of memory that must be loaded into shared memory on both sides of a block corresponds to the filter radius (24 for the transversal filter and 8 for the longitudinal filter).

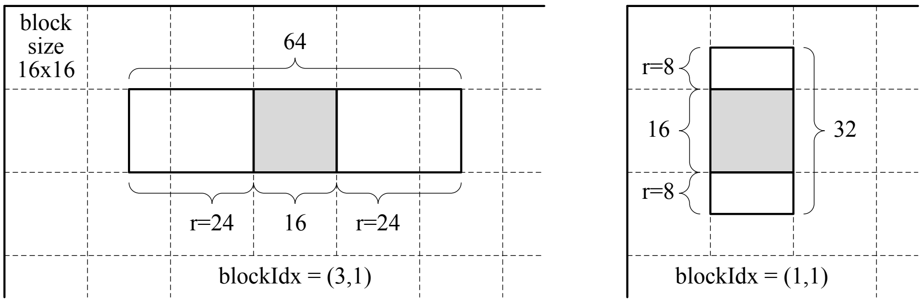


Figure 5.4 This image shows the current thread block (gray) and the memory area it loads into shared memory for transversal (left) and longitudinal (right) filtering.

It is possible to copy the data in such a way that it is read from global memory in a coalesced manner and written to shared memory without bank confilcts.

To ensure that all data is copied to shared memory before the first thread starts using the data, the `__syncthreads()` intrinsic function must be inserted as a barrier between memory copy and the actual convolution. It synchronizes all threads in a thread block.

Performance of Shared Memory Approach

To allow a fair comparison between the unoptimized kernels and the ones using shared memory, the timing of the unoptimized version was measured with the new filter lengths, too (see table 5.8). Table 5.9 shows the timing for the optimized version. The speedup for the transversal kernel is 2 and also the speedup for the longitudinal kernel is remarkable. Note that this optimization technique produces no additional overhead on the host side (in contrast to using texture memory).

Kernel	**%**	**Time [ms]**	**Calls**
Total	100.0	205.10	1
Total Kernels	40.4	82.88	549
Transversal Filtering	15.0	**30.84**	84
Component Velocities	11.4	23.30	68
Longitudinal Filtering	7.5	**15.46**	84
Instantaneous Frequencies	4.0	8.16	76
Back-Rotating	1.2	2.41	136
Rotating	1.0	1.97	84
Optical Flow	0.4	0.73	17

Table 5.8 Timing: modified directional filter lengths (SW rev. 79).

Kernel	**%**	**Time [ms]**	**Calls**
Total	100.0	183.97	1
Total Kernels	34.0	62.55	549
Component Velocities	12.7	23.31	68
Transversal Filtering	8.1	**14.91**	84
Longitudinal Filtering	6.0	**11.05**	84
Instantaneous Frequencies	4.4	8.16	76
Back-Rotating	1.3	2.41	136
Rotating	1.1	1.97	84
Optical Flow	0.4	0.73	17

Table 5.9 Timing: directional filtering using shared memory (SW rev. 82).

The speedup can be explained with the reduced number of redundant reads of source values. In the unoptimized version, every pixel of the source image is read 49 times from global memory (transversal filtering). Using shared memory, this number is reduced to four reads. For the longitudinal filtering it is 17 vs. 2 reads of the same source value.

Texture Memory

Using texture memory instead of shared memory to optimize the directional filtering leads to a very compact implementation of the kernels since no copying and no boundary checking has to be performed. However, binding the texture to the global memory introduces some overhead on the host side.

Tables 5.10 and 5.11 show the timing measurements for this optimized version. Table 5.10 is with the original filter lengths and table 5.11 is with the filter lenghts used for the shared memory approach.

Kernel	**%**	**Time [ms]**	**Calls**
Total	100.0	181.35	1
Total Kernels	32.7	59.34	549
Component Velocities	12.8	23.22	68
Transversal Filtering	8.8	**16.03**	84
Instantaneous Frequencies	4.5	8.16	76
Longitudinal Filtering	3.8	**6.82**	84
Back-Rotating	1.3	2.42	136
Rotating	1.1	1.97	84
Optical Flow	0.4	0.73	17

Table 5.10 Timing: directional filtering using texture memory (SW rev. 83).

Kernel	**%**	**Time [ms]**	**Calls**
Total	100.0	180.17	1
Total Kernels	32.2	58.00	549
Component Velocities	12.9	23.23	68
Transversal Filtering	7.8	**14.10**	84
Instantaneous Frequencies	4.5	8.17	76
Longitudinal Filtering	4.1	**7.38**	84
Back-Rotating	1.3	2.41	136
Rotating	1.1	1.97	84
Optical Flow	0.4	0.73	17

Table 5.11 Timing: directional filtering using texture memory (SW rev. 84).

Optimized Version

The two optimization approaches led to almost the same execution time for the transversal kernel (tables 5.9 and 5.11). Since the shared memory approach does not introduce overhead on the host side, this version was finally used for transversal filtering.

For the longitudinal filtering, the texture memory approach is more efficient (tables 5.9 and 5.10). Table 5.12 shows the timing for the fully optimized directional filtering which uses shared memory for longitudinal filtering and texture memory for transversal filtering.

Kernel	%	Time [ms]	Calls
Total	100.0	179.04	1
Total Kernels	32.56	58.30	549
Component Velocities	13.0	23.29	68
Transversal Filtering	8.3	**14.91**	84
Instantaneous Frequencies	4.6	8.16	76
Longitudinal Filtering	3.8	**6.83**	84
Back-Rotating	1.3	2.41	136
Rotating	1.1	1.97	84
Optical Flow	0.4	0.73	17

Table 5.12 Timing: optimized directional filtering (SW rev. 85).

5.4.5 Using Vector-Datatypes

Vector-Datatypes

In this optimization step, some of the `float` buffers were pairwise combined to `float2` buffers (see table 5.13). `float2` is a vector data type provided by CUDA with the fields `x` and `y`.

New `float2` Buffer	Content of the `x` Field	Content of the `y` Field
`gFilterResp`	`gFilterRespRe`	`gFilterRespIm`
`gInstFreq`	`gInstFreqSpat`	`gInstFreqTemp`
`gCompVel`	`gCompVel`	`gCompVelConf`
`dCompVelRotPtrs`	`dCompVelRotPtrs`	`dCompVelConfRotPtrs`

Table 5.13 Mapping between new `float2` buffers and old `float` buffers.

Other candidates to pair together would be the resulting flow vector components `gVelX` and `gVelY`. However, these values are returned to Matlab as real and imaginary parts in a complex valued array. Matlab expects the real and imaginary part to be in separate buffers wheras the use of `float2` would tumble them.

Special Case

For the special case where the component velocities of one direction should be returned instead of the flow vectors (`NumDir=0`, see section 5.4.1), a new kernel to copy the component velocities to the output buffer was introduced (see also figure 5.5). It reads the `float2` values from the `gCompVel` buffer and stores their component velocity values and their confidence values into separate buffers so that they can be returned to Matlab.

Texture Memory

Vector data types can not only be used with global and shared memory, but also with texture memory. Listing 5.1 shows the declaration of the texture and the complete back-rotating kernel. Line 18 reads the component velocity and its confidence value for the current output pixel at the same time and writes these values to the global memory. Both values (`x` and `y` fields) are bi-linear interpolated when read from texture memory.

Listing 5.1 Back-rotating of component velocities and their confidence values using texture memory.

```
texture<float2, 2, cudaReadModeElementType> texBackRot;

__global__ void texbackrotating_Kernel(float2* dstImg,
   unsigned int pitchedOutputWidth, float cosTheta, float sinTheta,
   float minX, float minY, unsigned int pads)
{
    // calculate normalized texture coordinates
    unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int idy = blockIdx.y*blockDim.y + threadIdx.y;

    // transform coordinates
    float tu = (float)(idx+pads)*cosTheta - (float)(idy+pads)*sinTheta
       - (float)minX;
    float tv = (float)(idy+pads)*cosTheta + (float)(idx+pads)*sinTheta
       - (float)minY;

    // read from texture and write to global memory
    dstImg[idy*pitchedOutputWidth + idx] = tex2D(texBackRot, tu, tv);
}
```

Performance

As can be seen from table 5.14, the two kernels using texture memory for reading (back-rotation and longitudinal filtering) benefit from this new data type. There is also an improvement for the instantaneous frequencies kernel, which reads `float2` values from global memory.

But there was no improvment for the component velocities kernel, and the optical flow kernel needs even more time than before.

However, there was still some over all improvement which is assumed to be because of the reduced number of memory allocation operations needed.

Kernel	%	Time [ms]	Calls
Total	100.0	169.66	1
Total Kernels	34.0	57.67	481
Component Velocities	13.7	**23.30**	68
Transversal Filtering	8.8	14.91	84
Instantaneous Frequencies	4.6	**7.74**	76
Longitudinal Filtering	4.0	**6.74**	84
Rotating	1.2	1.97	84
Back-Rotating	0.9	**1.58**	68
Optical Flow	0.8	**1.42**	17

Table 5.14 Timing: Basic Algorithm with vector datatypes (SW rev. 90).

5.4.6 Optimized Program Flow

Goals

The last and most important optimization step was to revise the structure of the `mexFunction`. The goals were:

- Reduce the number of texture binding operations.
- Reduce the number of memory allocation operations.
- Reduce the memory consumption.
- Simplify the structure to have a good basis for the implementation of the multiresolution algorithm.

New Structure

The new structure is shown in figure 5.5. Basically it processes frame by frame (compare with figure 5.2). Therefore, a source image can be loaded into texture memory once and then be rotated several times. This reduces the overhead on the host side.

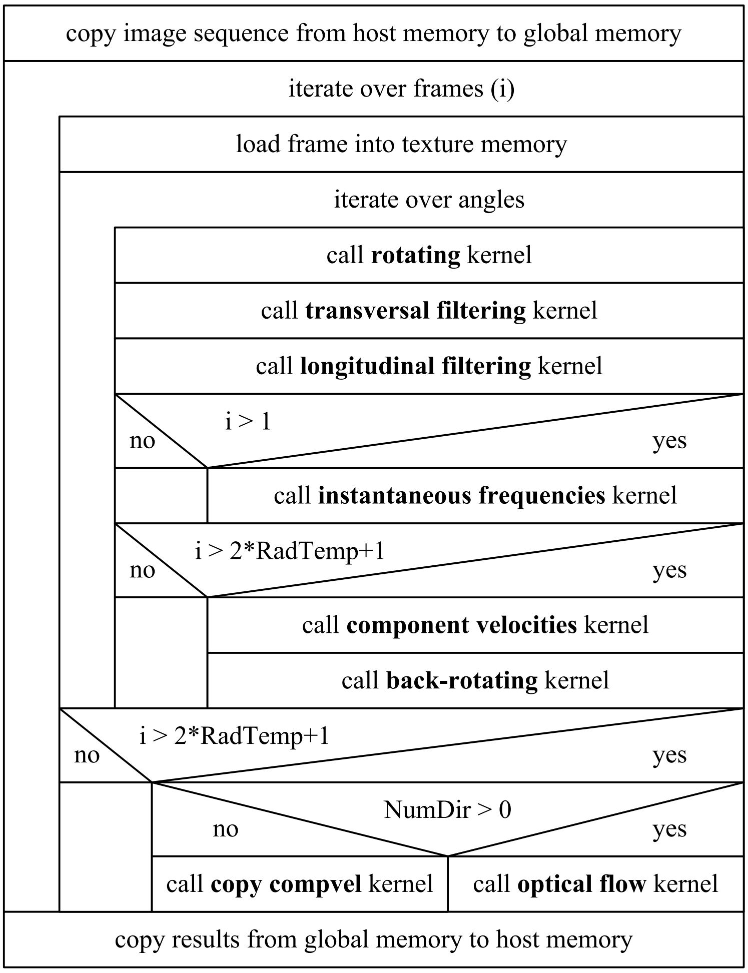

copy image sequence from host memory to global memory
iterate over frames (i)
load frame into texture memory
iterate over angles
call **rotating** kernel
call **transversal filtering** kernel
call **longitudinal filtering** kernel
i > 1
no
yes
call **instantaneous frequencies** kernel
i > 2*RadTemp+1
no
yes
call **component velocities** kernel
call **back-rotating** kernel
i > 2*RadTemp+1
no
yes
NumDir > 0
no
yes
call **copy compvel** kernel
call **optical flow** kernel
copy results from global memory to host memory

Figure 5.5 Structogram: Basic Algorithm - optimized.

Memory Allocations

Since the rotated images have different dimensions, in the unoptimized version, memory was allocated according to the actual image dimensions when iterating over the angles. This led to many memory allocation and release operations which increased the execution time on the host side.

With this new structure it is now possible to allocate memory (which has to be big enough to hold any rotated image dimension) once outside of the main loop.

FIFOs

When processing the image sequence frame by frame, less intermediade results must be stored in memory. However, the instantaneous frequencies kernel and the component velocities kernel need input data of more than one frame to process one output frame. For these two kernels, FIFO-like buffers for the input data are introduced.

Figure 5.6 shows how the filter responses of three frames are stored in a FIFO (there is one such FIFO per direction). The output of the directional filter always overwrites the oldest frame in the FIFO. Therefore, the frames in the FIFO are generally not in the correct order. But since the lenght of the FIFO is only 3 (fixed), the frames can be rearranged easily by passing three different pointers (one for the current frame, one for the previous frame and one for the next frame) to the instantaneous frequencies kernel.

The filter response of frame i is written to the position $(i \bmod 3)$ in the FIFO. As soon as there is enough data in the FIFO ($i = 2$), the current frame can be accessed at position $((i-1) \bmod 3)$, the previous frame at position $((i-2) \bmod 3)$, and the next frame at position $(i \bmod 3)$.

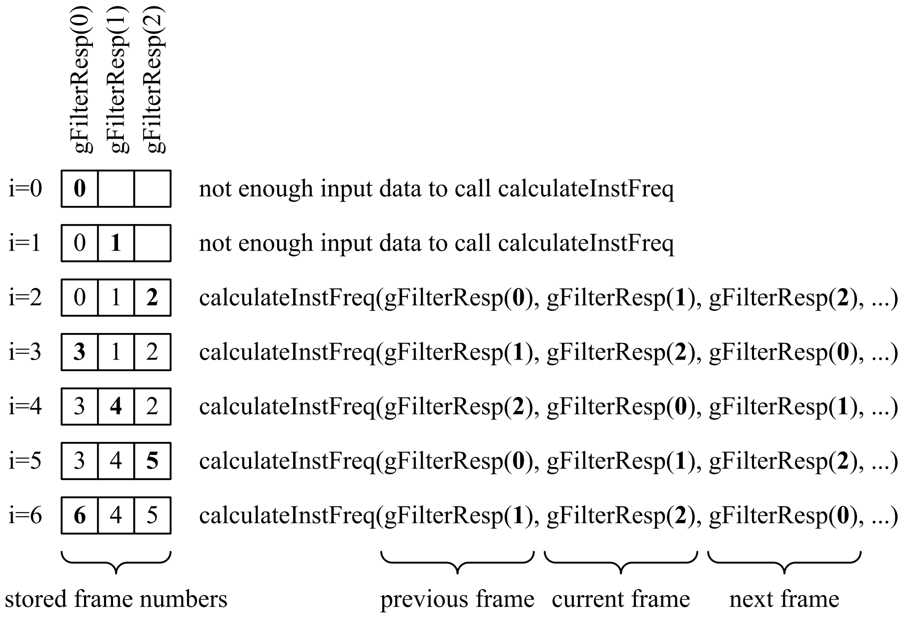
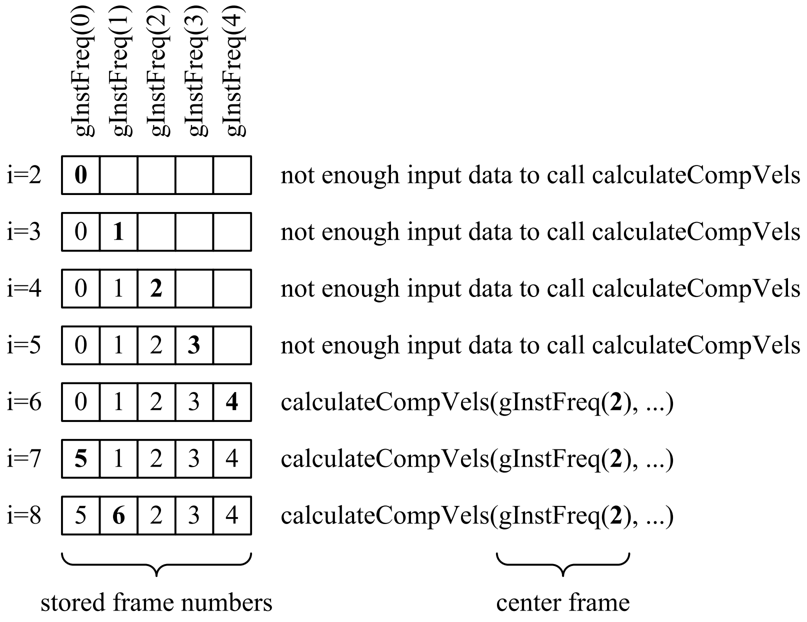

gFilterResp(0)
gFilterResp(1)
gFilterResp(2)
i=0 | 0 | | | not enough input data to call calculateInstFreq
i=1 | 0 | 1 | | not enough input data to call calculateInstFreq
i=2 | 0 | 1 | 2 | calculateInstFreq(gFilterResp(0), gFilterResp(1), gFilterResp(2), ...)
i=3 | 3 | 1 | 2 | calculateInstFreq(gFilterResp(1), gFilterResp(2), gFilterResp(0), ...)
i=4 | 3 | 4 | 2 | calculateInstFreq(gFilterResp(2), gFilterResp(0), gFilterResp(1), ...)
i=5 | 3 | 4 | 5 | calculateInstFreq(gFilterResp(0), gFilterResp(1), gFilterResp(2), ...)
i=6 | 6 | 4 | 5 | calculateInstFreq(gFilterResp(1), gFilterResp(2), gFilterResp(0), ...)
stored frame numbers
previous frame
current frame
next frame

Figure 5.6 FIFO to store the filter responses for 3 frames.

The number of input frames for the component velocities kernel depends on the `RadTemp` parameter, so the lenght of the FIFO to store the instantaneous frequencies is not fixed. The result of frame i is stored at position $((i-2) \bmod (2 \cdot \mathrm{RadTemp} + 1))$ in the FIFO. However, since the component velocity kernel is insensitive to the order of the input frames, rearranging of the frames is not necessary. Instead, just the pointer to the center frame can be passed to the kernel (see figure 5.7). The kernel then accesses the `RadTemp` previous and the `RadTemp` next frames in the FIFO to build the matrix (3.34).

Optical Flow Kernel

In the unoptimized implementation, the component velocities of all directions and for all frames were stored in buffers before they were combined to the resulting flow vectors. With the new frame-by-frame approach, the component



	gInstFreq(0)	gInstFreq(1)	gInstFreq(2)	gInstFreq(3)	gInstFreq(4)	
i=2	**0**					not enough input data to call calculateCompVels
i=3	0	**1**				not enough input data to call calculateCompVels
i=4	0	1	**2**			not enough input data to call calculateCompVels
i=5	0	1	2	**3**		not enough input data to call calculateCompVels
i=6	0	1	2	3	**4**	calculateCompVels(gInstFreq(**2**), ...)
i=7	**5**	1	2	3	4	calculateCompVels(gInstFreq(**2**), ...)
i=8	5	**6**	2	3	4	calculateCompVels(gInstFreq(**2**), ...)

Figure 5.7 FIFO to store the instantaneous frequencies for `RadTemp=2`.

velocities (for all directions) of only one frame must be stored. This reduces the memory consumption and it also makes passing the data to the kernel easier (the array of pointers in constant GPU memory is no longer needed).

Performance

Table 5.15 shows a speed up of factor 2 for the optical flow kernel and a huge absolute speedup for the non-kernel code. Table 5.16 illustrates the improvements in the memory consumption. It shows the number of input frames that can be computed without an *out of memory* error for various image dimensions and parameters.

Kernel	**%**	**Time [ms]**	**Calls**
Total	100.0	96.65	1
Total Kernels	59.0	57.06	481
Component Velocities	24.3	23.50	68
Transversal Filtering	15.3	14.83	84
Instantaneous Frequencies	7.8	7.56	76
Longitudinal Filtering	7.3	7.02	84
Rotating	2.0	1.90	84
Back-Rotating	1.6	1.51	68
Optical Flow	0.8	**0.74**	17

Table 5.15 Timing: optimized program flow (SW rev. 96).

Image Dimensions	`NumDir`	`RadTemp`	Unoptimized (SW rev. 74)	Optimized (SW rev. 96)
100 × 200	4	1	388	2344
100 × 200	8	1	262	2299
100 × 200	4	3	388	2314
500 × 1000	4	1	23	70
500 × 1000	8	1	18	32
500 × 1000	4	3	25	45

Table 5.16 Maximum number of input frames for unoptimized and optimized implementation (`Pads=15`).

5.4.7 Speedup

Introduction

Table 5.17 shows the speedup of the optimized implementation compared to a Matlab implementation for some test sequences (see appendix C).

Test Sequence	Matlab [min]	CUDA [s]	Speedup
Taxi	10.3	0.124 (0.087)	4977
Rubik Cube	12.8	0.141 (0.097)	5434
Ettlinger Tor	343.2	1.372 (0.785)	15009

Table 5.17 Basic Algorithm speedup (SW rev. 96).

When processing the Ettlinger Tor sequence (512 × 512 pixels), Matlab seems to need so much memory, that Windows has to use the swap file extensively, which makes the computation very slow.

Measurement

The speedup was computed with the execution times reported by the Matlab GUI. For the CUDA implementation, these values also include the time Matlab needs to set up the algorithm and calling the *.mex* file. The times in brackets are the total execution times returned by the *.mex* file. They correspond to the total time in the tables above.

5.5 Pyramid Algorithm

Introduction

This section describes the CUDA implementation of the Pyramid Algorithm explained in chapter 4.

5.5.1 Using the MEX File

Introduction

The output parameters and most of the input parameters are the same as for the Basic Algorithm (see section 5.4.1). In this section, only the usage of the additional input parameters is described.

```
[Flow Cert Time] = AmFmPyrCUDA(ImgSeq, Pads, NumDir, AngleOffset,
   RadSpat, RadTemp, CompVelThr, OptFlowThr, SmoothCoefs,
   MaxLevel, TransCoefs, LongCoefs)
```

`SmoothCoefs`

This parameter is used to pass the coefficients of the scaling-filter (smoothing filter) to the *.mex* file. The *.mex* file supports separable and non-separable two-dimensional filters. When passing a row vector of length N, the separable case is assumed and the coefficients are used for both the row filter and the column filter. The non-separable case is applied when passing an array of $N \times N$ coefficients. In both cases, N must be odd and not smaller than 3. The expected data type for the coefficients is `single` and the maximum number of coefficients is $17^2 = 289$.

`MaxLevel`

This parameter defines the maximum level for the pyramid decomposition. The velocities are estimated on (MaxLevel + 1) levels. When `MaxLevel` is 0, no downsampling and no compensation is performed (but, in contrast to the Basic Algorithm, the scale-space filter is applied before estimating the velocities).

For test purposes, a negative `MaxLevel` argument can be passed to the *.mex* file. In this case, only the velocities on the level (−MaxLevel) are estimated. The results are then upsampled to the original image size and returned (no integration of the pyramid levels is performed). Note that the confidence values of the component velocities are not passed to lower levels and therefore the computation of the final flow vectors does not work unless `NumDir` is 0 or `OptFlowThr` is less than zero. For the same reason, this test mode can not be used from the Matlab GUI.

`TransCoefs`, `LongCoefs`

These parameters can be used to pass custom filter coefficients for the directional filter to the *.mex* file.

For the transversal part (`TransCoefs`), up to 55 real-valued (`single`) coefficients can be passed as a row vector.

For the longitudinal part (`LongCoefs`), up to 15 complex-valued (`single`) coefficients can be passed as a row vector.

Passing `0` to one of these parameters causes the mex function to use the standard coefficients for the respective filter part.

5.5.2 Implementation Notes

Introduction

This section basically describes the differences between this pyramid implementation and the implementation of the Basic Algorithm.

Scale-Space Kernels

Several new kernels for the scale-space filtering and the downsampling were implemented (see table 5.18). These five kernels are optimized using texture memory (see also section 5.4.4).

	Downsampling + Scale-Space Filtering	Scale-Space Filtering Only (for Level 0)
Separable	`separableRowFilterDS_Kernel` `separableColFilter_Kernel`	`separableRowFilter_Kernel` `separableColFilter_Kernel`
Non-Separable	`nonseparableFilterDS_Kernel`	`nonseparableFilter_Kernel`

Table 5.18 Scale-space filtering kernels.

Another kernel was implemented for the upsampling of the compensation values. These values are signed integers, and no interpolation is performed. The upsampling is done directly on the rotated data (not back-rotating, upsampling, rotating).

Compensation

To implement the compensation described in section 4.3, a rhomboid-shaped region of the directional filter response is used. This applies to the computation of the instantaneous frequencies as well as to the computation of the component velocities. The compensation can be different for every single component velocity value. Therefore, the instantaneous frequencies can no longer be precomputed for a whole frame, as it was done in the Basic Algorithm (figure 5.5).

The new implementation computes the instantaneous frequencies for the local ST-slice of the current component velocity directly in the component velocity kernel. Therefore, a separate instantaneous frequency kernel is no longer used.

Program Structure

Figure 5.8 shows the structure of the main function of the Pyramid Algorithm. There are three differences compared to the structure of the Basic Algorithm (figure 5.5):

- Each frame is first filtered and downsampled to get its representations for all the pyramid levels.
- Before computing the component velocities, the `getCompensation` function is called to get the compensation values from the higher level(s).
- The instantaneous frequencies are no longer computed separately.

The computation of the compensation values is recursive: the `getCompensation` function calls itself to get the compensation from the next higher level.

Component Velocity Kernel

The component velocity kernel also computes the compensation values (the rounding and scaling shown in figure 4.5 is done inside the kernel). The kernel takes the compensation values from the next higher level as inputs and provides the component velocities and the new compensation values as outputs. Depending on which of these pointers are provided, the kernel behaves differently:

- On the highest level, a null-pointer is passed for the compensation input. In this case, the kernel uses zero compensation for every pixel.
- When a valid pointer is provided for the compensation output (and a null-pointer for the component velocity output), the compensation values are computed according to (4.6). This applies to all levels except for level 0.
- On level 0, a valid component velocity pointer and a null-pointer for the compensation output is passed to the kernel. In this case, the compensation values from the higher levels are added to the estimated component velocities, which result in the final component velocities (see equation 4.3).

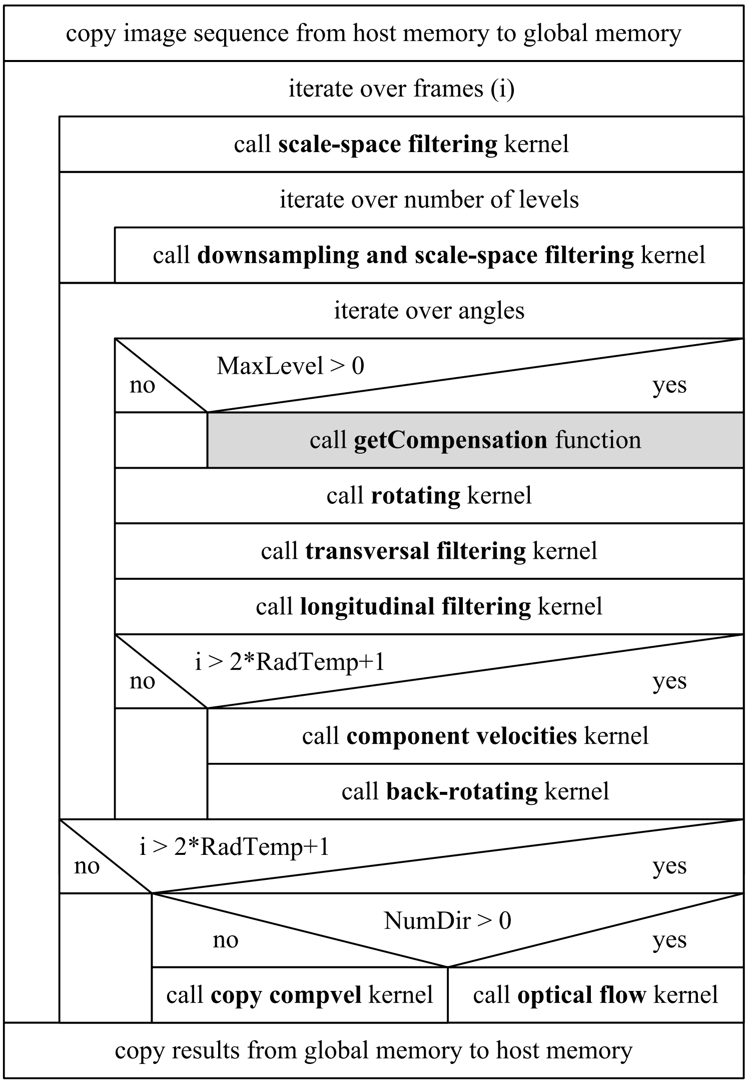

copy image sequence from host memory to global memory
iterate over frames (i)
call **scale-space filtering** kernel
iterate over number of levels
call **downsampling and scale-space filtering** kernel
iterate over angles
MaxLevel > 0
no
yes
call **getCompensation** function
call **rotating** kernel
call **transversal filtering** kernel
call **longitudinal filtering** kernel
i > 2*RadTemp+1
no
yes
call **component velocities** kernel
call **back-rotating** kernel
i > 2*RadTemp+1
no
yes
NumDir > 0
no
yes
call **copy compvel** kernel
call **optical flow** kernel
copy results from global memory to host memory

Figure 5.8 Structogram: Pyramid Algorithm, `mexFunction`.

Directional Filter

There are some small differences and enhancements compared to the Basic Algorithm which are not caused by the multiresolution scheme. One difference is that the transversal part of the directional filter is optimized using texture memory instead of shared memory (see also section 5.4.4). This is because the texture memory approach is more flexible with respect to the filter lenght, and therefore more suitable to handle custom directional filters.

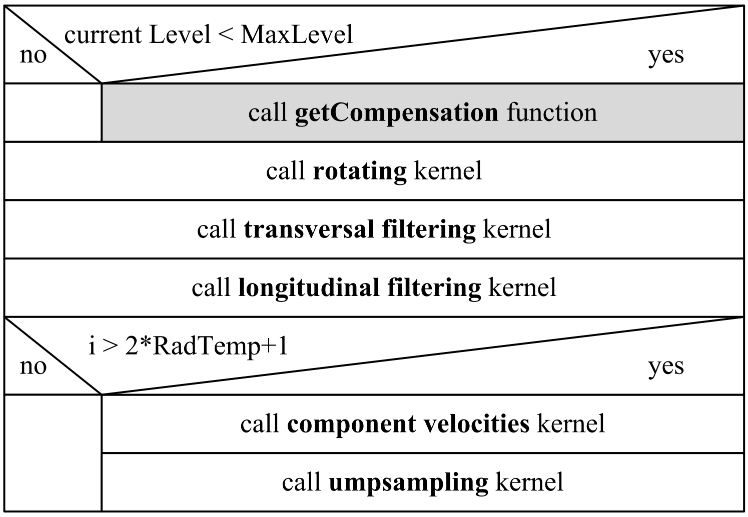

current Level < MaxLevel
no
yes
call **getCompensation** function
call **rotating** kernel
call **transversal filtering** kernel
call **longitudinal filtering** kernel
i > 2*RadTemp+1
no
yes
call **component velocities** kernel
call **umpsampling** kernel

Figure 5.9 Structogram: Pyramid Algorithm, `getCompensation`.

5.5.3 Performance

One Level

Due to the modifications described in section 5.5.2, the Pyramid Algorithm is more computational complex than the Basic Algorithm, even when only one pyramid level is computed (MaxLevel=0). Table 5.19 shows the timing for the Rubik Cube sequence. The results can be compared with table 5.15 in section 5.4.6. The same parameters were used as for the tests in section 5.4. Additionally, a 13 × 13 non-separable Pauwels filter and no custom directional filter is passed to the *.mex* file.

Note that the results in section 5.5 for the pyramid implementation were computed using a newer version of Parallel Nsight than the results in section 5.4.

Kernel	%	Time [ms]	Calls
Total	100.0	256.93	1
Total Kernels	84.0	215.93	426
Component Velocities	70.3	**180.55**	68
Transversal Filtering	6.4	**16.46**	84
Longitudinal Filtering	2.9	7.49	84
Scale-Space Non-Separable	2.8	**7.28**	21
Rotating	0.7	1.88	84
Back-Rotating	0.6	1.52	68
Optical Flow	0.3	0.74	17

Table 5.19 Timing: pyramid with MaxLevel=0 (SW rev. 138).

The most noticeable slow-down occured in the new component velocity kernel (180.55 *ms*), which replaces the old component velocity kernel (23.5 *ms*) and the instantaneous frequency kernel (7.56 *ms*) of the Basic Algorithm.

This can be explained by the fact that now the instantaneous frequencies have to be computed again for each local ST-slice and can no longer be pre-computed for the whole image (see section 5.5.2). With the parameters used for table 5.19 (RadTemp=1, RadSpat=3), 21 instantaneous frequencies have to be computed for every component velocity. For the Basic Algorithm, only one instantaneous frequency is computed for every component velocity. A rough calculation with the timings of the Basic Algorithm from table 5.15 shows, that the new execution time of the component velocity kernel is reasonable:

$$21 \cdot 7.56\,ms + 23.5\,ms = 182.26\,ms \quad \approx \quad 180.55\,ms \tag{5.2}$$

Another considerable amount of additional computation effort comes from the new scale-space filtering kernel. The use uf texture memory instead of shared memory for the transversal filter kernel has no big influence, and the time spent in non-kernel code is almost the same as in the Basic Algorithm (41 *ms* vs. 39.59 *ms*).

Optimization Potential The speed of the pyramid implementation could be improved by handling the highest level separately. In this case, the number of instantaneous frequency computations in the component velocity kernel could be reduced since there is no compensation on the highest level. Furthermore, the new component velocity kernel performs a large number of global memory accesses to read the directional filter response. By reducing the number of reads from global memory (using shared memory or texture memory), it should be possible to reduce the computation time also in the case where compensation is necessary.

Number of Levels Figure 5.10 shows how the execution time of the mex function increases with the number of levels for two different image sequences. As expected (due to the downsampling), the execution time grows asymptotically. The values shown are the total execution times returend by the *.mex* file (average of five values) computed using the default parameters and with a 13 × 13 Pauwels scale-space filter.

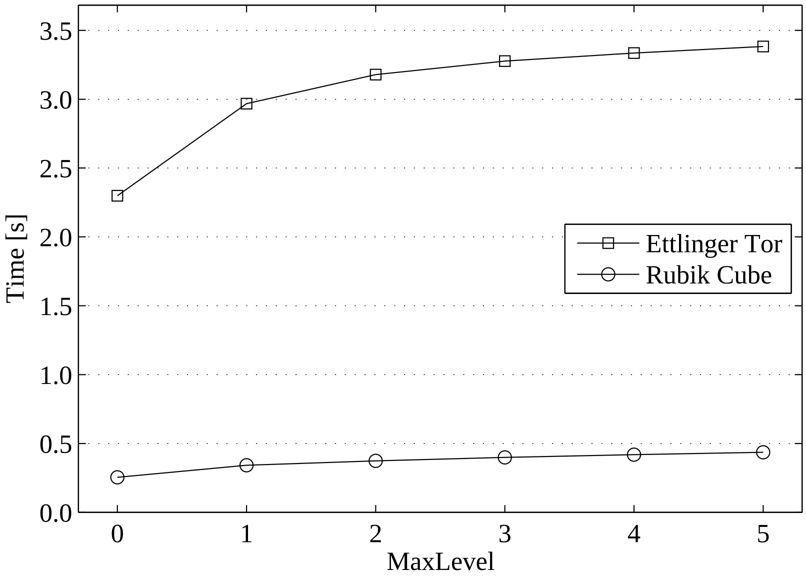


Figure 5.10 Execution time vs. MaxLevel (SW rev. 138).

Scale-Space Filter

Table 5.20 compares the execution times for different scale-space filters when implemented as a separable or as a non-separable filter. A Gaussian kernel was used in both cases.

The non-separable implementation (of the same filter) needs more time than the separable implementation. Since the order of complexity is $O(r^2)$ for the non-separable implementation but only $O(r)$ for the separable implementation (where r is the filter radius, assuming a fixed image size), the difference becomes more considerable for large filters.

The table shows the mean value of three measurements with the default parameters and with MaxLevel=0.

Gaussian	Separable		Non-Separable		Difference	
Filter Size	Kernel [ms]	Total [ms]	Kernel [ms]	Total [ms]	Kernel [ms]	Total [ms]
9×9	1.37	249.4	3.82	253.6	2.45	4.2
13×13	1.69	251.7	7.28	259.0	5.59	7.3
17×17	1.93	252.3	11.71	266.7	9.78	14.4

Table 5.20 Execution times for separable and non-separable scale-space filters (SW rev. 139).

6 Results

Introduction In this chapter, some selected results are presented. For the sake of clarity, only the Fleets angular error (average of one frame) is stated here. More details can be found in appendix A. Furthermore, a *.mat* file with all the information (parameters, standard deviations, densities, and other measuremets) can be found on the DVD for every data point in these two chapters.

Image Sequences Nine image sequences were used for the evaluation of the algorithms: The Diverging and Translating Tree sequences, the famous Yosemite sequence (without clouds), and six sequences with higher speeds from Middlebury College (see appendix C).

6.1 Number of Pyramid Levels

Introduction This section analyzes the performance of the multiresolution implementation of the algorithm by looking at the influence of the number of pyramid levels. The number of pyramid levels is determined by the MaxLevel parameter. A MaxLevel setting of N means that the original image sequence is downsampled N times, or that the component velocities are estimated on $N + 1$ levels.

Results The optimum MaxLevel parameter for the nine test sequences can be found in the rightmost column of table 6.1 (see section A.1 for more details). The table also shows the maximum speed (of the ground truth) in the evaluated frame. Some of the sequences have their maximum speed in the border regions, which was skipped for the accuracy measurement. The third column of table 6.1 shows the maximum speed within the evaluated region.

Postprocessing The parameter for the outlier removal in the postprocessing (maximum speed) was set according to the MaxLevel parameter: $1.5 \cdot 2^{MaxLevel+1}$

The fourth column in table 6.1 shows the expected optimum MaxLevel parameter according to the a priori knowledge of the maximum speed.

Image Sequence	**Maximum Speed (Full Image)**	**Maximum Speed (Skipped Border)**	**MaxLevel (Theoretical)**	**MaxLevel (Best)**
Diverging Tree	1.97	1.53	0	1
Translating Tree	2.25	2.20	0	0
Rubber Whale	4.62	4.62	1	0
Yosemite Cloudless	5.35	4.81	1	1
Grove 2	5.03	5.03	1	1
Hydrangea	11.12	11.12	2	1
Urban 3	17.61	17.49	3	3
Grove 3	18.61	18.38	3	2
Urban 2	22.19	22.19	3	4

Table 6.1 Maximum speeds and optimum number of levels.

Conclusion This evaluation shows that the pyramid scheme described in chapter 4 works fine for image sequences with high speeds. However, using too many pyramid levels impairs the accuracy of the motion estimation (see plots in section A.1).

Degrading Performance Figure 6.1 shows the error plots for the Translating Tree sequence for the optimum number of levels (top) and for the case with too many levels (bottom). The smoothing of the images when going from one level to the next makes it more and more difficult to estimate the component velocities reliably on higher levels, since there is less and less structure in the images (depending on the spatial frequency content of the images). The errors of higher levels are then propagated to level 0, leading to a decreased performance.

Figure 6.2 suggests a further reason for the decreasing performance: The spatial support of the directional filters grows rapidly (in relation to the original image resolution) when going to higher levels. For complex image sequences with non-uniform motion (e.g. Grove 3 sequence where small objects are moving with a different velocity than their surrounding), the estimation on high levels might be too bulky.

Improvement Potential In situations like the one in figure 6.1, it could be helpful to apply additional postprocessing steps on the estimates of higher levels, before they are upsampled and used on the next lower level. The goal of these steps would be to remove outliers due to missing structure in the downsampled image. One could think about using median filters combined with a clever thresholding based on the certainties of the estimates.

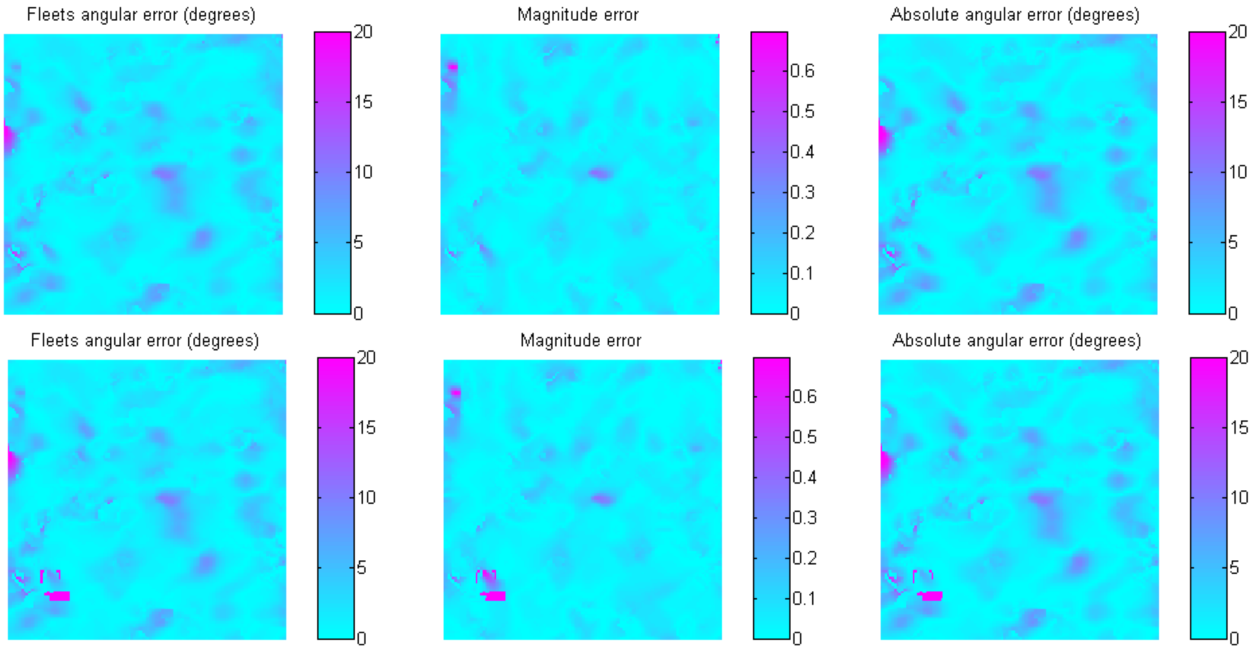


Figure 6.1 Error plots for the Translating Tree sequence, MaxLevel=1 (top) and MaxLevel=2 (bottom).

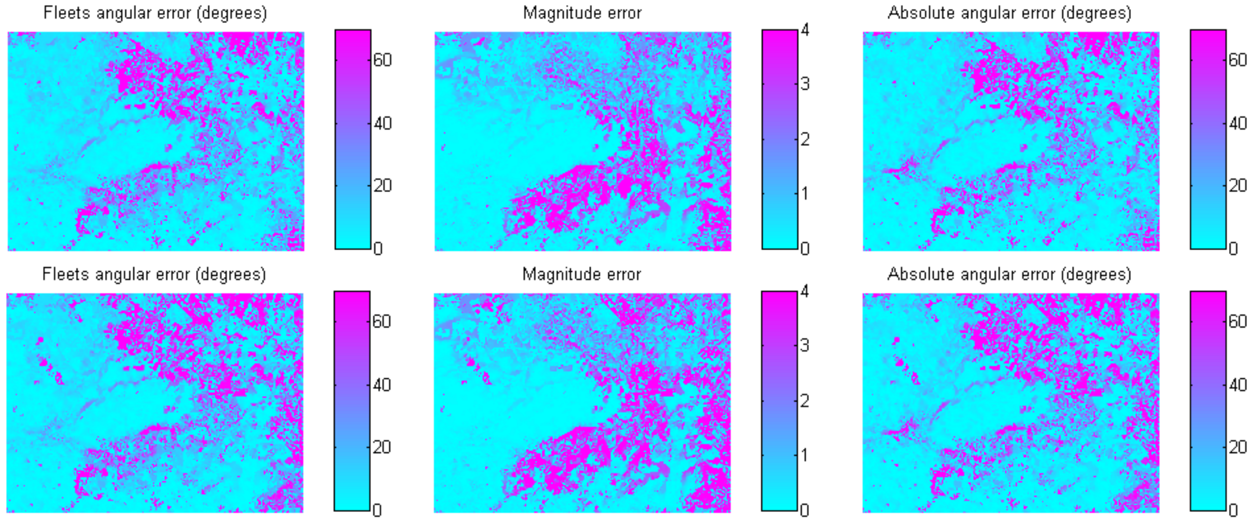


Figure 6.2 Error plots for the Grove 3 sequence, MaxLevel=2 (top) and MaxLevel=4 (bottom).

6.2 Smoothing Filter

Introduction

This section compares the performance of different smoothing filters. Particularly, we want to check if it is worthwile to use a special Pauwels filter (non-separable) instead of a separable Gaussian filter.

Parameters

The MaxLevel parameter was set according to the optimum values found in the previous section.

Conclusion

As can be seen from table 6.2, there is no optimum smoothing filter for all sequences. For some low-speed sequences it is even advantageous to use no smoothing filter at all (select Dummy-Filter in the Matlab GUI). In these cases, the loss of texture due to the smoothing is worse than possible aliasing.

The influence of the σ parameter for the Gaussian filter was not examined.

Test Sequence		**None**	**Gaussian (σ = 1.5)**			**Pauwels**	
Name	**MaxLevel**	**–**	**9 × 9**	**13 × 13**	**17 × 17**	**13 × 13**	**17 × 17**
Diverging Tree	1	**2.394**	2.742	2.744	2.745	2.509	2.513
Translating Tree	0	**1.561**	1.970	1.967	1.967	1.798	1.788
Rubber Whale	0	**9.957**	9.996	10.010	10.010	10.192	10.220
Yosemite Cloudless	1	9.318	**8.079**	8.263	8.456	8.115	8.239
Grove 2	1	15.138	7.705	7.702	7.701	**7.391**	7.451
Hydrangea	1	11.758	9.348	**9.347**	9.348	10.023	10.072
Urban 3	3	26.501	19.805	19.868	19.869	19.669	**19.658**
Grove 3	2	20.987	**15.110**	15.197	15.196	15.763	15.788
Urban 2	4	50.759	35.890	35.891	**35.885**	36.128	36.410

Table 6.2 Comparison of different smoothing filters (Fleets angular error after postprocessing).

6.3 Other Parameters

Number of Directions Basically, the accuracy increases with the number of directions. At least 3 or 4 directions are needed for good results. Using 7 directions seems to be a good choice for most of the sequences, so this setting was used for the following evaluations. The plots showing the accuracy vs. NumDir can be found in section A.2.

Temporal Radius and Spatial Radius The temporal and spatial radius parameters define the size of the local ST-slice from which the structural tensor is built. First, the optimum spatial radius for every sequence was identified using the parameters values found above. Then, using the optimum spatial radius, the influence of the temporal radius was examined. Note that the Middlebury sequences consist of only 8 frames, which does not allow a temporal radius of more than 2.

The results in sections A.4 and A.3 show that for sequences with a uniform motion field (e.g. Translating Tree), larger radii are advantageous. In all other cases, the accuracy degrades from a certain size of the local ST-slice, depending on the size of the structures in the image.

Note that the computational complexity increases considerably with the area of the local ST-slice. Not only because the structural tensor has to be built from more values, but mainly because with the Pyramid Algorithm, $(2\cdot \mathit{RadTemp}+1)(2\cdot \mathit{RadSpat}+1)$ instantaneous frequencies have to be computed for every component velocity (see section 5.5.3).

6.4 Comparison with other Algorithms

Introduction In this section, the Basic Algorithm and the Pyramid Algorithm are compared with the following algorithms:

- Lucas-Kanade Extended (CUDA implementation of Marc Länzlinger and Fabian Braun [3], non-pyramid)
- Lucas-Kanade (OpenCV, non-pyramid)
- Lucas-Kanade Pyramid (OpenCV)
- Horn-Schunck (OpenCV, non-pyramid)

Other Algorithms Lucas-Kanade and Horn-Schunck are two well known optical flow estimation algorithms, but in the last 20 years, many other (more powerful) algorithms have been proposed.

Middlebury Sequences and Online Evaluation Baker et al. [12] developed a system for evaluating optical flow algorithms based on eight test sequences. The results of many algorithms for these sequences can be compared online[1]. However, the ground truth for these sequences is not publicly available, which makes it impossible to compare our results with the published ones.

On their webpage, they provide a second set of test sequences, for which the ground truth is available (the Rubber Whale, Hydrangea, Grove, and Urban sequences used in this chapter). However, no results are published for these sequences.

[1] http://vision.middlebury.edu/flow/

It would be possible to add the results of our algorithm to the online evaluation by submitting the estimated flow vectors. But one known problem of our algorithm is the poor behaviour in the border regions, due to the large spatial support of the directional filters. This problem was ignored during this project by skipping the border regions for the accuracy computations. When submitting flow vectors to the online evaluation tool, there is no possibility to mask out certain regions or to provide certainty indications for the flow vectors. Therefore it was decided to not use this evaluation tool to compare our algorithm.

Results

The results of the comparison are summarized in table 6.3. It shows the Fleets angular error for the best parameter set of every algorithm. In case of the Lucas-Kanade Extended algorithm, the table always shows the result of parameter set 4. The other parameter sets of this algorithm result in better accuracy measures but with a very poor density. Detailed results and the parameters used for comparison can be found in section A.5.

	Pyramid Algorithm	**Basic Algorithm**	**Lucas-Kanade Extended**	**Lucas-Kanade**	**Lucas-Kanade Pyramid**	**Horn-Schunck**
Sequence	CUDA	CUDA	CUDA	OpenCV	OpenCV	OpenCV
Diverging Tree	**1.322**	1.750	2.147	4.631	3.634	7.011
Translating Tree	0.581	0.874	0.823	11.310	**0.307**	14.508
Rubber Whale	**8.453**	9.173	13.022	17.339	13.249	18.051
Yosemite Cloudless	6.163	9.625	4.136	7.852	**3.996**	8.225
Grove 2	**5.224**	31.437	10.907	17.862	5.689	20.151
Hydrangea	**7.562**	84.220	23.635	24.473	8.336	25.317
Urban 3	16.105	73.862	35.990	44.164	**11.369**	44.969
Grove 3	12.556	46.217	26.547	29.798	**12.483**	30.994
Urban 2	28.559	52.872	54.494	49.804	**18.692**	48.815

Table 6.3 Comparison of the algorithms (Fleets angular error in °).

Conclusion

The Pyramid Algorithm performs better than the other algorithms on the Diverging Tree, Rubber Whale, Grove 2 and Hydrangea sequences. For the Grove 3 sequence, it comes very close to the Lucas-Kanade Pyramid algorithm. For all other sequences, the Lucas-Kanade Pyramid algorithm performs best. As expected, the non-pyramid algorithms show very poor results for sequences with high speeds.

Error Plots

For the sequences, where the Pyramid Algorithm gives worse results than the Lucas-Kanade Pyramid algorithm, the error plots can be found in section A.5.

Yosemite Cloudless

Lucas-Kanade Pyramid performs better on the Yosemite Cloudless sequence than the Pyramid Algorithm. The error plots show that both of these algorithms have problems handling the transition to the sky region, which has no texture. Due to the large directional filters, our Pyramid Algorithm has more difficulties to handle such motion discontinuities than the Lucas-Kanade algorithm.

By increasing the certainty threshold for the accuracy measurement from 51% to a value close to 100%, the erroneous flow vectors can be excluded. With a setting of 94%, the Fleets angular error can be reduced to 3.46°, but at the cost of a poor density of 40.1%.

One could also argue that the velocity in the sky region is undefined (instead of zero) and therefore mask this region out for the accuracy measurement. In this case, the two algorithms would probably give quite similar results.

Timing

Table 6.4 compares the execution times for the results in table 6.3. It shows the times reported by the Matlab GUI, which include the times Matlab needs to initialize the algorithms and call the *.mex* files. The postprocessing is handled separately by the Matlab GUI and is not included in these times. However, the execution time of the postprocessing should be approximately the same for all the algorithms, since they all use the same settings. The table shows the average of five timing measurements (for the entire sequences) taken on the Notebook (see table 5.2) with software revision 146.

The Lucas-Kanade Extended algorithm is less computational expensive than the algorithms presented in this report. However, Lucas-Kanade Extended is not a pyramid algorithm and therefore unable to handle the sequences with high speeds properly. But also for the sequences without high speeds, the results of the Lucas-Kanade Extended algorithm are worse (except for the Yosemite Cloudless sequence).

Note that the execution time of the Pyramid Algorithm is much higher than the one of the Basic Algorithm; especially for those sequences where a large local ST-slice is used (large RadTemp and RadSpat values). The speed of the Pyramid Algorithm could be improved considerably when the instantaneous frequencies were pre-computed for the whole frame before computing the component velocities (see section 5.5.3). This would especially improve the execution time for the Diverging and Translating Tree sequences, since they use a large local ST-slice and only one or two pyramid levels. For these sequences, it should be possible to achieve execution times which are comparable to the execution times of the Basic Algorithm.

As expected, the OpenCV implementation of Lucas-Kanade is slower than the CUDA implementation (Lucas-Kanade Extended). Comparing the execution times of the two most accurate algorithms, the Pyramid Alorithm and the Lucas-Kanade Pyramid algorithm, is quite pointless since they are implemented on different architectures. However, the comparison shows that the new efficient CUDA implementation makes the Pyramid Algorithm suitable for practical applications.

Sequence	**Pyramid Algorithm** CUDA	**Basic Algorithm** CUDA	**Lucas-Kanade Extended** CUDA	**Lucas-Kanade** OpenCV	**Lucas-Kanade Pyramid** OpenCV	**Horn-Schunck** OpenCV
Diverging Tree	12.27	0.90	0.11	1.26	13.84	1.99
Translating Tree	15.65	0.90	0.10	1.16	32.06	1.97
Rubber Whale	3.51	0.73	0.14	2.26	25.80	3.83
Yosemite Cloudless	4.52	0.76	0.11	1.54	15.68	2.58
Grove 2	7.15	0.91	0.18	3.08	87.89	5.26
Hydrangea	4.15	0.72	0.14	2.29	122.00	3.89
Urban 3	4.12	0.90	0.17	3.07	88.45	5.28
Grove 3	5.32	0.91	0.18	3.05	88.11	5.29
Urban 2	3.14	0.91	0.18	3.11	166.84	5.28

Table 6.4 Comparison of the computation times in s.

6.5 Conclusion

Accuracy

For most of the nine test sequences, the Pyramid Algorithm performs quite well compared to the OpenCV implementation of the Lucas-Kanade Pyramid algorithm. However, the problems of the Pyramid Algorithm in the border regions due to the large spatial support of the directional filters (and the size of the local ST-slice) were ignored by skipping the border region for the evaluation. However, this fundamental problem can also be seen at other motion discontinuities, particularly in the Yosemite Cloudless sequence, where the motion is undefined on one side of the motion discontinuity.

Timing

The computational complexity of the Pyramid Algorithm is much higher than for the Lucas-Kanade algorithm. However, the huge speedup by using the parallel computing power of a GPU makes the Pyramid Algorithm usable for practical applications.

Outlook

To further improve the performance of the Pyramid Algorithm, the two following issues should be studied in more detail:

- The poor results in the border regions. With this problem solved, the computed flow vectors could be submitted to Middlebury online evaluation tool to compare the results with many other algorithms.
- Unreliable estimates from higher levels. These estimates should be postprocessed with the goal that increasing the number of levels does not negatively affect the accuracy.

Furthermore, the computation time could be reduced considerably by implementing the improvements suggested in section 5.5.3.

A Additional Plots and Tables

A.1 Number of Pyramid Levels

Introduction

The plots in this section show the Fleets angular error before and after postprocessing (see table A.1 for the settings used) and the density of the flow vectors after postprocessing.

Parameter	**Setting**
Smoothing Filter Type	Pauwels (2D)
Smoothing Filter Size	13 × 13
Algorithm	Pyramid Algorithm (CUDA)
Padding	15
Number of Directions	4
Angle Offset	0
Spatial Radius	3
Temporal Radius	1
Component Velocity Threshold	0.8
Optical Flow Threshold	0.8
Maximum Pyramid Level	0:4 or 0:6
Postprocessing Maximum Magnitude	$1.5 \cdot 2^{MaxLevel+1}$ (absolute)
WVM Filter Window Size	3
WVM Filter Minimum Certainty	75%
WVM Filter Number of Passes	2
Accuracy Minimum Threshold	51%
Accuracy Border Skip	15

Table A.1 Parameters for the MaxLevel evaluation.

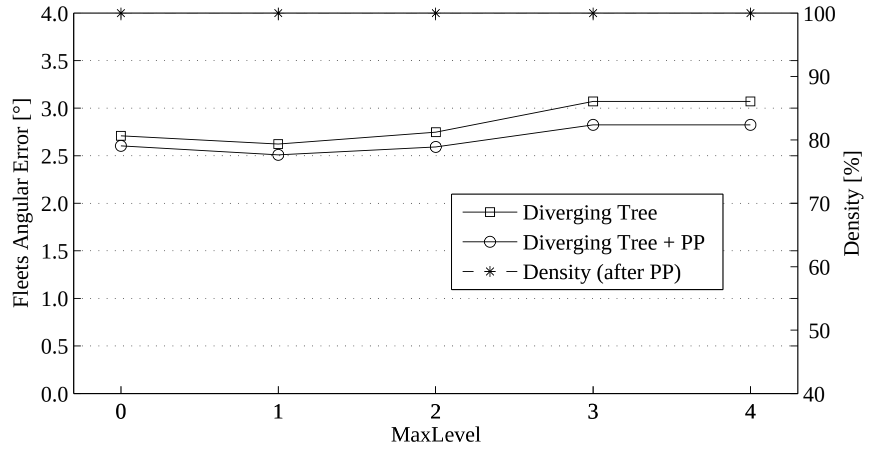


Figure A.1 Accuracy vs. MaxLevel for the Diverging Tree sequence (frame 19).

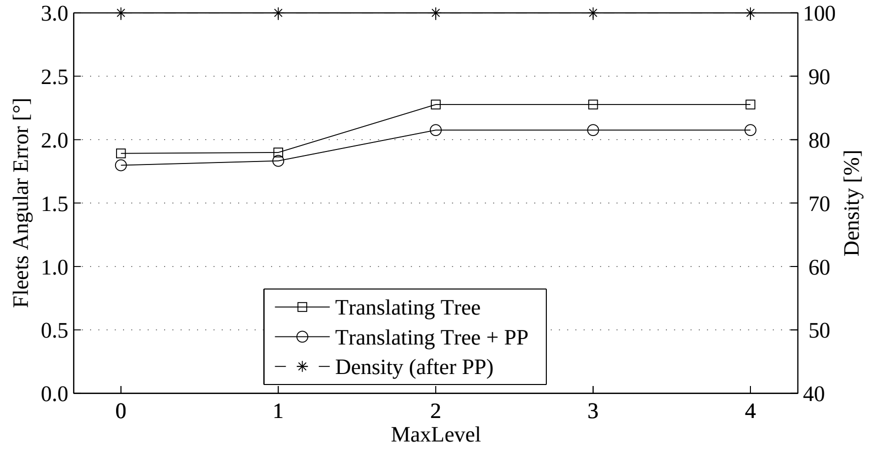


Figure A.2 Accuracy vs. MaxLevel for the Translating Tree sequence (fr. 19).

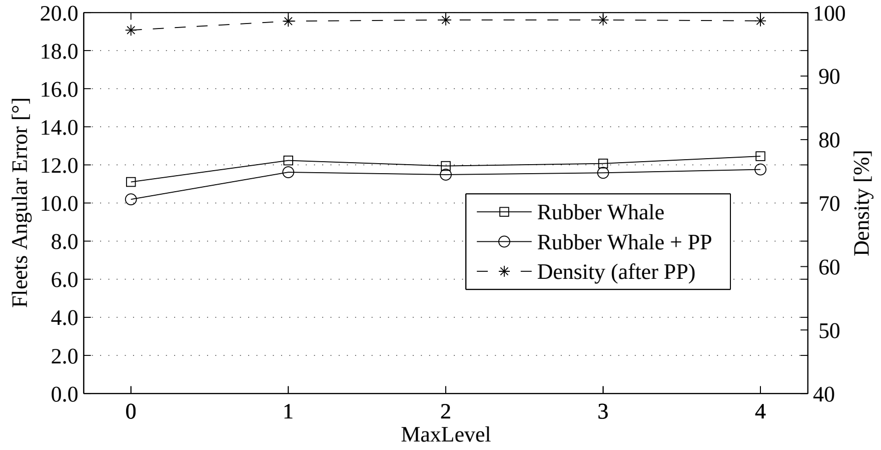


Figure A.3 Accuracy vs. MaxLevel for the Rubber Whale sequence (frame 4).

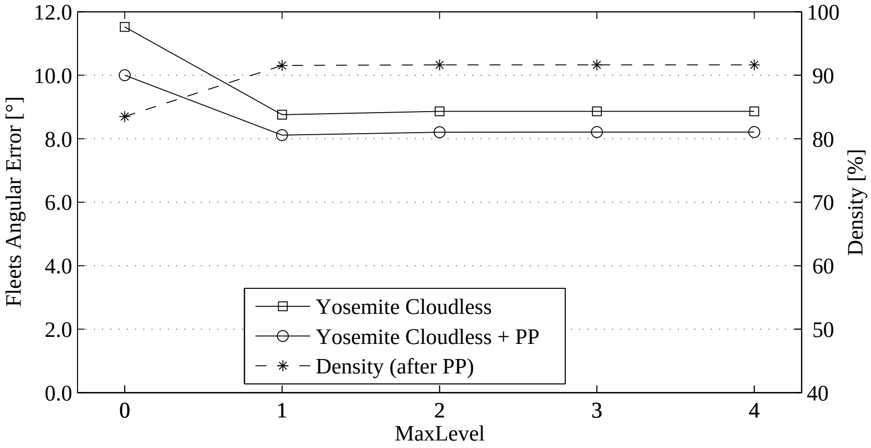


Figure A.4 Accuracy vs. MaxLevel for the Yosemite Cloudless sequence (fr. 7).

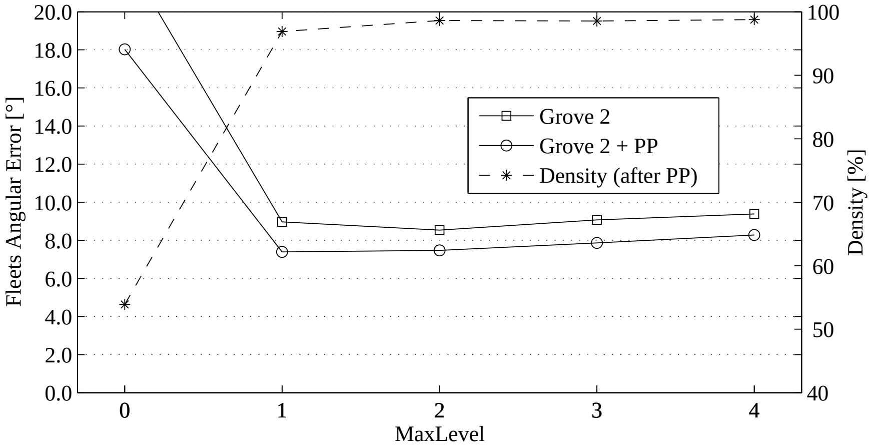


Figure A.5 Accuracy vs. MaxLevel for the Grove 2 sequence (frame 4).

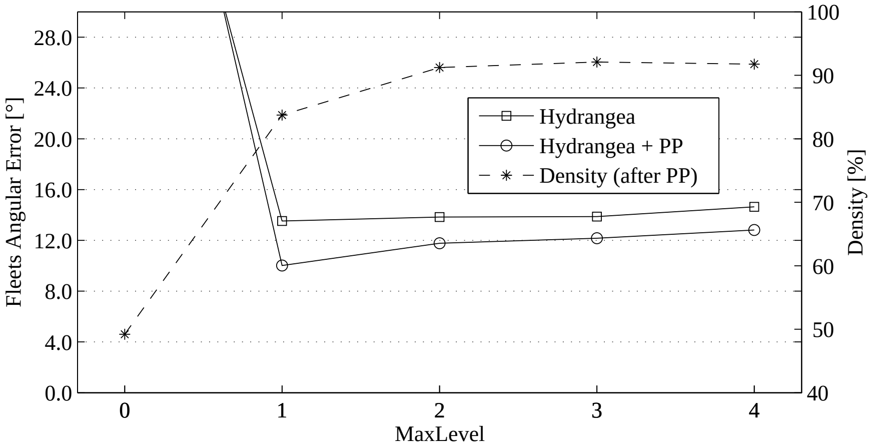


Figure A.6 Accuracy vs. MaxLevel for the Hydrangea sequence (frame 4).

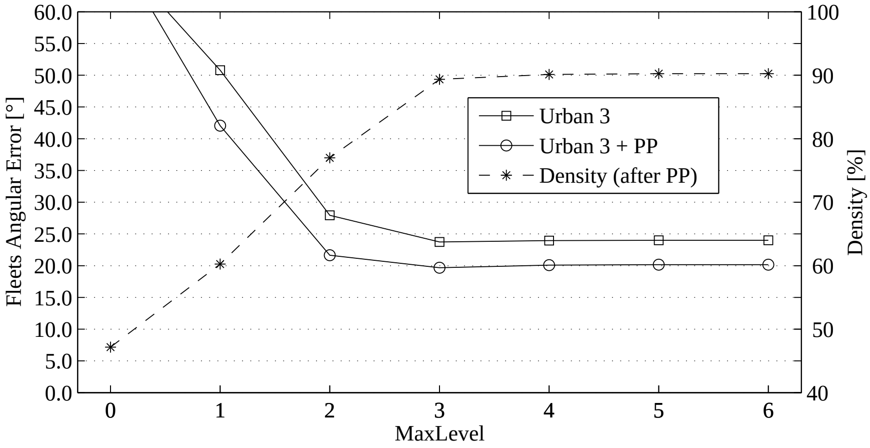


Figure A.7 Accuracy vs. MaxLevel for the Urban 3 sequence (frame 4).

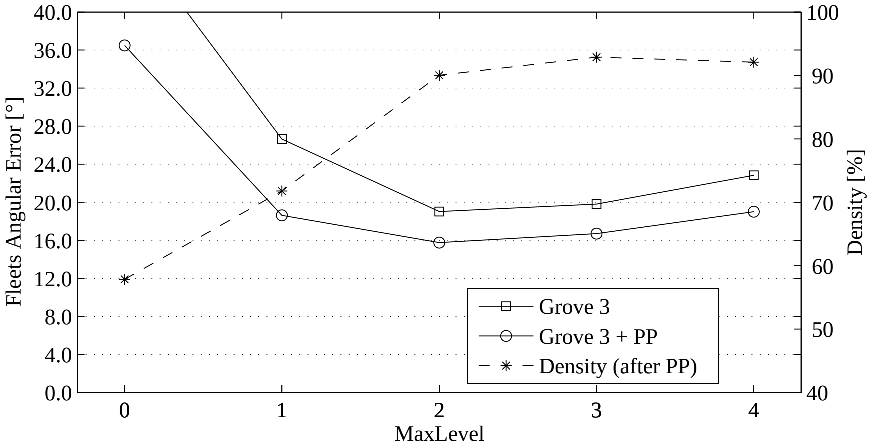


Figure A.8 Accuracy vs. MaxLevel for the Grove 3 sequence (frame 4).

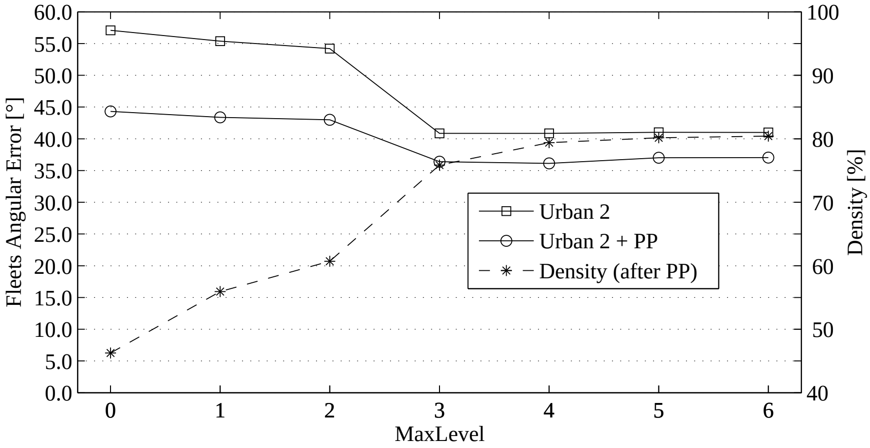


Figure A.9 Accuracy vs. MaxLevel for the Urban 2 sequence (frame 4).

A.2 Number of Directions

Description The plots on the following pages show the influence of the NumDir argument for every sequence. The parameters are the same as in table A.1, but with the optimum MaxLevel and smoothing filter parameters from table 6.2.

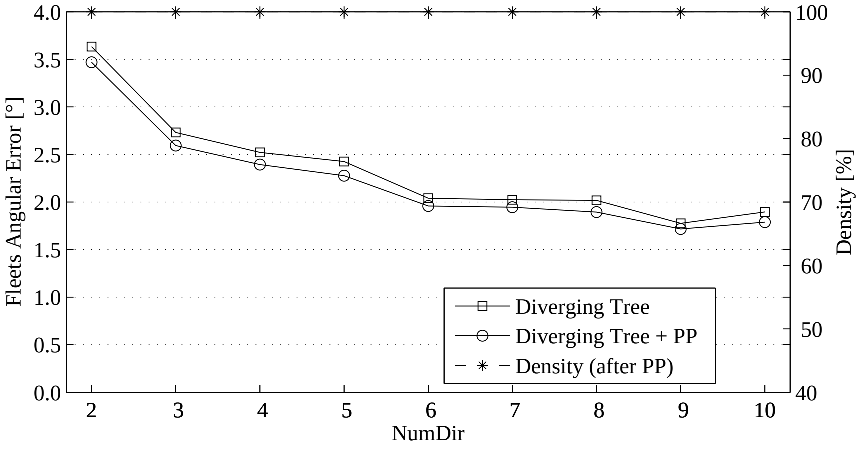


Figure A.10 Accuracy vs. NumDir for the Diverging Tree sequence (frame 19).

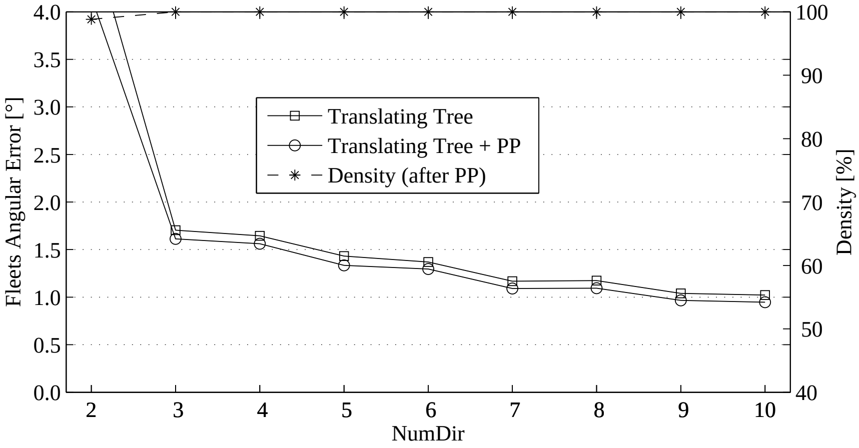


Figure A.11 Accuracy vs. NumDir for the Translating Tree sequence (fr. 19).

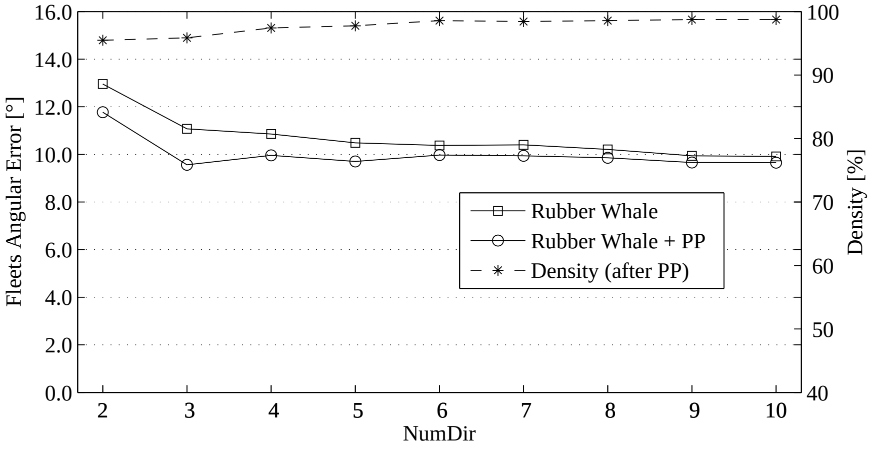


Figure A.12 Accuracy vs. NumDir for the Rubber Whale sequence (frame 4).

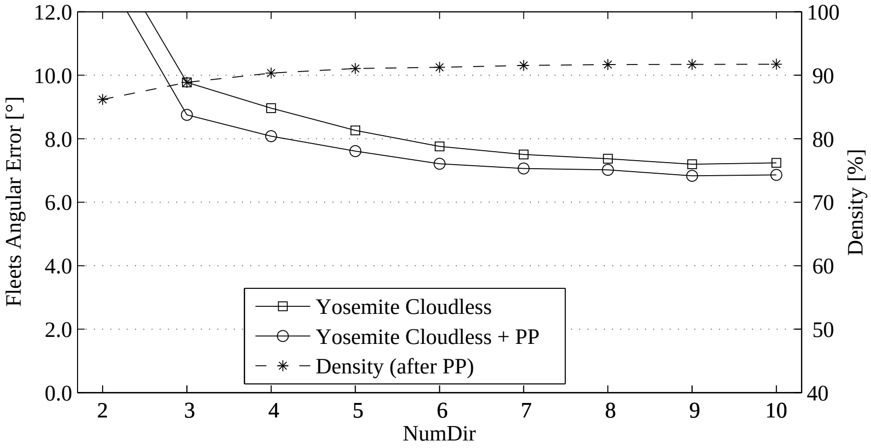


Figure A.13 Accuracy vs. NumDir for the Yosemite Cloudless sequence (fr. 7).

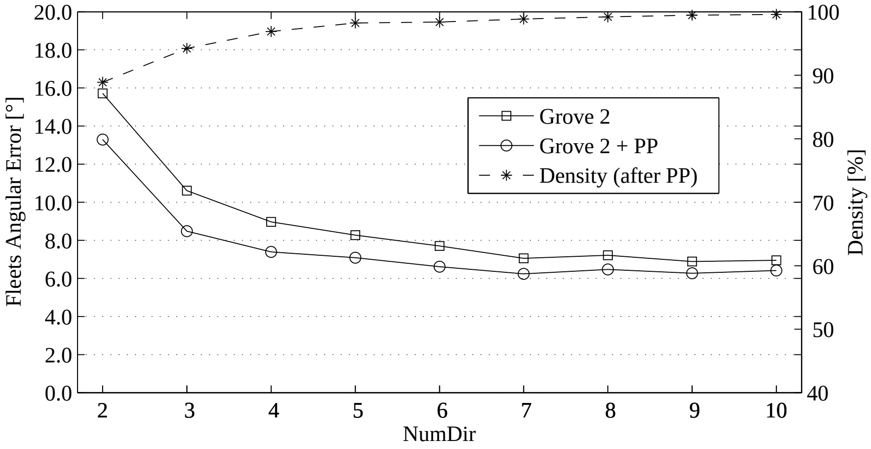


Figure A.14 Accuracy vs. NumDir for the Grove 2 sequence (frame 4).

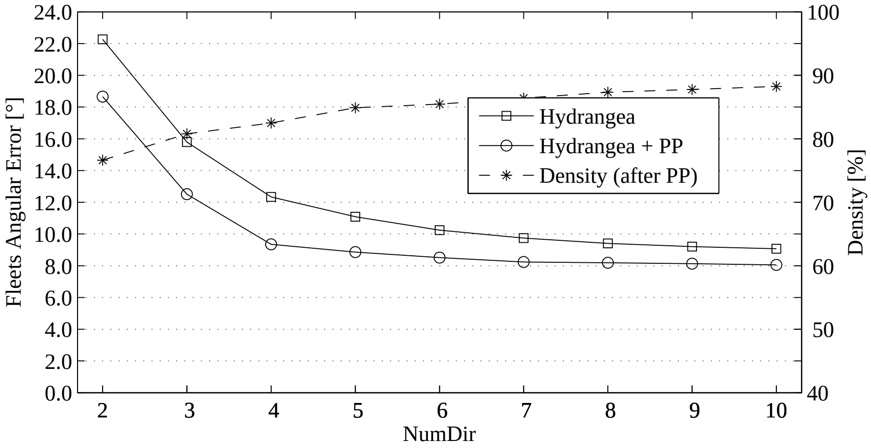


Figure A.15 Accuracy vs. NumDir for the Hydrangea sequence (frame 4).

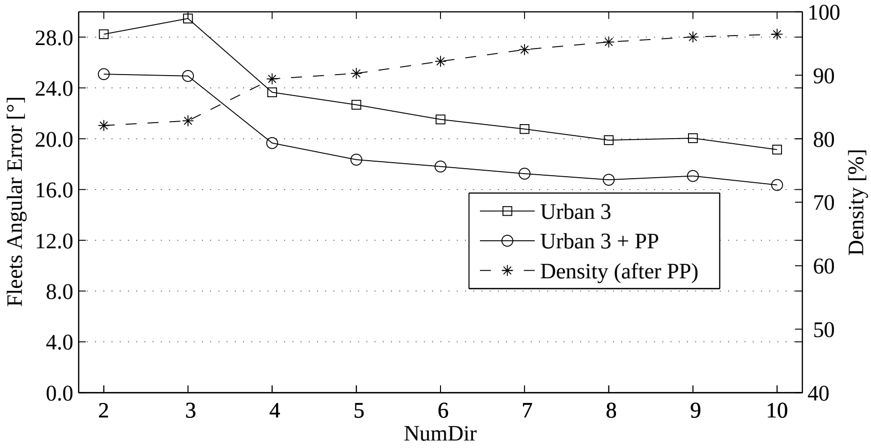


Figure A.16 Accuracy vs. NumDir for the Urban 3 sequence (frame 4).

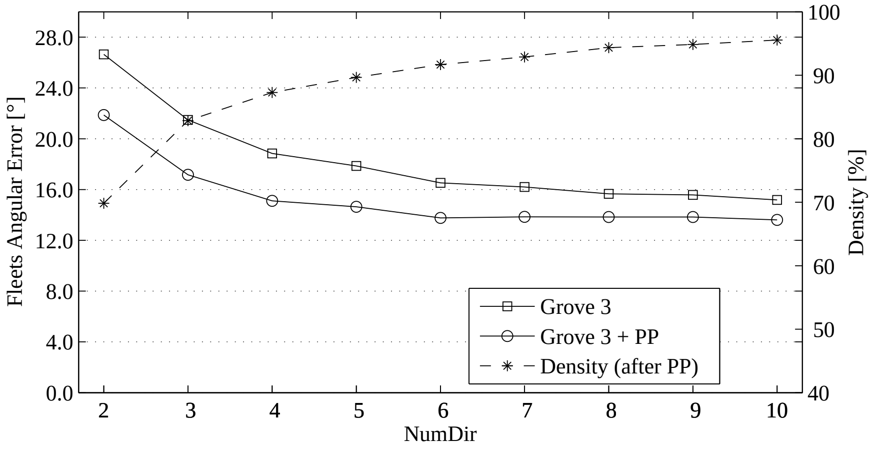


Figure A.17 Accuracy vs. NumDir for the Grove 3 sequence (frame 4).

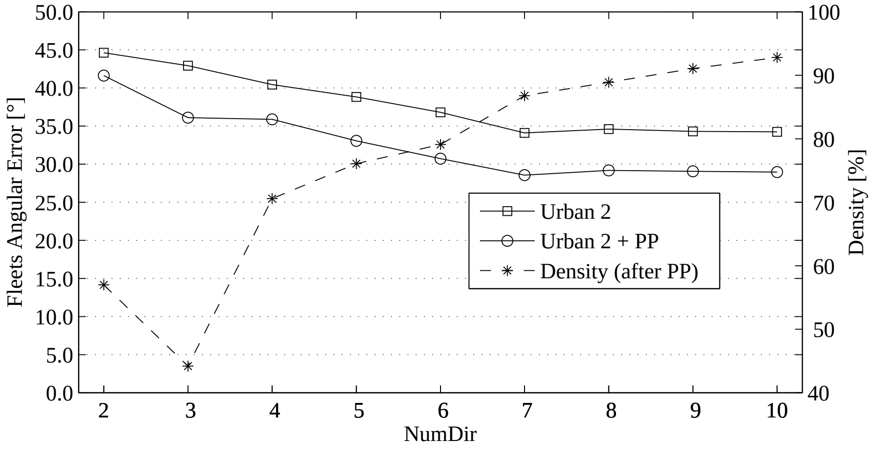


Figure A.18 Accuracy vs. NumDir for the Urban 2 sequence (frame 4).

A.3 Temporal Radius

Description

The plots on the following pages show the influence of the RadTemp argument for every sequence. The parameters are the same as in table A.1, but with the optimum MaxLevel and smoothing filter parameters from table 6.2 and NumDir=7.

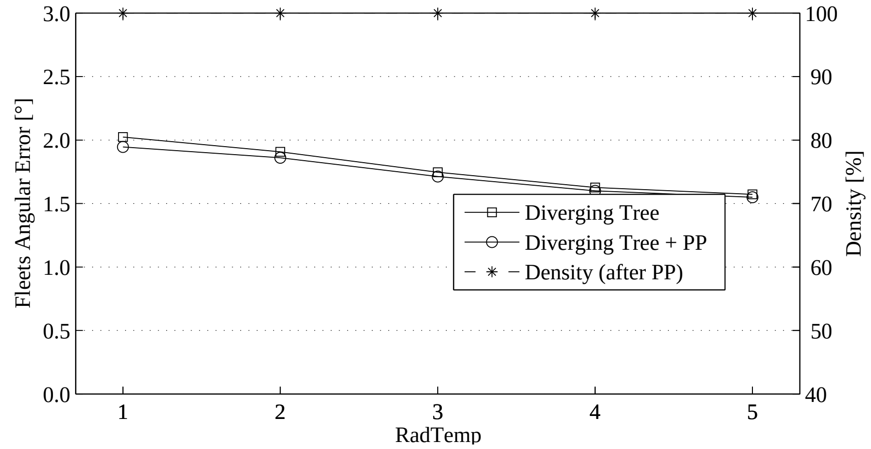


Figure A.19 Accuracy vs. RadTemp for the Diverging Tree sequence (fr. 19).

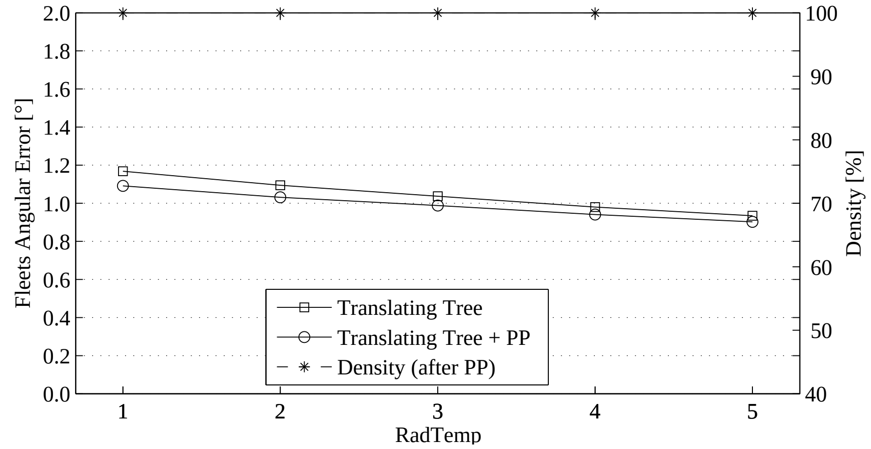


Figure A.20 Accuracy vs. RadTemp for the Translating Tree sequence (fr. 19).

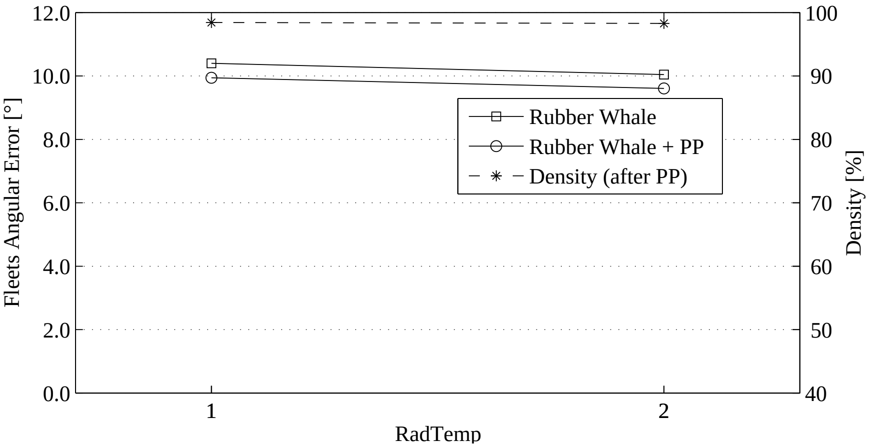


Figure A.21 Accuracy vs. RadTemp for the Rubber Whale sequence (frame 4).

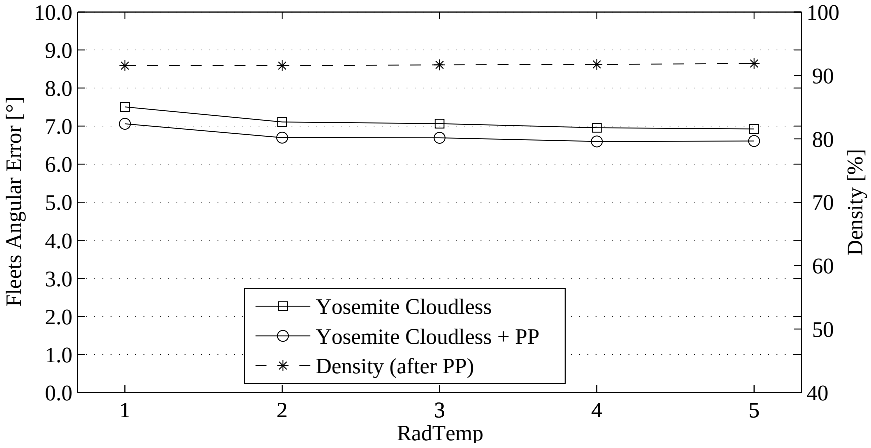


Figure A.22 Accuracy vs. RadTemp for the Yosemite Cloudless seq. (fr. 7).

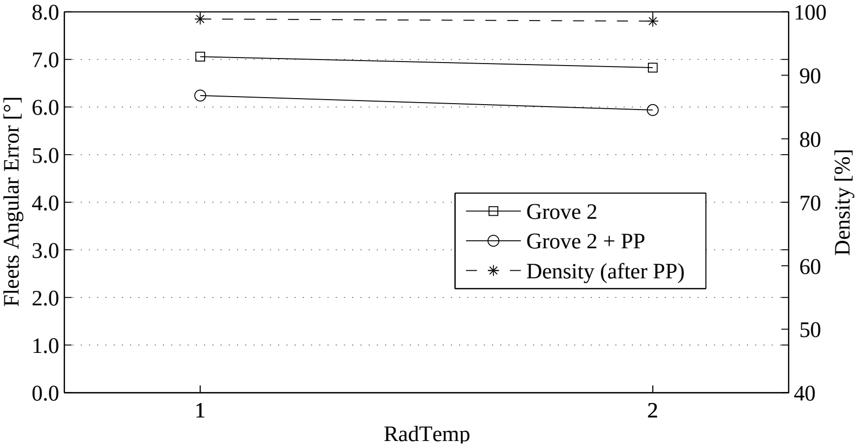


Figure A.23 Accuracy vs. RadTemp for the Grove 2 sequence (frame 4).

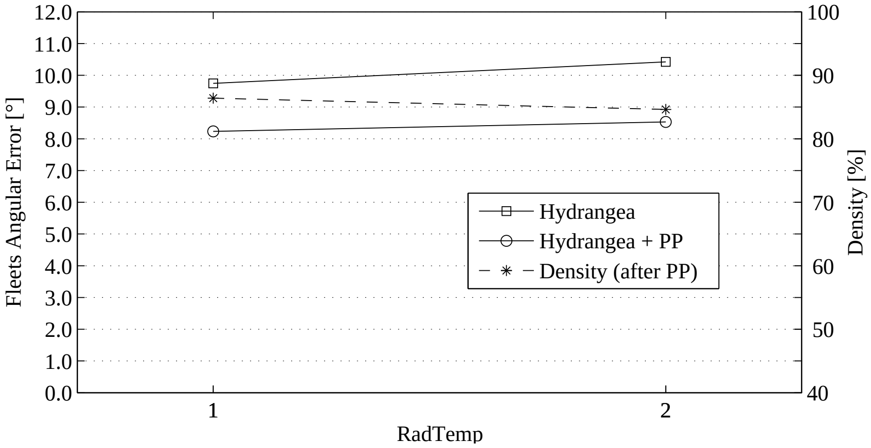


Figure A.24 Accuracy vs. RadTemp for the Hydrangea sequence (frame 4).

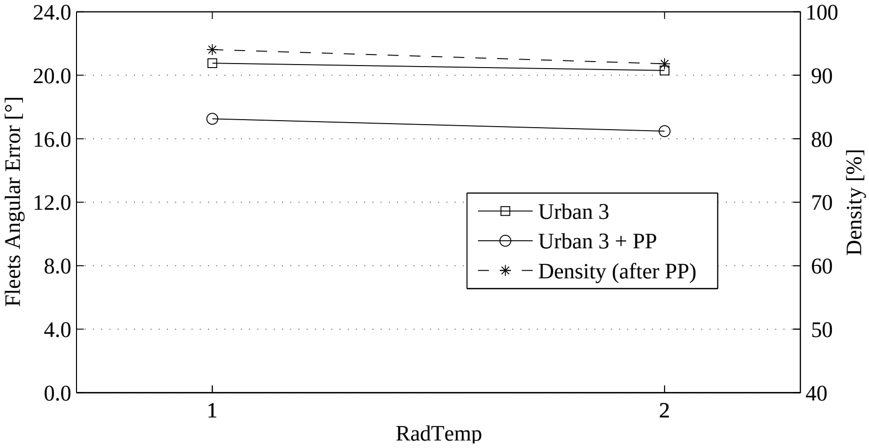


Figure A.25 Accuracy vs. RadTemp for the Urban 3 sequence (frame 4).

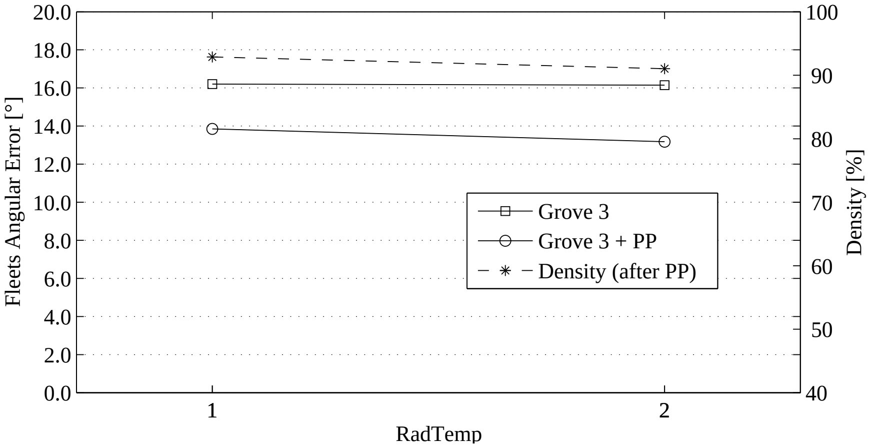


Figure A.26 Accuracy vs. RadTemp for the Grove 3 sequence (frame 4).

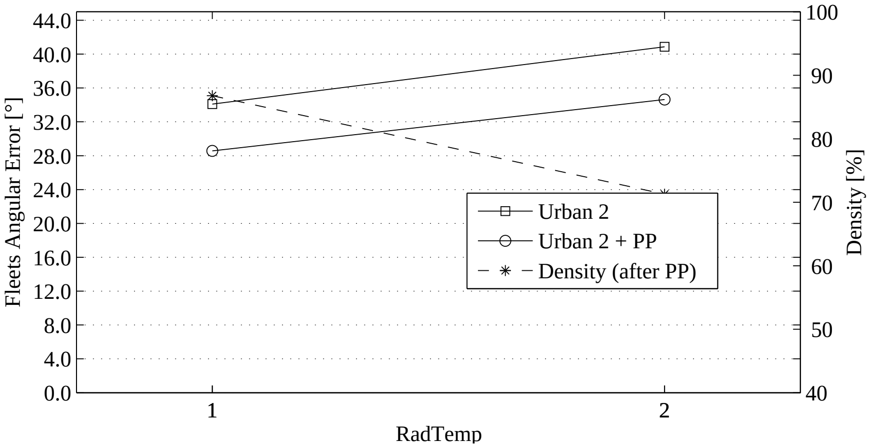


Figure A.27 Accuracy vs. RadTemp for the Urban 2 sequence (frame 4).

A.4 Spatial Radius

Description

The plots on the following pages show the influence of the RadSpat argument for every sequence. The parameters are the same as in table A.1, but with the optimum MaxLevel and smoothing filter parameters from table 6.2, NumDir=7, and the optimum RadTemp argument from the previous section.

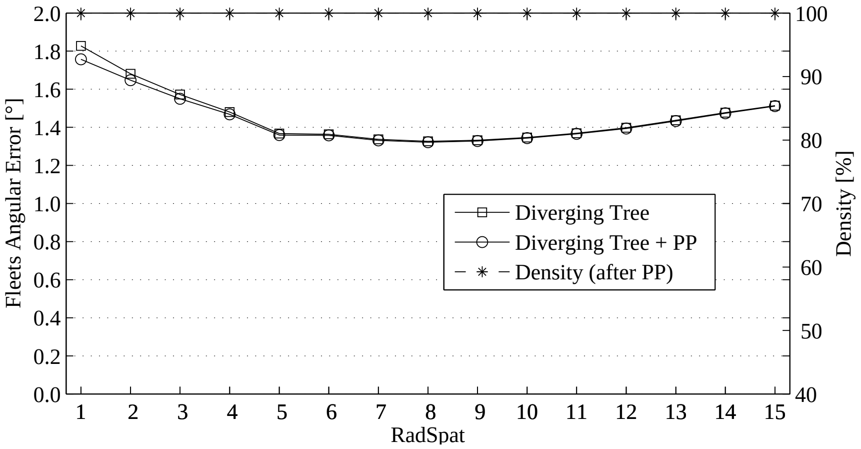


Figure A.28 Accuracy vs. RadSpat for the Diverging Tree sequence (frame 19).

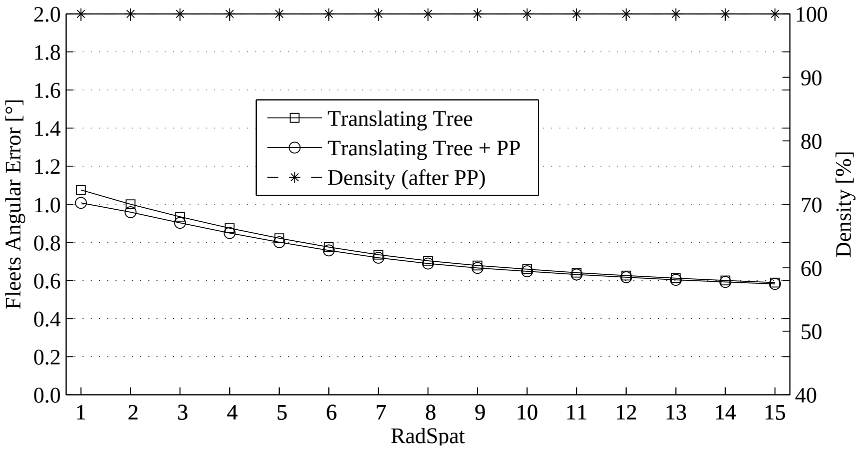


Figure A.29 Accuracy vs. RadSpat for the Translating Tree sequence (fr. 19).

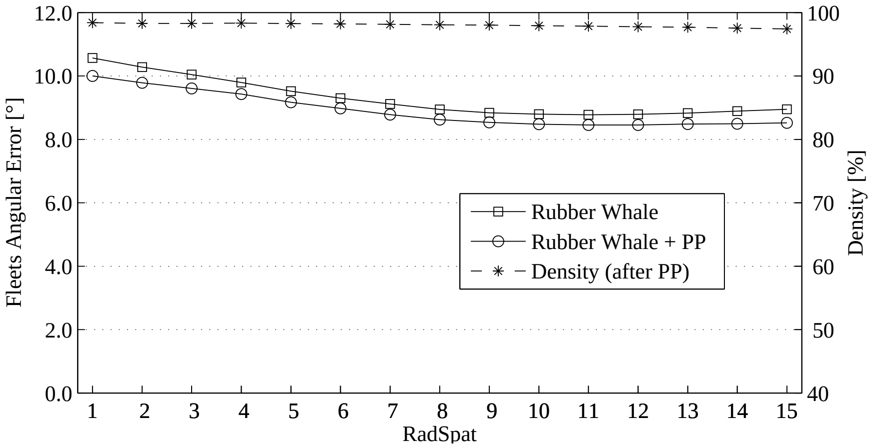


Figure A.30 Accuracy vs. RadSpat for the Rubber Whale sequence (frame 4).

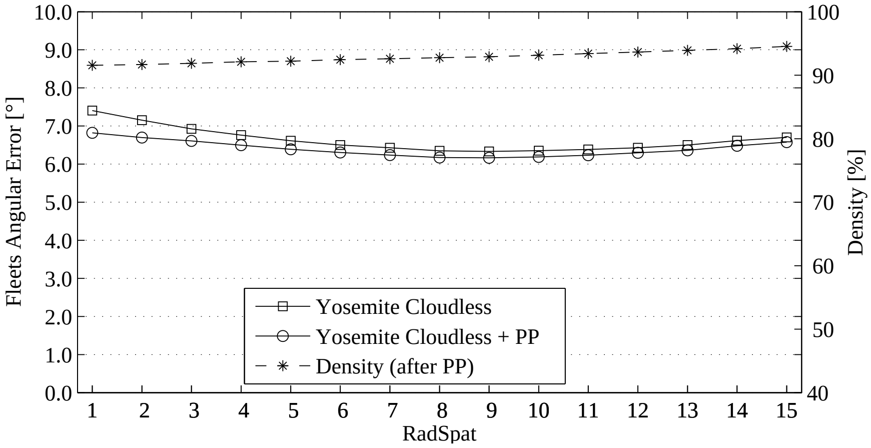


Figure A.31 Accuracy vs. RadSpat for the Yosemite Cloudless sequence (fr. 7).

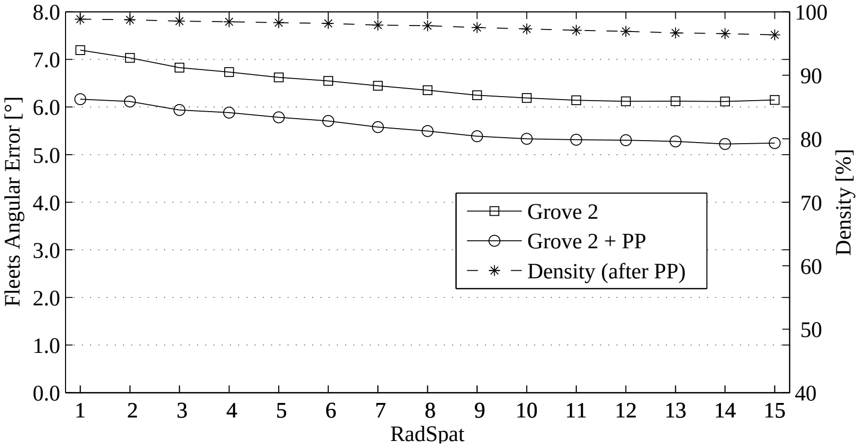


Figure A.32 Accuracy vs. RadSpat for the Grove 2 sequence (frame 4).

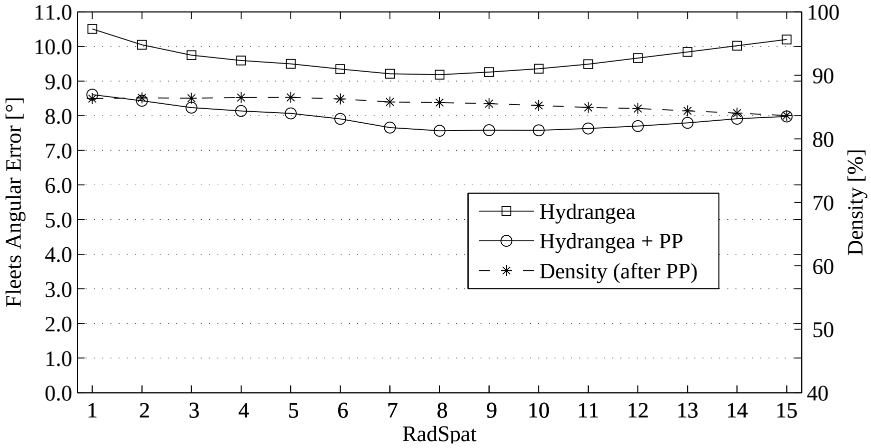


Figure A.33 Accuracy vs. RadSpat for the Hydrangea sequence (frame 4).

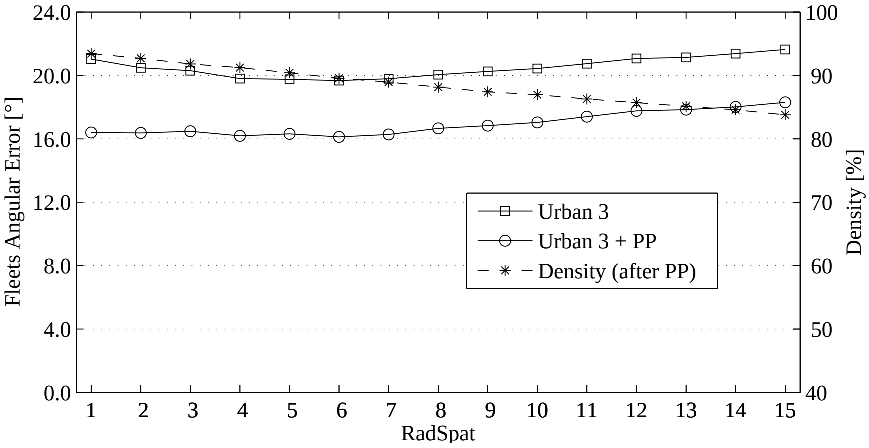


Figure A.34 Accuracy vs. RadSpat for the Urban 3 sequence (frame 4).

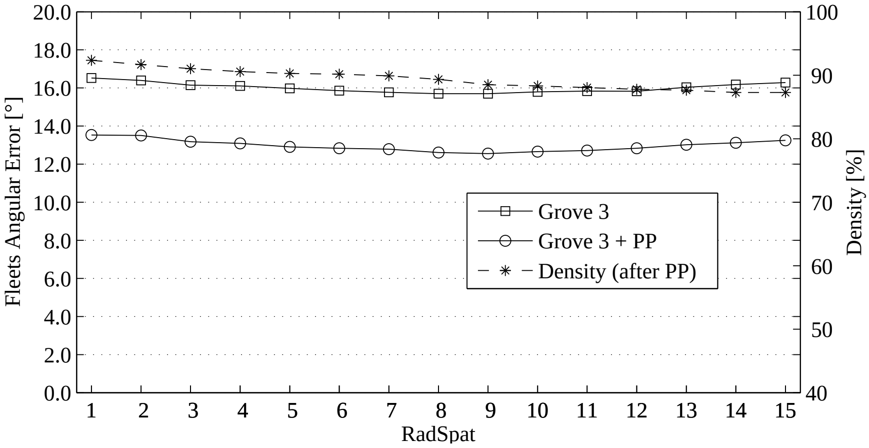


Figure A.35 Accuracy vs. RadSpat for the Grove 3 sequence (frame 4).

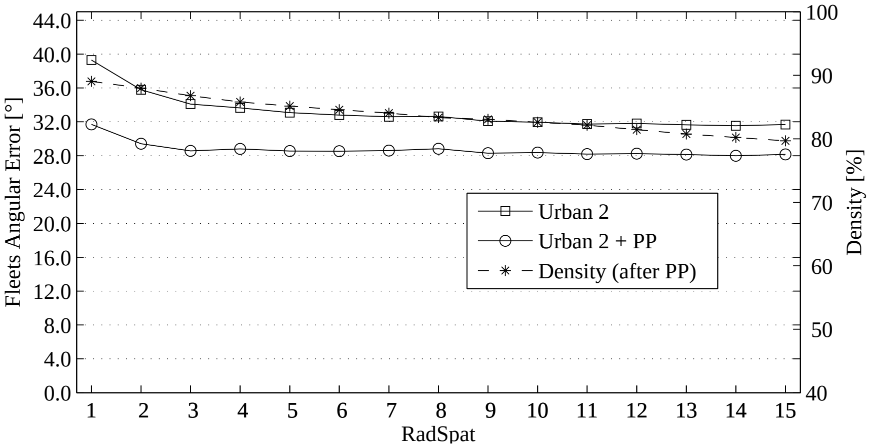


Figure A.36 Accuracy vs. RadSpat for the Urban 2 sequence (frame 4).

A.5 Comparison with other Algorithms

Introduction

This section contains the details for the comparison of the algorithms in section 6.4. Tables A.2 to A.6 show the parameter sets which are common for all test sequences.

Pyramid Algorithm

The parameter sets for the Pyramid Algorithm differ between the sequences and are given in the respective subsections.

For every sequence, three parameter sets for the Pyramid Algorithm were used:

Pyramid Algorithm 1 This parameter set basically uses the default parameters from the Matlab GUI, which were also used as starting point for the optimizations in chapter 6. However, the MaxLevel parameter was set according to the a priori knowledge of the maximum speeds in the sequences (table 6.1) and the threshold for the maximum speed (postprocessing) was set to $1.5 \cdot 2^{MaxLevel+1}$.

Pyramid Algorithm 2 This parameter set uses the optimized parameters found in the previous sections.

Pyramid Algorithm 3 This parameter set is the same for all the sequences and is intended to give good (but not optimal) results for all the sequences.

Conditions

All results in this appendix and in chapter 6 were generated on the Notebook (see table 5.2) with software revision 146 and OpenCV 2.1.

Parameter	**Basic Algorithm**
Smoothing Filter Type	None
Padding	15
Number of Directions	7
Angle Offset	0
Spatial Radius	6
Temporal Radius	2
Component Velocity Threshold	0.8
Optical Flow Threshold	0.8
Postprocessing Maximum Magnitude	5 (absolute)
WVM Filter Window Size	3
WVM Filter Minimum Certainty	75%
WVM Filter Number of Passes	2
Accuracy Minimum Threshold	51%
Accuracy Border Skip	15

Table A.2 Parameter sets for the Basic Algorithm.

Parameter	**Lucas-Kanade Extended 1**	**Lucas-Kanade Extended 2**	**Lucas-Kanade Extended 3**	**Lucas-Kanade Extended 4**
Smoothing Filter Type	Gaussian (3D)			
Smoothing Filter Size Spatial	7 × 7			
Smoothing Filter Sigma Spatial	1.5			
Smoothing Filter Size Temporal	7			
Smoothing Filter Sigma Temporal	1.5			
Window Size	5			
Downsampling Level	0			
Weighting Window	None	Gaussian		
Window Sigma/Radius	-	1.1		
Tau	1		7	0
Postprocessing Maximum Magnitude	5 (absolute)			
WVM Filter Window Size	3			
WVM Filter Minimum Certainty	75%			
WVM Filter Number of Passes	2			
Accuracy Minimum Threshold	51%			
Accuracy Border Skip	15			

Table A.3 Parameter sets for the Lucas-Kanade Extended algorithm.

Parameter	**Lucas-Kanade 1**	**Lucas-Kanade 2**	**Lucas-Kanade 3**
Smoothing Filter Type	Gaussian (3D)		
Smoothing Filter Size Spatial	7 × 7		
Smoothing Filter Sigma Spatial	1.5		
Smoothing Filter Size Temporal	7		
Smoothing Filter Sigma Temporal	1.5		
Window Size	5	9	13
Postprocessing Maximum Magnitude	24 (absolute)		
WVM Filter Window Size	3		
WVM Filter Minimum Certainty	75%		
WVM Filter Number of Passes	2		
Accuracy Minimum Threshold	51%		
Accuracy Border Skip	15		

Table A.4 Parameter sets for the Lucas-Kanade algorithm.

Parameter	Lucas-Kanade Pyramid 1	Lucas-Kanade Pyramid 2	Lucas-Kanade Pyramid 3
Smoothing Filter Type	Gaussian (3D)		
Smoothing Filter Size Spatial	7 × 7		
Smoothing Filter Sigma Spatial	1.5		
Smoothing Filter Size Temporal	7		
Smoothing Filter Sigma Temporal	1.5		
Window Size	5	9	13
Pyramid Levels	3		
Iterations	3		
Postprocessing Maximum Magnitude	24 (absolute)		
WVM Filter Window Size	3		
WVM Filter Minimum Certainty	75%		
WVM Filter Number of Passes	2		
Accuracy Minimum Threshold	51%		
Accuracy Border Skip	15		

Table A.5 Parameter sets for the Lucas-Kanade Pyramid algorithm.

Parameter	Horn-Schunck
Smoothing Filter Type	Gaussian (3D)
Smoothing Filter Size Spatial	7 × 7
Smoothing Filter Sigma Spatial	1.5
Smoothing Filter Size Temporal	7
Smoothing Filter Sigma Temporal	1.5
Alpha	1
Iterations	100
Postprocessing Maximum Magnitude	24 (absolute)
WVM Filter Window Size	3
WVM Filter Minimum Certainty	75%
WVM Filter Number of Passes	2
Accuracy Minimum Threshold	51%
Accuracy Border Skip	15

Table A.6 Parameter sets for the Horn-Schunck algorithm.

A.5.1 Diverging Tree Sequence

Parameter	**Pyramid Algorithm 1**	**Pyramid Algorithm 2**	**Pyramid Algorithm 3**
Smoothing Filter Type	Pauwels (2D)	None	Pauwels (2D)
Smoothing Filter Size	13 × 13	-	13 × 13
Padding	15		
Number of Directions	4	7	
Angle Offset	0		
Spatial Radius	3	8	6
Temporal Radius	1	5	2
Component Velocity Threshold	0.8		
Optical Flow Threshold	0.8		
Maximum Pyramid Level	0	1	3
Postprocessing Maximum Magnitude	3 (absolute)	6 (absolute)	24 (absolute)
WVM Filter Window Size	3		
WVM Filter Minimum Certainty	75%		
WVM Filter Number of Passes	2		
Accuracy Minimum Threshold	51%		
Accuracy Border Skip	15		

Table A.7 Parameter sets for the Diverging Tree sequence.

Algorithm	Density	Fleets Angular		Magnitude		Absolute Angular	
	[%]	μ [°]	σ [°]	μ	σ	μ [°]	σ [°]
Pyramid Algorithm 1	100.0	2.709	1.882	0.039	0.032	4.372	6.833
+Postprocessing	100.0	2.603	1.650	0.038	0.030	4.147	6.197
Pyramid Algorithm 2	100.0	1.327	0.825	0.021	0.019	2.048	3.164
+Postprocessing	100.0	**1.322**	0.814	0.021	0.019	2.037	3.132
Pyramid Algorithm 3	100.0	1.773	1.166	0.026	0.023	2.970	5.581
+Postprocessing	100.0	1.755	1.129	0.026	0.022	2.938	5.535
Basic Algorithm	100.0	1.765	1.079	0.026	0.020	2.950	5.308
+Postprocessing	100.0	**1.750**	1.040	0.026	0.020	2.930	5.248
Lucas-Kanade Extended 1	64.3	2.089	1.970	0.030	0.032	3.743	8.267
+Postprocessing	59.1	1.881	1.629	0.027	0.028	3.324	7.414
Lucas-Kanade Extended 2	56.7	1.988	1.969	0.028	0.031	3.690	8.997
+Postprocessing	50.4	1.750	1.567	0.025	0.026	3.169	7.554
Lucas-Kanade Extended 3	27.4	1.647	1.386	0.025	0.025	2.705	5.603
+Postprocessing	19.9	1.438	1.012	0.022	0.020	2.084	2.984
Lucas-Kanade Extended 4	100.0	2.507	2.464	0.035	0.040	4.664	9.225
+Postprocessing	100.0	**2.147**	1.779	0.030	0.029	4.051	7.865
Lucas-Kanade 1	100.0	10.083	10.150	0.158	0.229	15.409	20.252
+Postprocessing	100.0	7.785	7.443	0.117	0.145	12.061	16.238
Lucas-Kanade 2	100.0	6.537	6.502	0.098	0.121	10.496	14.909
+Postprocessing	100.0	5.722	5.463	0.085	0.096	9.263	13.261
Lucas-Kanade 3	100.0	5.020	4.866	0.074	0.089	8.337	12.548
+Postprocessing	100.0	**4.631**	4.340	0.068	0.078	7.755	11.856
Lucas-Kanade Pyramid 1	100.0	3.784	2.983	0.054	0.052	6.593	10.934
+Postprocessing	100.0	**3.634**	2.691	0.052	0.049	6.325	10.436
Lucas-Kanade Pyramid 2	100.0	3.767	2.868	0.058	0.050	6.219	11.072
+Postprocessing	100.0	3.735	2.761	0.058	0.049	6.162	10.965
Lucas-Kanade Pyramid 3	100.0	4.240	2.573	0.069	0.054	6.323	11.986
+Postprocessing	100.0	4.232	2.559	0.069	0.054	6.308	11.979
Horn-Schunck	100.0	8.823	7.253	0.142	0.140	12.743	15.792
+Postprocessing	100.0	**7.011**	5.900	0.113	0.110	10.149	13.278

Table A.8 Results for the Diverging Tree sequence.

A.5.2 Translating Tree Sequence

Parameter	**Pyramid Algorithm 1**	**Pyramid Algorithm 2**	**Pyramid Algorithm 3**
Smoothing Filter Type	Pauwels (2D)	None	Pauwels (2D)
Smoothing Filter Size	13 × 13	-	13 × 13
Padding	15		
Number of Directions	4	7	
Angle Offset	0		
Spatial Radius	3	15	6
Temporal Radius	1	5	2
Component Velocity Threshold	0.8		
Optical Flow Threshold	0.8		
Maximum Pyramid Level	0		3
Postprocessing Maximum Magnitude	3 (absolute)		24 (absolute)
WVM Filter Window Size	3		
WVM Filter Minimum Certainty	75%		
WVM Filter Number of Passes	2		
Accuracy Minimum Threshold	51%		
Accuracy Border Skip	15		

Table A.9 Parameter sets for the Translating Tree sequence.

Algorithm	Density	Fleets Angular		Magnitude		Absolute Angular	
	[%]	μ [°]	σ [°]	μ	σ	μ [°]	σ [°]
Pyramid Algorithm 1	100.0	1.891	1.831	0.047	0.048	1.916	2.087
+Postprocessing	100.0	1.798	1.630	0.042	0.036	1.833	1.877
Pyramid Algorithm 2	100.0	0.589	0.336	0.025	0.023	0.509	0.373
+Postprocessing	100.0	**0.581**	0.329	0.025	0.023	0.504	0.368
Pyramid Algorithm 3	100.0	1.068	0.778	0.030	0.023	1.064	0.911
+Postprocessing	100.0	1.047	0.755	0.029	0.022	1.041	0.886
Basic Algorithm	100.0	0.910	0.627	0.036	0.040	0.793	0.690
+Postprocessing	100.0	**0.874**	0.585	0.034	0.035	0.772	0.664
Lucas-Kanade Extended 1	58.3	0.604	0.614	0.025	0.024	0.527	0.652
+Postprocessing	52.8	0.531	0.524	0.022	0.021	0.462	0.555
Lucas-Kanade Extended 2	49.0	0.649	0.661	0.027	0.027	0.564	0.693
+Postprocessing	42.4	0.567	0.565	0.023	0.023	0.490	0.594
Lucas-Kanade Extended 3	15.6	0.523	0.532	0.021	0.019	0.460	0.591
+Postprocessing	9.0	0.472	0.520	0.018	0.015	0.422	0.593
Lucas-Kanade Extended 4	100.0	1.027	1.230	0.047	0.058	0.824	1.236
+Postprocessing	100.0	**0.823**	0.872	0.038	0.041	0.658	0.895
Lucas-Kanade 1	100.0	18.400	18.231	0.621	0.624	17.566	22.395
+Postprocessing	100.0	16.051	15.617	0.542	0.509	15.073	18.859
Lucas-Kanade 2	100.0	14.123	14.161	0.490	0.466	13.187	16.865
+Postprocessing	100.0	13.109	13.004	0.454	0.419	12.169	15.328
Lucas-Kanade 3	100.0	11.887	11.982	0.416	0.389	10.981	14.085
+Postprocessing	100.0	**11.310**	11.294	0.395	0.362	10.437	13.248
Lucas-Kanade Pyramid 1	100.0	0.454	0.399	0.023	0.021	0.342	0.420
+Postprocessing	100.0	0.417	0.341	0.021	0.018	0.312	0.367
Lucas-Kanade Pyramid 2	100.0	0.311	0.211	0.018	0.013	0.212	0.223
+Postprocessing	100.0	**0.307**	0.206	0.018	0.012	0.208	0.218
Lucas-Kanade Pyramid 3	100.0	0.391	1.442	0.022	0.049	0.262	1.448
+Postprocessing	100.0	0.389	1.440	0.022	0.048	0.260	1.448
Horn-Schunck	100.0	16.189	13.820	0.550	0.477	14.810	16.523
+Postprocessing	100.0	**14.508**	11.978	0.502	0.391	12.877	14.019

Table A.10 Results for the Translating Tree sequence.

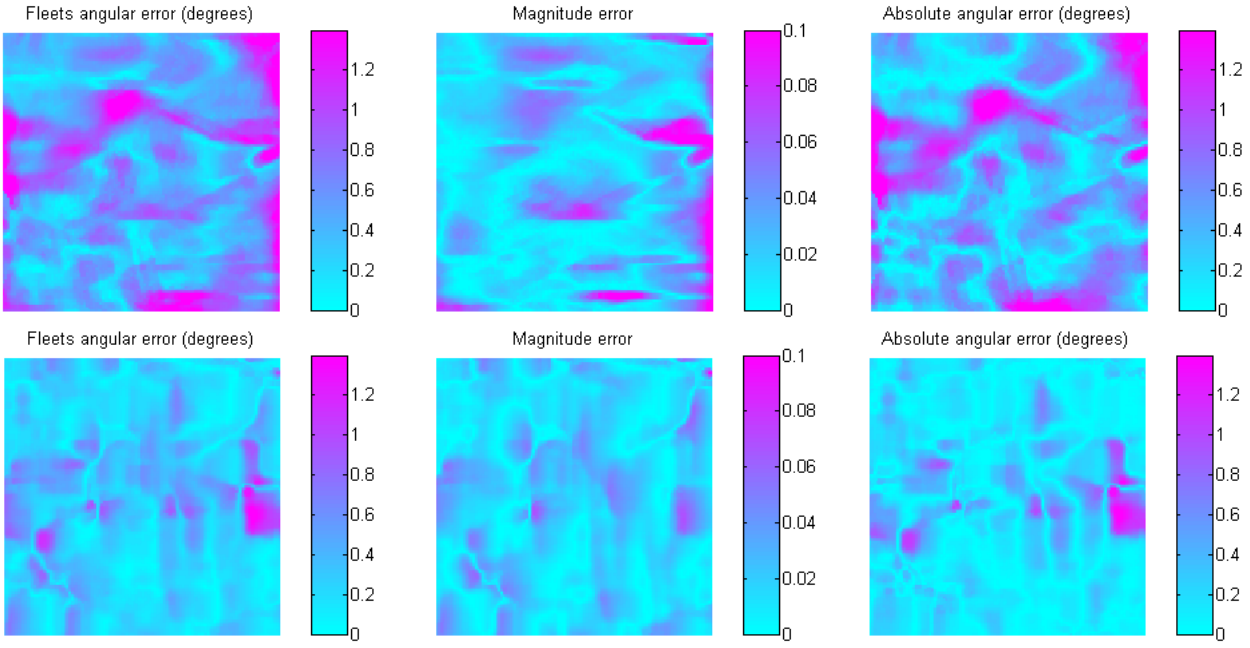


Figure A.37 Error plots for the Translating Tree sequence, Pyramid Algorithm 2 (top) and Lucas-Kanade Pyramid 2 (bottom).

A.5.3 Rubber Whale Sequence

<table>
<tr><th>Parameter</th><th>Pyramid Algorithm 1</th><th>Pyramid Algorithm 2</th><th>Pyramid Algorithm 3</th></tr>
<tr><td>Smoothing Filter Type</td><td>Pauwels (2D)</td><td>None</td><td>Pauwels (2D)</td></tr>
<tr><td>Smoothing Filter Size</td><td>13×13</td><td>-</td><td>13×13</td></tr>
<tr><td>Padding</td><td colspan="3">15</td></tr>
<tr><td>Number of Directions</td><td>4</td><td colspan="2">7</td></tr>
<tr><td>Angle Offset</td><td colspan="3">0</td></tr>
<tr><td>Spatial Radius</td><td>3</td><td>11</td><td>6</td></tr>
<tr><td>Temporal Radius</td><td>1</td><td colspan="2">2</td></tr>
<tr><td>Component Velocity Threshold</td><td colspan="3">0.8</td></tr>
<tr><td>Optical Flow Threshold</td><td colspan="3">0.8</td></tr>
<tr><td>Maximum Pyramid Level</td><td>1</td><td>0</td><td>3</td></tr>
<tr><td>Postprocessing Maximum Magnitude</td><td>6 (absolute)</td><td>3 (absolute)</td><td>24 (absolute)</td></tr>
<tr><td>WVM Filter Window Size</td><td colspan="3">3</td></tr>
<tr><td>WVM Filter Minimum Certainty</td><td colspan="3">75%</td></tr>
<tr><td>WVM Filter Number of Passes</td><td colspan="3">2</td></tr>
<tr><td>Accuracy Minimum Threshold</td><td colspan="3">51%</td></tr>
<tr><td>Accuracy Border Skip</td><td colspan="3">15</td></tr>
</table>

Table A.11 Parameter sets for the Rubber Whale sequence.

Algorithm	**Density**	**Fleets Angular**		**Magnitude**		**Absolute Angular**	
	[%]	μ [°]	σ [°]	μ	σ	μ [°]	σ [°]
Pyramid Algorithm 1	99.1	12.233	22.095	0.197	0.403	16.232	34.259
+Postprocessing	98.7	11.619	21.212	0.185	0.369	15.523	33.516
Pyramid Algorithm 2	98.9	8.775	19.864	0.146	0.356	11.561	30.339
+Postprocessing	97.8	**8.453**	19.338	0.137	0.335	11.189	29.841
Pyramid Algorithm 3	99.1	9.466	18.164	0.153	0.301	12.737	29.885
+Postprocessing	99.0	9.226	17.747	0.148	0.285	12.452	29.502
Basic Algorithm	99.0	9.362	20.644	0.153	0.376	12.312	31.290
+Postprocessing	98.9	**9.173**	20.399	0.150	0.371	12.095	31.094
Lucas-Kanade Extended 1	20.3	12.315	19.521	0.231	0.363	15.339	32.195
+Postprocessing	14.4	11.475	18.981	0.205	0.309	14.556	31.612
Lucas-Kanade Extended 2	13.1	12.819	20.191	0.234	0.396	16.374	33.392
+Postprocessing	7.5	12.492	20.462	0.203	0.329	16.675	34.149
Lucas-Kanade Extended 3	1.3	15.556	22.460	0.216	0.328	22.633	40.243
+Postprocessing	0.6	14.629	20.853	0.210	0.321	21.336	37.918
Lucas-Kanade Extended 4	99.1	14.840	21.642	0.363	0.736	17.475	32.887
+Postprocessing	97.7	**13.022**	19.518	0.280	0.401	15.460	30.979
Lucas-Kanade 1	99.1	23.914	23.938	0.514	0.699	28.184	36.688
+Postprocessing	99.1	20.855	22.439	0.434	0.573	24.416	34.512
Lucas-Kanade 2	99.1	19.811	22.383	0.411	0.556	23.444	34.341
+Postprocessing	99.1	18.550	21.808	0.380	0.513	21.962	33.651
Lucas-Kanade 3	99.1	18.024	21.841	0.366	0.501	21.453	33.723
+Postprocessing	99.1	**17.339**	21.533	0.349	0.479	20.669	33.400
Lucas-Kanade Pyramid 1	99.1	13.463	22.276	0.281	0.610	16.622	34.303
+Postprocessing	99.1	**13.249**	22.131	0.271	0.567	16.405	34.203
Lucas-Kanade Pyramid 2	99.1	13.831	23.137	0.245	0.446	17.671	36.179
+Postprocessing	99.1	13.772	23.095	0.243	0.438	17.603	36.141
Lucas-Kanade Pyramid 3	99.1	14.757	24.233	0.252	0.446	19.129	38.099
+Postprocessing	99.1	14.729	24.213	0.251	0.444	19.093	38.076
Horn-Schunck	99.1	19.752	20.667	0.412	0.520	23.231	33.503
+Postprocessing	99.1	**18.051**	20.453	0.379	0.489	21.227	33.251

Table A.12 Results for the Rubber Whale sequence.

A.5.4 Yosemite Cloudless Sequence

Parameter	**Pyramid Algorithm 1**	**Pyramid Algorithm 2**	**Pyramid Algorithm 3**
Smoothing Filter Type	Pauwels (2D)	Gaussian (2D)	Pauwels (2D)
Smoothing Filter Size	13×13	9×9	13×13
Smoothing Filter Sigma	-	1.5	-
Padding	15		
Number of Directions	4	7	
Angle Offset	0		
Spatial Radius	3	9	6
Temporal Radius	1	5	2
Component Velocity Threshold	0.8		
Optical Flow Threshold	0.8		
Maximum Pyramid Level	1		3
Postprocessing Maximum Magnitude	6 (absolute)		24 (absolute)
WVM Filter Window Size	3		
WVM Filter Minimum Certainty	75%		
WVM Filter Number of Passes	2		
Accuracy Minimum Threshold	51%		
Accuracy Border Skip	15		

Table A.13 Parameter sets for the Yosemite Cloudless sequence.

Algorithm	**Density**	**Fleets Angular**		**Magnitude**		**Absolute Angular**	
	[%]	μ [°]	σ [°]	μ	σ	μ [°]	σ [°]
Pyramid Algorithm 1	92.6	8.758	11.943	0.213	0.315	13.151	22.900
+Postprocessing	91.5	8.115	11.183	0.194	0.281	12.119	21.118
Pyramid Algorithm 2	94.0	6.334	10.222	0.146	0.217	11.679	21.792
+Postprocessing	92.9	**6.163**	9.988	0.141	0.204	10.790	19.696
Pyramid Algorithm 3	94.1	6.763	9.973	0.154	0.212	11.854	20.972
+Postprocessing	93.2	6.570	9.739	0.149	0.203	10.995	18.988
Basic Algorithm	89.7	9.877	19.323	0.276	0.524	13.272	26.991
+Postprocessing	88.7	**9.625**	18.976	0.272	0.525	12.468	25.631
Lucas-Kanade Extended 1	48.9	3.496	5.429	0.076	0.126	5.072	10.583
+Postprocessing	46.2	3.125	4.877	0.065	0.110	4.648	9.883
Lucas-Kanade Extended 2	42.3	3.492	5.282	0.073	0.120	5.210	10.825
+Postprocessing	38.7	3.069	4.667	0.062	0.104	4.739	10.104
Lucas-Kanade Extended 3	10.0	2.968	4.344	0.050	0.085	5.677	13.130
+Postprocessing	5.6	2.446	3.333	0.037	0.064	5.537	14.726
Lucas-Kanade Extended 4	80.2	4.893	7.365	0.172	0.268	6.052	11.595
+Postprocessing	79.4	**4.136**	6.192	0.150	0.233	5.048	9.835
Lucas-Kanade 1	100.0	11.677	16.475	0.366	0.573	16.497	21.982
+Postprocessing	100.0	10.055	14.103	0.332	0.523	13.882	18.179
Lucas-Kanade 2	100.0	9.312	13.163	0.310	0.488	13.173	18.331
+Postprocessing	100.0	8.633	12.138	0.294	0.464	12.068	16.469
Lucas-Kanade 3	100.0	8.225	11.554	0.282	0.442	11.797	17.042
+Postprocessing	100.0	**7.852**	10.988	0.272	0.428	11.146	15.788
Lucas-Kanade Pyramid 1	81.9	4.337	7.647	0.122	0.199	5.996	13.553
+Postprocessing	81.1	**3.996**	6.942	0.113	0.169	5.451	12.332
Lucas-Kanade Pyramid 2	84.0	4.897	8.562	0.130	0.200	6.986	15.695
+Postprocessing	83.2	4.624	8.009	0.124	0.180	6.539	14.782
Lucas-Kanade Pyramid 3	86.0	6.048	9.435	0.157	0.211	8.577	16.335
+Postprocessing	85.2	5.795	9.011	0.152	0.199	8.149	15.472
Horn-Schunck	100.0	9.670	10.764	0.319	0.470	20.495	24.054
+Postprocessing	100.0	**8.225**	9.521	0.291	0.447	18.546	23.582

Table A.14 Results for the Yosemite Cloudless sequence.

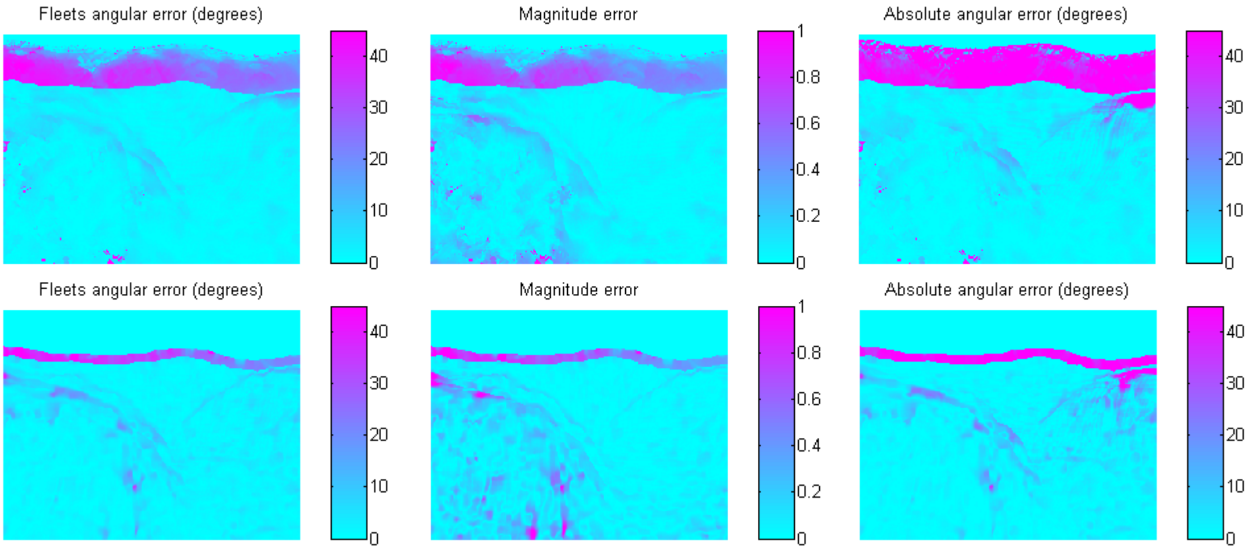


Figure A.38 Error plots for the Yosemite Cloudless sequence, Pyramid Algorithm 2 (top) and Lucas-Kanade Pyramid 1 (bottom).

A.5.5 Grove 2 Sequence

Parameter	Pyramid Algorithm 1	Pyramid Algorithm 2	Pyramid Algorithm 3
Smoothing Filter Type	Pauwels (2D)		
Smoothing Filter Size	13 × 13		
Padding	15		
Number of Directions	4	7	
Angle Offset	0		
Spatial Radius	3	14	6
Temporal Radius	1	2	
Component Velocity Threshold	0.8		
Optical Flow Threshold	0.8		
Maximum Pyramid Level	1		3
Postprocessing Maximum Magnitude	6 (absolute)		24 (absolute)
WVM Filter Window Size	3		
WVM Filter Minimum Certainty	75%		
WVM Filter Number of Passes	2		
Accuracy Minimum Threshold	51%		
Accuracy Border Skip	15		

Table A.15 Parameter sets for the Grove 2 sequence.

Algorithm	Density	Fleets Angular		Magnitude		Absolute Angular	
	[%]	μ [°]	σ [°]	μ	σ	μ [°]	σ [°]
Pyramid Algorithm 1	99.6	8.970	16.453	0.301	0.609	9.216	17.983
+Postprocessing	96.9	7.391	13.804	0.241	0.456	7.536	14.940
Pyramid Algorithm 2	98.7	6.115	14.080	0.183	0.444	6.309	15.145
+Postprocessing	96.6	**5.224**	12.059	0.153	0.362	5.372	12.900
Pyramid Algorithm 3	99.7	6.318	13.493	0.197	0.456	6.455	14.240
+Postprocessing	98.9	5.774	12.330	0.177	0.394	5.889	12.933
Basic Algorithm	99.6	33.132	42.911	1.037	1.125	37.625	56.401
+Postprocessing	96.6	**31.437**	41.965	1.007	1.111	35.843	55.865
Lucas-Kanade Extended 1	25.9	10.812	19.226	0.495	0.706	10.630	22.073
+Postprocessing	20.2	10.303	18.477	0.463	0.655	10.094	21.359
Lucas-Kanade Extended 2	18.6	11.612	20.511	0.528	0.746	11.469	23.688
+Postprocessing	12.3	11.235	19.713	0.501	0.695	11.062	23.003
Lucas-Kanade Extended 3	1.5	14.595	23.569	0.612	0.786	14.860	28.761
+Postprocessing	0.4	15.989	23.396	0.620	0.748	16.299	28.635
Lucas-Kanade Extended 4	100.0	13.123	20.041	0.651	0.803	12.284	23.017
+Postprocessing	96.3	**10.907**	16.529	0.572	0.653	9.761	18.494
Lucas-Kanade 1	100.0	26.160	27.032	1.153	1.018	25.232	33.250
+Postprocessing	100.0	23.489	23.702	1.107	0.891	21.752	28.566
Lucas-Kanade 2	100.0	21.076	22.087	1.044	0.872	19.535	26.498
+Postprocessing	100.0	19.921	20.606	1.018	0.825	18.079	24.265
Lucas-Kanade 3	100.0	18.512	19.566	0.976	0.805	16.605	22.724
+Postprocessing	100.0	**17.862**	18.727	0.958	0.780	15.824	21.486
Lucas-Kanade Pyramid 1	100.0	6.479	12.996	0.350	0.614	6.035	14.334
+Postprocessing	100.0	6.254	12.629	0.334	0.530	5.819	13.884
Lucas-Kanade Pyramid 2	100.0	5.729	11.771	0.288	0.469	5.348	12.553
+Postprocessing	100.0	**5.689**	11.726	0.285	0.467	5.312	12.500
Lucas-Kanade Pyramid 3	100.0	5.822	11.690	0.286	0.457	5.468	12.289
+Postprocessing	100.0	5.808	11.676	0.285	0.456	5.457	12.274
Horn-Schunck	100.0	21.359	18.145	1.114	0.838	18.146	20.105
+Postprocessing	100.0	**20.151**	17.350	1.081	0.786	16.766	18.899

Table A.16 Results for the Grove 2 sequence.

A.5.6 Hydrangea Sequence

Parameter	**Pyramid Algorithm 1**	**Pyramid Algorithm 2**	**Pyramid Algorithm 3**
Smoothing Filter Type	Pauwels (2D)	Gaussian (2D)	Pauwels (2D)
Smoothing Filter Size	13 × 13		
Smoothing Filter Sigma	-	1.5	-
Padding	15		
Number of Directions	4	7	
Angle Offset	0		
Spatial Radius	3	8	6
Temporal Radius	1		2
Component Velocity Threshold	0.8		
Optical Flow Threshold	0.8		
Maximum Pyramid Level	2	1	3
Postprocessing Maximum Magnitude	12 (absolute)	6 (absolute)	24 (absolute)
WVM Filter Window Size	3		
WVM Filter Minimum Certainty	75%		
WVM Filter Number of Passes	2		
Accuracy Minimum Threshold	51%		
Accuracy Border Skip	15		

Table A.17 Parameter sets for the Hydrangea sequence.

Algorithm	Density	Fleets Angular		Magnitude		Absolute Angular	
	[%]	μ [°]	σ [°]	μ	σ	μ [°]	σ [°]
Pyramid Algorithm 1	93.8	13.830	25.257	0.776	1.487	15.036	30.906
+Postprocessing	91.2	11.771	22.504	0.624	1.087	12.806	28.052
Pyramid Algorithm 2	93.9	9.184	17.090	0.561	1.017	9.369	20.780
+Postprocessing	85.7	**7.562**	14.300	0.388	0.668	7.780	18.278
Pyramid Algorithm 3	93.6	12.264	23.324	0.677	1.275	13.103	28.350
+Postprocessing	91.7	10.823	21.313	0.596	1.098	11.557	26.360
Basic Algorithm	94.2	81.895	44.389	2.400	1.243	102.312	63.628
+Postprocessing	86.8	**84.220**	42.359	2.433	1.185	106.774	62.226
Lucas-Kanade Extended 1	22.4	37.070	28.302	1.652	1.521	39.993	40.157
+Postprocessing	17.9	36.855	26.374	1.618	1.522	39.608	38.973
Lucas-Kanade Extended 2	16.7	39.117	29.804	1.636	1.500	42.891	42.353
+Postprocessing	11.6	38.711	27.303	1.572	1.478	42.501	40.923
Lucas-Kanade Extended 3	1.5	34.984	25.219	1.213	1.077	40.537	41.977
+Postprocessing	0.4	33.899	22.893	1.139	1.024	40.047	41.723
Lucas-Kanade Extended 4	94.4	24.941	30.926	1.463	1.522	25.251	39.049
+Postprocessing	74.9	**23.635**	28.121	1.284	1.293	23.578	36.644
Lucas-Kanade 1	94.4	34.415	31.742	1.733	1.484	34.823	41.584
+Postprocessing	94.4	31.269	28.414	1.711	1.386	30.459	37.664
Lucas-Kanade 2	94.4	28.488	27.620	1.645	1.371	27.716	36.561
+Postprocessing	94.4	27.014	25.897	1.636	1.349	25.658	34.347
Lucas-Kanade 3	94.4	25.328	25.049	1.595	1.341	23.778	33.054
+Postprocessing	94.4	**24.473**	23.973	1.588	1.333	22.558	31.548
Lucas-Kanade Pyramid 1	94.4	11.550	16.280	1.411	11.845	10.550	21.563
+Postprocessing	94.2	11.236	15.746	1.017	1.060	10.210	20.963
Lucas-Kanade Pyramid 2	94.4	9.125	12.684	0.939	1.738	8.198	17.732
+Postprocessing	94.3	9.072	12.630	0.900	1.014	8.152	17.685
Lucas-Kanade Pyramid 3	94.4	8.354	11.595	0.865	1.106	7.427	16.143
+Postprocessing	94.4	**8.336**	11.582	0.857	1.018	7.412	16.128
Horn-Schunck	94.4	26.641	23.747	1.697	1.364	23.977	30.410
+Postprocessing	94.4	**25.317**	22.461	1.673	1.321	22.196	28.919

Table A.18 Results for the Hydrangea sequence.

A.5.7 Urban 3 Sequence

Parameter	Pyramid Algorithm 1	Pyramid Algorithm 2	Pyramid Algorithm 3
Smoothing Filter Type	Pauwels (2D)		
Smoothing Filter Size	13 × 13	17 × 17	13 × 13
Padding	15		
Number of Directions	4	7	
Angle Offset	0		
Spatial Radius	3	4	6
Temporal Radius	1	2	
Component Velocity Threshold	0.8		
Optical Flow Threshold	0.8		
Maximum Pyramid Level	3		
Postprocessing Maximum Magnitude	24 (absolute)		
WVM Filter Window Size	3		
WVM Filter Minimum Certainty	75%		
WVM Filter Number of Passes	2		
Accuracy Minimum Threshold	51%		
Accuracy Border Skip	15		

Table A.19 Parameter sets for the Urban 3 sequence.

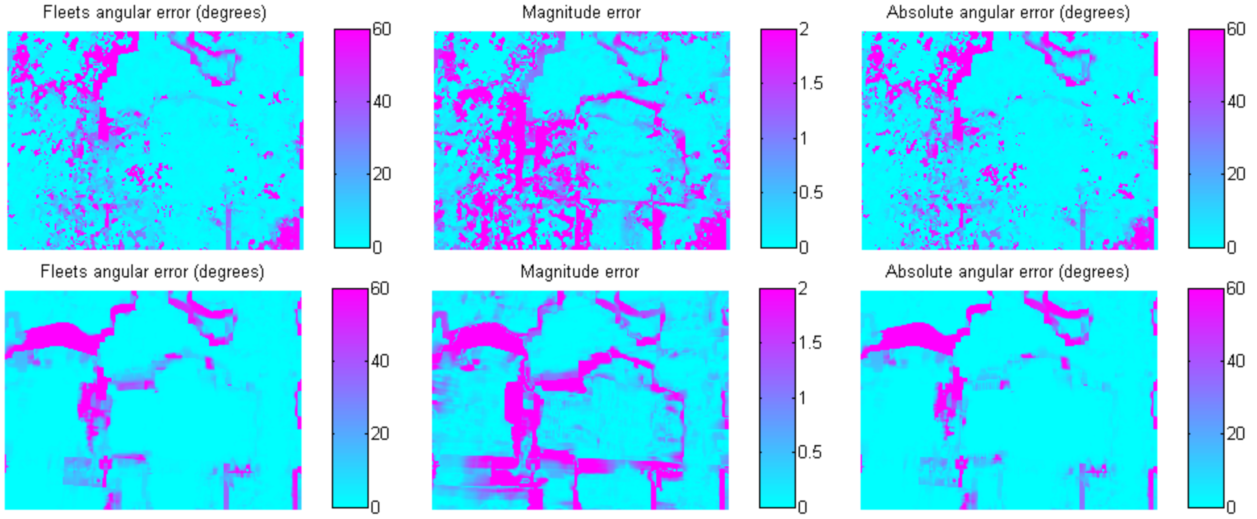


Figure A.39 Error plots for the Urban 3 sequence, Pyramid Algorithm 3 (top) and Lucas-Kanade Pyramid 2 (bottom).

Algorithm	Density	Fleets Angular		Magnitude		Absolute Angular	
	[%]	μ [°]	σ [°]	μ	σ	μ [°]	σ [°]
Pyramid Algorithm 1	95.7	23.723	36.804	2.095	3.549	24.625	40.432
+Postprocessing	89.4	19.669	33.582	1.758	2.984	20.336	36.959
Pyramid Algorithm 2	96.7	19.802	34.423	1.737	2.987	20.735	38.493
+Postprocessing	91.2	16.188	31.245	1.462	2.554	16.925	35.181
Pyramid Algorithm 3	95.8	19.632	34.075	1.744	3.018	20.507	37.876
+Postprocessing	89.6	**16.105**	31.017	1.459	2.572	16.788	34.637
Basic Algorithm	92.3	76.308	50.771	5.022	4.041	84.342	63.286
+Postprocessing	72.2	**73.862**	50.171	4.745	3.924	82.902	64.336
Lucas-Kanade Extended 1	7.0	32.918	39.641	3.233	3.716	32.563	46.894
+Postprocessing	4.2	31.941	38.427	2.884	3.372	31.339	46.739
Lucas-Kanade Extended 2	4.6	34.586	41.715	3.080	3.409	34.603	49.750
+Postprocessing	2.4	36.141	41.030	3.011	3.269	35.983	51.168
Lucas-Kanade Extended 3	0.3	55.000	51.403	2.099	2.717	59.649	61.793
+Postprocessing	0.1	55.270	46.053	1.437	1.735	60.575	55.620
Lucas-Kanade Extended 4	100.0	44.253	43.210	5.092	8.224	44.255	48.972
+Postprocessing	62.3	**35.990**	38.760	3.795	4.227	35.324	46.475
Lucas-Kanade 1	100.0	52.722	40.751	4.976	5.144	52.712	48.646
+Postprocessing	97.3	49.635	38.729	4.635	3.895	48.792	46.707
Lucas-Kanade 2	100.0	48.167	38.732	4.835	4.521	47.445	46.188
+Postprocessing	98.2	46.432	37.746	4.609	3.933	45.305	45.225
Lucas-Kanade 3	100.0	45.377	37.392	4.785	4.398	44.136	44.530
+Postprocessing	98.6	**44.164**	36.757	4.606	3.978	42.650	43.846
Lucas-Kanade Pyramid 1	100.0	12.459	30.796	1.219	2.836	12.460	33.594
+Postprocessing	99.8	12.067	30.419	1.158	2.596	12.044	33.175
Lucas-Kanade Pyramid 2	100.0	11.483	30.117	1.102	2.449	11.362	32.963
+Postprocessing	100.0	**11.369**	30.024	1.086	2.419	11.248	32.865
Lucas-Kanade Pyramid 3	100.0	11.928	30.602	1.214	2.581	11.709	33.568
+Postprocessing	100.0	11.898	30.580	1.212	2.580	11.679	33.545
Horn-Schunck	100.0	46.053	35.057	4.911	4.236	42.784	41.190
+Postprocessing	98.6	**44.969**	34.451	4.783	3.962	41.310	40.538

Table A.20 Results for the Urban 3 sequence.

A.5.8 Grove 3 Sequence

Parameter	**Pyramid Algorithm 1**	**Pyramid Algorithm 2**	**Pyramid Algorithm 3**
Smoothing Filter Type	Pauwels (2D)	Gaussian (2D)	Pauwels (2D)
Smoothing Filter Size	13 × 13	9 × 9	13 × 13
Smoothing Filter Sigma	-	1.5	-
Padding	15		
Number of Directions	4	7	
Angle Offset	0		
Spatial Radius	3	9	6
Temporal Radius	1	2	
Component Velocity Threshold	0.8		
Optical Flow Threshold	0.8		
Maximum Pyramid Level	3	2	3
Postprocessing Maximum Magnitude	24 (absolute)	12 (absolute)	24 (absolute)
WVM Filter Window Size	3		
WVM Filter Minimum Certainty	75%		
WVM Filter Number of Passes	2		
Accuracy Minimum Threshold	51%		
Accuracy Border Skip	15		

Table A.21 Parameter sets for the Grove 3 sequence.

Algorithm	**Density**	**Fleets Angular**		**Magnitude**		**Absolute Angular**	
	[%]	μ [°]	σ [°]	μ	σ	μ [°]	σ [°]
Pyramid Algorithm 1	98.2	19.814	29.336	1.506	2.763	20.375	32.667
+Postprocessing	92.9	16.708	26.283	1.191	1.916	17.068	29.367
Pyramid Algorithm 2	97.5	15.698	25.263	1.233	2.133	15.764	27.546
+Postprocessing	88.5	**12.556**	21.954	0.918	1.498	12.447	23.716
Pyramid Algorithm 3	98.7	16.321	26.370	1.248	2.139	16.333	28.703
+Postprocessing	95.4	14.411	24.168	1.089	1.721	14.294	26.190
Basic Algorithm	98.4	50.803	52.958	1.888	2.035	57.658	64.990
+Postprocessing	86.1	**46.217**	51.479	1.723	1.943	53.348	64.724
Lucas-Kanade Extended 1	48.4	30.833	35.810	1.793	2.230	31.697	43.214
+Postprocessing	39.7	28.809	34.661	1.744	2.289	29.544	42.590
Lucas-Kanade Extended 2	38.7	31.718	37.059	1.757	2.213	32.798	44.595
+Postprocessing	29.1	28.337	35.114	1.642	2.245	29.288	43.216
Lucas-Kanade Extended 3	5.7	21.313	31.699	1.048	1.683	21.860	38.929
+Postprocessing	2.7	17.255	28.316	0.832	1.480	17.428	35.220
Lucas-Kanade Extended 4	100.0	30.738	36.251	1.703	2.142	31.563	42.651
+Postprocessing	85.5	**26.547**	33.069	1.607	2.070	26.967	40.006
Lucas-Kanade 1	100.0	37.329	36.316	1.878	2.111	38.698	43.550
+Postprocessing	99.9	34.946	34.395	1.849	2.000	35.727	41.374
Lucas-Kanade 2	100.0	33.028	33.680	1.818	2.020	33.636	40.390
+Postprocessing	100.0	31.871	32.629	1.813	2.005	32.177	39.118
Lucas-Kanade 3	100.0	30.489	31.930	1.793	2.014	30.619	38.251
+Postprocessing	100.0	**29.798**	31.275	1.792	2.014	29.747	37.444
Lucas-Kanade Pyramid 1	100.0	13.501	23.011	1.036	1.718	13.308	26.686
+Postprocessing	100.0	13.227	22.639	1.018	1.637	12.997	26.213
Lucas-Kanade Pyramid 2	100.0	12.551	20.856	0.999	1.549	12.081	23.962
+Postprocessing	100.0	**12.483**	20.767	0.997	1.545	11.999	23.832
Lucas-Kanade Pyramid 3	100.0	12.570	20.022	1.021	1.536	11.831	22.805
+Postprocessing	100.0	12.546	19.991	1.020	1.536	11.802	22.751
Horn-Schunck	100.0	32.690	31.576	1.844	2.024	32.406	37.049
+Postprocessing	100.0	**30.994**	30.652	1.813	1.972	30.403	36.061

Table A.22 Results for the Grove 3 sequence.

A.5.9 Urban 2 Sequence

Parameter	Pyramid Algorithm 1	Pyramid Algorithm 2	Pyramid Algorithm 3
Smoothing Filter Type	Pauwels (2D)	Gaussian (2D)	Pauwels (2D)
Smoothing Filter Size	13×13	17×17	13×13
Smoothing Filter Sigma	-	1.5	-
Padding	15		
Number of Directions	4	7	
Angle Offset	0		
Spatial Radius	3		6
Temporal Radius	1		2
Component Velocity Threshold	0.8		
Optical Flow Threshold	0.8		
Maximum Pyramid Level	3	4	3
Postprocessing Maximum Magnitude	24 (absolute)	48 (absolute)	24 (absolute)
WVM Filter Window Size	3		
WVM Filter Minimum Certainty	75%		
WVM Filter Number of Passes	2		
Accuracy Minimum Threshold	51%		
Accuracy Border Skip	15		

Table A.23 Parameter sets for the Urban 2 sequence.

Algorithm	Density	Fleets Angular		Magnitude		Absolute Angular	
	[%]	μ [°]	σ [°]	μ	σ	μ [°]	σ [°]
Pyramid Algorithm 1	93.6	40.857	35.940	4.164	5.424	43.439	43.469
+Postprocessing	75.9	36.391	33.265	3.218	4.120	38.844	41.460
Pyramid Algorithm 2	96.9	34.097	36.121	4.943	8.266	34.267	43.214
+Postprocessing	86.8	**28.559**	32.083	3.765	4.879	28.208	39.359
Pyramid Algorithm 3	88.1	43.629	36.900	4.664	5.846	45.111	44.720
+Postprocessing	65.6	38.485	34.329	3.621	4.296	39.476	42.647
Basic Algorithm	88.4	57.431	42.224	6.498	7.145	60.077	50.186
+Postprocessing	62.9	**52.872**	39.887	5.601	6.845	55.344	48.234
Lucas-Kanade Extended 1	7.8	46.209	32.833	7.790	7.879	55.060	47.640
+Postprocessing	3.4	44.821	28.951	6.723	8.131	58.573	48.275
Lucas-Kanade Extended 2	4.5	44.826	32.750	7.343	7.714	54.980	48.966
+Postprocessing	1.5	41.492	26.050	5.498	7.467	58.068	49.387
Lucas-Kanade Extended 3	0.1	35.666	18.801	1.486	2.876	62.170	49.908
+Postprocessing	0.0	41.431	20.984	1.092	0.788	56.476	37.582
Lucas-Kanade Extended 4	100.0	60.788	38.229	6.406	7.198	67.420	48.528
+Postprocessing	60.7	**54.494**	31.839	4.927	6.749	64.058	47.656
Lucas-Kanade 1	100.0	56.799	39.423	5.965	6.775	62.783	48.908
+Postprocessing	98.2	54.301	37.702	5.974	6.723	59.874	47.652
Lucas-Kanade 2	100.0	53.114	37.656	6.006	6.799	58.248	47.141
+Postprocessing	99.2	51.823	36.666	6.032	6.786	56.633	46.336
Lucas-Kanade 3	100.0	50.642	36.257	6.079	6.872	55.045	45.815
+Postprocessing	99.6	**49.804**	35.568	6.092	6.855	53.950	45.210
Lucas-Kanade Pyramid 1	99.8	29.017	30.649	3.649	7.532	31.915	39.613
+Postprocessing	96.7	27.218	29.241	3.074	5.010	30.046	38.606
Lucas-Kanade Pyramid 2	100.0	21.689	23.374	2.916	4.982	23.267	32.385
+Postprocessing	99.3	21.071	22.798	2.777	4.696	22.612	31.911
Lucas-Kanade Pyramid 3	100.0	18.926	20.727	2.661	4.461	19.911	29.550
+Postprocessing	99.9	**18.692**	20.476	2.613	4.363	19.676	29.343
Horn-Schunck	100.0	49.851	35.042	5.988	6.752	53.929	45.004
+Postprocessing	99.5	**48.815**	34.329	6.052	6.833	52.449	44.284

Table A.24 Results for the Urban 2 sequence.

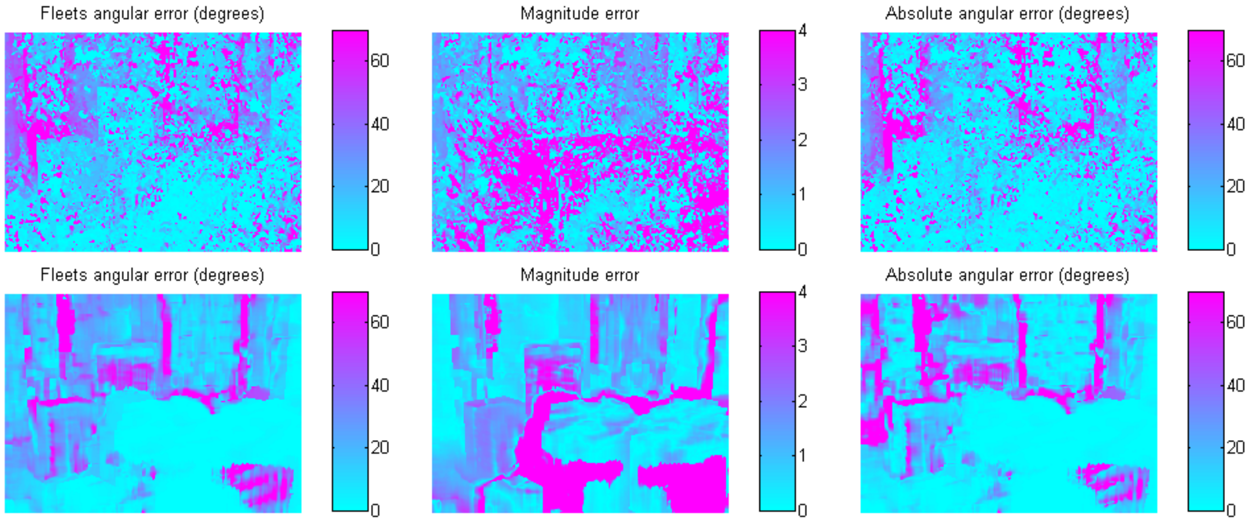


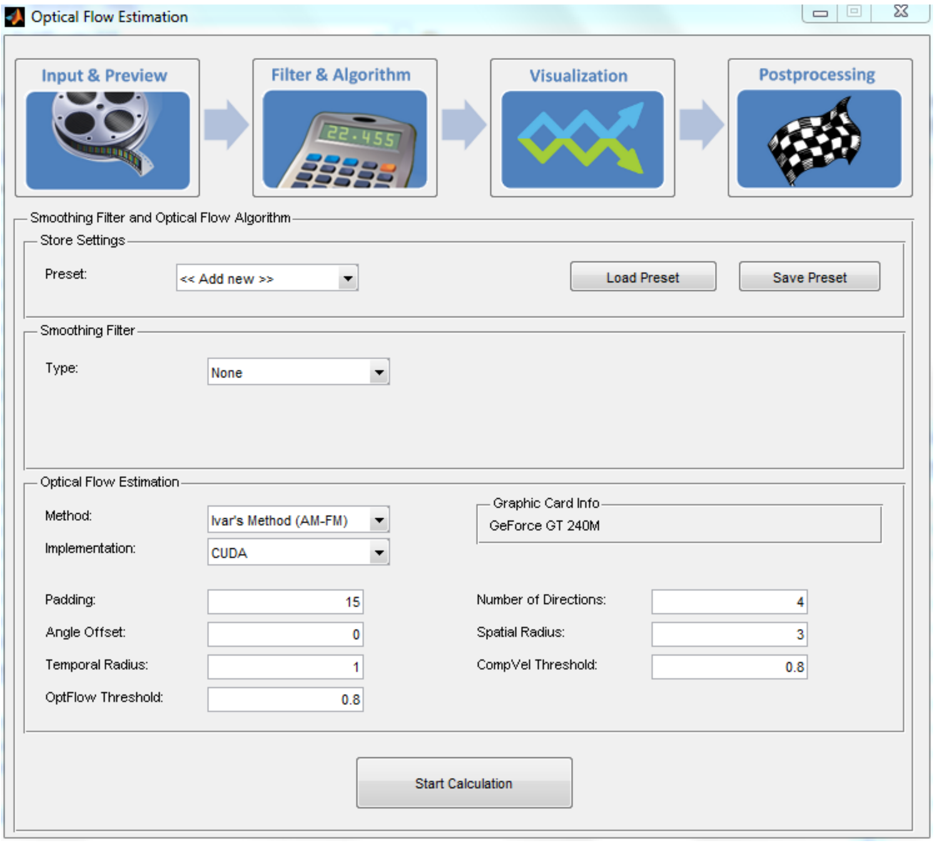
Figure A.40 Error plots for the Urban 3 sequence, Pyramid Algorithm 2 (top) and Lucas-Kanade Pyramid 3 (bottom).

B

Matlab GUI

Introduction The implementation and usage of the Matlab GUI is explained in [3]. This appendix describes some of the extensions made during this project.

New Algorithms Figures B.1 and B.2 show the input masks for the Basic Algorithm and the Pyramid Algorithm. The parameters are described in section 5.4.1 and 5.5.1, respectively.



Smoothing Filter and Optical Flow Algorithm

Store Settings

Preset: << Add new >>

Smoothing Filter

Type: None

Optical Flow Estimation

Method: Ivar's Method (AM-FM)

Implementation: CUDA

Graphic Card Info: GeForce GT 240M

Padding: 15

Angle Offset: 0

Temporal Radius: 1

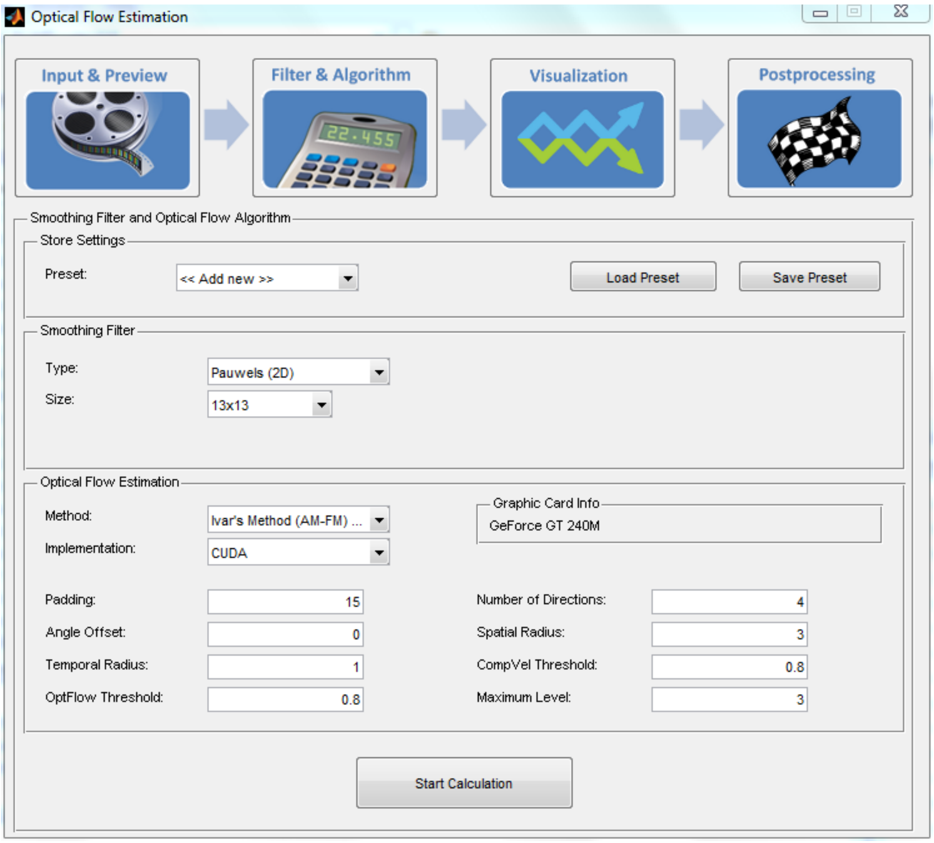
OptFlow Threshold: 0.8

Number of Directions: 4

Spatial Radius: 3

CompVel Threshold: 0.8

Figure B.1 GUI settings for the Basic Algorithm.



Smoothing Filter and Optical Flow Algorithm

Store Settings

Preset: << Add new >>

Smoothing Filter

Type: Pauwels (2D)

Size: 13x13

Optical Flow Estimation

Method: Ivar's Method (AM-FM) ...

Implementation: CUDA

Graphic Card Info

GeForce GT 240M

Padding:	15	Number of Directions:	4
Angle Offset:	0	Spatial Radius:	3
Temporal Radius:	1	CompVel Threshold:	0.8
OptFlow Threshold:	0.8	Maximum Level:	3

Figure B.2 GUI settings for the Pyramid Algorithm.

Smoothing Filter Three new smooting filter types were implemented:

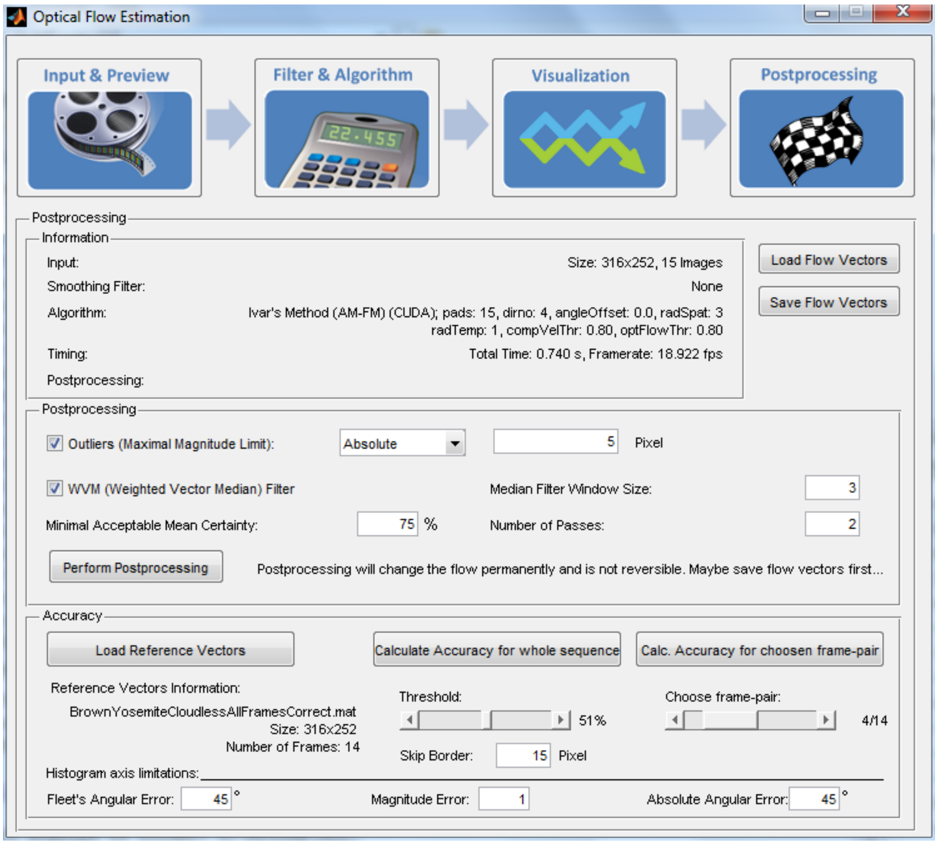
None This setting is for algorithms which don't support a smoothing filter (e.g. Basic Algorithm).

Pauwels (2D) A non-separable 2D filter type for using with the Pyramid Algorithm. Two sizes are available, 13 × 13 and 17 × 17.

Dummy-Filter (2D) This setting can be used with any algorithm that supports separable 2D filters. It behaves like no filter.

Postprocessing A new option for the postprocessing to skip the border regions from the accuracy measurement was added (see figure B.3).

Output to Workspace Each time the accuracy of a frame-pair is computed, all parameters and results are collected in a structure and added to the *QualitySummaries* vector in the Base Workspace. This makes it easier to collect data for tables and figures.



Postprocessing

Information

Input: Size: 316x252, 15 Images

Smoothing Filter: None

Algorithm: Ivar's Method (AM-FM) (CUDA); pads: 15, dirno: 4, angleOffset: 0.0, radSpat: 3 radTemp: 1, compVelThr: 0.80, optFlowThr: 0.80

Timing: Total Time: 0.740 s, Framerate: 18.922 fps

Postprocessing:

Postprocessing

☑ Outliers (Maximal Magnitude Limit): Absolute 5 Pixel

☑ WVM (Weighted Vector Median) Filter

Median Filter Window Size: 3

Minimal Acceptable Mean Certainty: 75 %

Number of Passes: 2

Postprocessing will change the flow permanently and is not reversible. Maybe save flow vectors first...

Accuracy

Reference Vectors Information:
BrownYosemiteCloudlessAllFramesCorrect.mat
Size: 316x252
Number of Frames: 14

Threshold: 51%

Choose frame-pair: 4/14

Skip Border: 15 Pixel

Histogram axis limitations:

Fleet's Angular Error: 45 °

Magnitude Error: 1

Absolute Angular Error: 45 °

Figure B.3 Postprocessing page of the Matlab GUI.

C

Test Image Sequences

Introduction

This appendix introduces the image sequences that were used to test and compare the algorithms developed during this project. It basically reproduces the appendix A of [3]. Further information on some of the sequences can be found in [18].

The sequences made available by the Middlebury College were not used in the previous project. More information about these sequences can be found in [12]. These sequences contain higher velocities than the other ones and were therefore used to test the multiresolution scheme.

Name: Ettlinger Tor

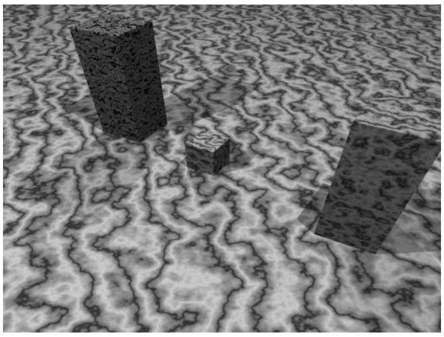
Type: Real

Filename: UKAettlingerTorImageData.mat

Source: `http://i21www.ira.uka.de/image_sequences/`

Dimensions: 512 × 512 × 50

Description: Traffic intersection sequence recorded at the Ettlinger-Tor in Karlsruhe by a stationary camera. [3]



Name: Marbled Block

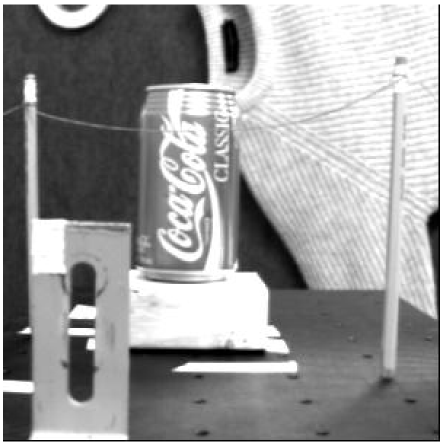
Type: Synthetic

Filename: UKAUKAmarbleBlockImageData.mat

Source: `http://i21www.ira.uka.de/image_sequences/`

Dimensions: 384 × 512 × 201

Description: Polyhedral scene with two moving marbled blocks and stationary camera. [3]



Name: NASA

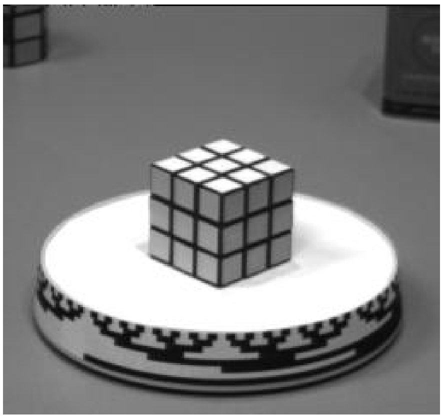
Type: Real

Filename: UWOnasaImageData.mat

Source: `ftp://ftp.csd.uwo.ca/pub/vision/TESTDATA/`

Dimensions: $300 \times 300 \times 37$

Description: The NASA sequence is primarily dilational; the camera moves along its line of sight toward the Coke can near the centre of the image. Image velocities are typically less than $1\,pixel/frame$. [18]



Name: Rubik Cube

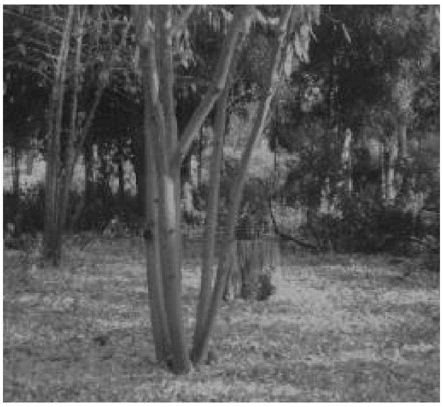
Type: Real

Filename: UWOrubicImageData.mat

Source: `ftp://ftp.csd.uwo.ca/pub/vision/TESTDATA/`

Dimensions: $240 \times 256 \times 21$

Description: Rotating Rubik cube on a microwave turntable. [3]



Name: SRI Trees

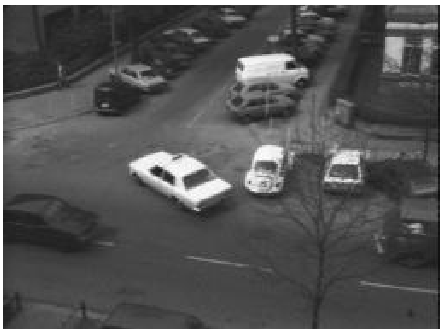
Type: Real

Filename: UWOsriTreesImageData.mat

Source: `ftp://ftp.csd.uwo.ca/pub/vision/TESTDATA/`

Dimensions: $233 \times 256 \times 21$

Description: The camera translates parallel to the ground plane, perpendicular to its line of sight, in front of clusters of trees. Velocities are as large as $2\,pixels/frame$. [18]



Name: Taxi

Type: Real

Filename: UWOtaxiImageData.mat

Source: `ftp://ftp.csd.uwo.ca/pub/vision/TESTDATA/`

Dimensions: $190 \times 256 \times 21$

Description: Traffic sequence showing a taxi in Hamburg. [3]

Name: Diverging Tree

Type: Synthetic

Filename: UWOtreeDivImageData.mat

Source: `ftp://ftp.csd.uwo.ca/pub/vision/TESTDATA/`

Dimensions: $150 \times 150 \times 40$

Description: The camera moves along its line of sight; the focus of expansion is at the centre of the image, and image speeds vary from 1.29 $pixels/frame$ on left side to 1.86 $pixels/frame$ on the right. [18]

Name: Translating Tree

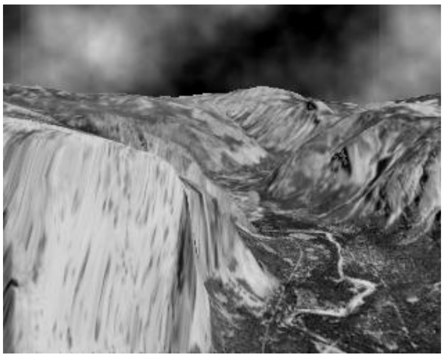
Type: Synthetic

Filename: UWOtreeTransImageData.mat

Source: `ftp://ftp.csd.uwo.ca/pub/vision/TESTDATA/`

Dimensions: $150 \times 150 \times 40$

Description: The camera moves normal to its line of sight along its x-axis, with velocities all parallel with the image x-axis, with speeds between 1.73 $pixels/frame$ and 2.26 $pixels/frame$. [18]



Name: Yosemite

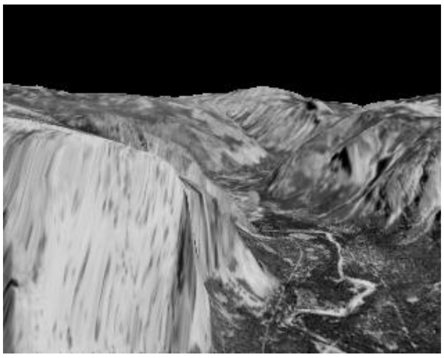
Type: Synthetic

Filename: BrownYosemiteCloudyImageData.mat

Source: `http://www.cs.brown.edu/~black/images.html`

Dimensions: $252 \times 316 \times 15$

Description: The motion in the upper right is mainly divergent, the clouds translate to the right with a speed of 1 $pixel/frame$, while velocities in the lower left are about 4 $pixels/frame$. [18]



Name: Yosemite Cloudless

Type: Synthetic

Filename: BrownYosemiteCloudlessImageData.mat

Source: `http://www.cs.brown.edu/~black/images.html`

Dimensions: $252 \times 316 \times 15$

Description: Yosemite sequence without clouds.

Name: Beanbags

Type: Real

Filename: MiddleburyBeanbagsImageData.mat

Source: `http://vision.middlebury.edu/flow/`

Dimensions: $480 \times 640 \times 8$

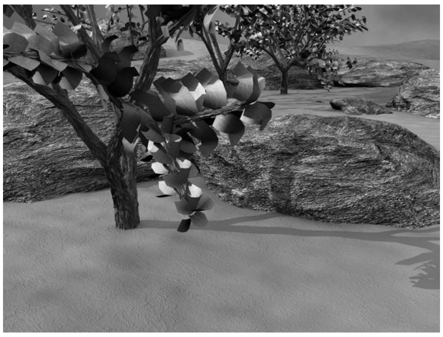
Name: Dog Dance

Type: Real

Filename: MiddleburyDogDanceImageData.mat

Source: `http://vision.middlebury.edu/flow/`

Dimensions: $480 \times 640 \times 8$



Name: Grove 2

Type: Synthetic

Filename: MiddleburyGrove2ImageData.mat

Source: `http://vision.middlebury.edu/flow/`

Dimensions: $480 \times 640 \times 8$

Name: Grove 3

Type: Synthetic

Filename: MiddleburyGrove3ImageData.mat

Source: `http://vision.middlebury.edu/flow/`

Dimensions: $480 \times 640 \times 8$

Name: Hydrangea

Type: Real

Filename: MiddleburyHydrangeaImageData.mat

Source: `http://vision.middlebury.edu/flow/`

Dimensions: 388 × 584 × 8

Name: Mini Cooper

Type: Real

Filename: MiddleburyMiniCooperImageData.mat

Source: `http://vision.middlebury.edu/flow/`

Dimensions: 480 × 640 × 8

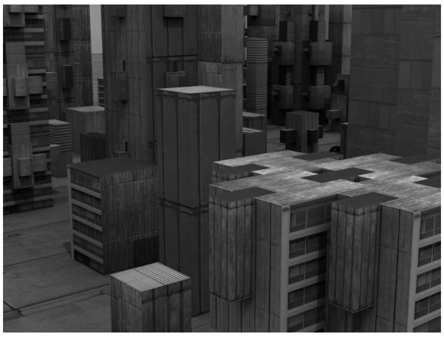
Name: Rubber Whale

Type: Real

Filename: MiddleburyRubberWhaleImageData.mat

Source: `http://vision.middlebury.edu/flow/`

Dimensions: 388 × 584 × 8



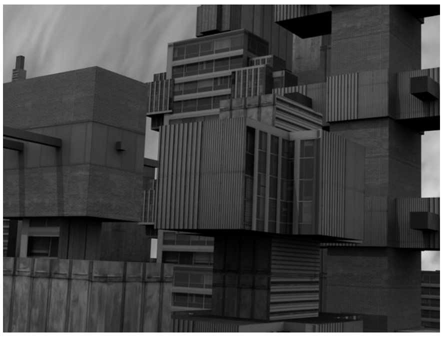
Name: Urban 2

Type: Synthetic

Filename: MiddleburyUrban2ImageData.mat

Source: `http://vision.middlebury.edu/flow/`

Dimensions: 480 × 640 × 8



Name: Urban 3

Type: Synthetic

Filename: MiddleburyUrban3ImageData.mat

Source: `http://vision.middlebury.edu/flow/`

Dimensions: 480 × 640 × 8

Name: Walking

Type: Real

Filename: MiddleburyWalkingImageData.mat

Source: `http://vision.middlebury.edu/flow/`

Dimensions: 480 × 640 × 8

D

OpenCV

Introduction The Matlab GUI developed during the previous project [3] lets the user select three algorithms from the OpenCV library [25]. For this project, these libraries (which are not provided as Win64 binaries by the developers) as well as the Matlab wrappers (*.mex* files) had to be ported to run on a Win64 system.

D.1 Building the Libraries

Introduction This section describes how to build the OpenCV library from its sources for Win32 and Win64 targets (see also [33]).

Prerequisites The following software was used to build OpenCV:

- OpenCV 2.1 sources[1]
- CMake 2.8.1[2]
- Microsoft Visual Studio 2008 SP1

Build Process The following steps were used to build OpenCV. Steps 3 to 9 were done twice, once for the Win32 binaries and once for the Win64 binaries.

1. Unpack the sources to *C:\OpenCV-2.1.0*
2. Start CMake and enter *C:\OpenCV-2.1.0* as source path
3. Enter *C:\OpenCV-2.1.0\Win32* or *C:\OpenCV-2.1.0\Win64* as build path for the binaries
4. Press *Configure*
5. Select *Visual Studio 9 2008* or *Visual Studio 9 2008 Win64* as generator
6. Use the default settings, but set *BUILD_NEW_PYTHON_SUPPORT* to *false*
7. Press *Configure* again
8. Press *Generate*
9. Open the solution *C:\OpenCV-2.1.0\Win32\OpenCV.sln* or *C:\OpenCV-2.1.0\Win64\OpenCV.sln* with Visual Studio and compile it (Release)

[1] `http://sourceforge.net/projects/opencvlibrary/files/`

[2] `http://www.cmake.org/cmake/resources/software.html`

D.2 Building and Using the MEX Wrappers

Win64 Port To build the *.mex* files for a Win64 system (*.mexw64*), the original makefiles and the project configurations had to be modified in a similar way as described in section 5.2. Also some datatypes had to be changed to allow a safe passing of pointers on Win32 and Win64 systems.

Prerequisites For compiling, the OpenCV header files and the *.lib* files are needed. These files can be generated as described in section D.1. But the relevant subdirectories for Win32 and Win64 can also be found packed in a *.zip* file on the disc of this project, so that there is no need to build OpenCV from sources again.

For the build process, the environmental variables listed in table D.1 must be created (restart computer afterwards, if necessary).

Variable	**Value**
`MATLAB_DIR`	*C:\Program Files\MATLAB\R2009b*
`OPENCV_ROOT`	*C:\OpenCV-2.1.0*
`OPENCV_WIN32`	*C:\OpenCV-2.1.0\Win32*
`OPENCV_WIN64`	*C:\OpenCV-2.1.0\Win64*

Table D.1 Required environment variables.

Runtime At runtime, the OpenCV *.dll* files are needed. Therefore, one of the following directories must be added to the Path environmental variable (depending on the current system):
C:\OpenCV-2.1.0\Win32\bin\Release or *C:\OpenCV-2.1.0\Win64\bin\Release*

Cross Compiling Building a *.mexw32* on a Win64 system and vice versa is not directly possible since Matlab installs only the libraries for the current platform. In this project, the *.mexw32* files were built by first copying the libraries for a 32-bit Matlab installation to the following directory:
C:\Program Files\MATLAB\R2009b\extern\lib\win32\microsoft

C++ Runtime To run the OpenCV implementations of the algorithm on a system with no Visual Studio installed, it might be necessary to install the *Microsoft Visual C++ 2008 Redistributable Package*.

E

Content of the DVD

Root
In the root of the DVD, the PDFs of this report, the poster, and of the original project description can be found. Furthermore, it contains the project planning as an Excel file.

Literature
The *Literature* directory contains the PDFs of almost all the references in the bibliography.

Source Code
The *Source Code* directory contains all the Matlab and C sources. There are three important entry points:

- *.\GUI\MainOpticalFlow.m*
  is the main file for the Matlab GUI.
- *.\MTcode\Algorithms\CUDA\AmFmCUDA\AmFmCUDA.sln*
  is the solution file for the CUDA implementation of the Basic Algorithm.
- *.\MTcode\Algorithms\CUDA\AmFmPyrCUDA\AmFmPyrCUDA.sln*
  is the solution file for the CUDA implementation of the Pyramid Algorithm.

Test Sequences
The *Test Sequences* directory contains the test image sequences described in appendix C and their ground truths (if available). All the sequences and ground truths are available in their original file format and as *.mat* files which can be loaded from the Matlab GUI.

Misc Code, Mathematica
The *Misc Code* and *Mathematica* directories contain some *.m* files and Mathematica notebooks which were used to verify certain steps of the development of the algorithms.

Software
The *Software* directory contains the installers for CUDA and Parallel Nsight. It also contains a *.zip* file with the Win32 and Win64 binaries of OpenCV 2.1 (see appendix D).

Results
The *Results* directory contains *.mat* files with all the information (complete statistics; parameters used) for every data point in chapter 6 and appendix A. The directory also contains some Matlab scripts to extract the values shown in this report from the *.mat* files.

Bibliography

Books

[1] **Gösta H. Granlund, Hans Knutsson**
Signal Processing for Computer Vision
Kluwer Academic Publishers, 1995

[2] **Bernd Jähne, Horst Haußecker, Peter Geißler**
Handbook of Computer Vision and Applications
Volume 2 - Signal Processing and Pattern Recognition
Academic Press, 1999

Theses

[3] **Fabian Braun, Marc Länzlinger**
Efficient Implementation and Evaluation of Methods for Estimating Optical Flow in Video
Bachelor Thesis, 2009
University of Applied Sciences Rapperswil, Switzerland and University of Stavanger, Norway

[4] **Ivar Austvoll**
Motion Estimation using Directional Filters
PhD Thesis, 1999
Stavanger College, Norway

[5] **Mathias Johansson**
The Hilbert Transform
Master Thesis, 1999
Växjö University, Sweden

Papers

[6] **Espen Kristoffersen**
AM-FM Signal Modelling in Motion Estimation
Dept. of Electrical and Computer Engineering, Stavanger University College, Norway

[7] **Espen Kristoffersen, Ivar Austvoll, Kjersti Engan**
Dense Motion Field Estimation Using Spatial Filtering and Quasi Eigenfunction Approximations
IEEE International Conference on Image Processing ICIP, 2005

[8] **Ivar Austvoll**
Directional Filters and a New Structure for Estimation of Optical Flow
IEEE International Conference on Image Processing ICIP, 2000

[9] **Ivar Austvoll**
A Study of the Yosemite Sequence Used as a Test Sequence for Estimation of Optical Flow
Lecture Notes in Computer Science, Springer, 2005

[10] **Ivar Austvoll**
Motion Analysis and Estimation Using Directional Filters and Orientation Tensors
Pattern Recognition and Image Analysis, Vol. 15, 2005

[11] **Simon Baker, Daniel Scharstein, J.P. Lewis, Stefan Roth, Michael J. Black, Richard Szeliski**
A Database and Evaluation Methodology for Optical Flow
IEEE International Conference on Computer Vision ICCV, 2007

[12] **Simon Baker, Daniel Scharstein, J.P. Lewis, Stefan Roth, Michael J. Black, Richard Szeliski**
A Database and Evaluation Methodology for Optical Flow
Technical Report, Microsoft Corporation, 2009

[13] **Pierre Bayerl, Heiko Neumann**
A Fast Biologically Inspired Algorithm for Recurrent Motion Estimation
IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 29, No. 2, 2007

[14] **Thomas Brox, Andrés Bruhn, Nils Papenberg, Joachim Weickert**
High Accuracy Optical Flow Estimation Based on a Theory for Warping
European Conference on Computer Vision, Springer LNCS, 2004

[15] **Andrés Bruhn, Joachim Weickert, Christoph Schnörr**
Lucas/Kanade Meets Horn/Schunck: Combining Local and Global Optic Flow Methods
International Journal of Computer Vision, Vol. 61, No. 3, Springer, 2005

[16] **Javier Díaz, Eduardo Ros, Francisco Pelayo, Eva M. Ortigosa, Sonia Mota**
FPGA-Based Real-Time Optical-Flow System
IEEE Transactions on Circuits and Systems for Video Technology, Vol. 16, No. 2, 2006

[17] **Peter Sand, Seth Teller**
Particle Video: Long-Range Motion Estimation using Point Trajectories
IEEE Computer Vision and Pattern Recognition CVPR, 2006

[18] **J.L. Barron, D.J. Fleet, S.S. Beauchemin**
Performance of Optical Flow Techniques
International Journal of Computer Vision, Vol. 12, No. 1, Springer, 1994

[19] **Eric J. Pauwels, Luc J. Van Gool, Peter Fiddelaers, Theo Moons**
An Extended Class of Scale-Invariant and Recursive Scale Space Filters
IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 17, No. 7, 1995

[20] **Victor Podlozhnyuk**
Image Convolution with CUDA
NVIDIA, 2007

Manuals

[21] *NVIDIA CUDA Programming Guide*
NVIDIA, Version 2.3.1, 2009

[22] *NVIDIA CUDA C Programming Best Practices Guide*
NVIDIA, Version 2.3, 2009

[23] *NVIDIA CUDA Reference Manual*
NVIDIA, Version 3.0 Beta, 2009

[24] *NVIDIA CUDA-GDB User Guide*
NVIDIA, Version 3.0, 2010

[25] *Open Source Computer Vision Library - Reference Manual*
Intel Corporation, 2001

Web Pages

[26] *Primitive Datentypen in C*
19.01.2010
`http://www.tnotes.de/CPrimitiveDatentypen`

[27] *How do I update MEX-files to use the large array handling API (-largeArrayDims)?*
19.01.2010
`http://www.mathworks.com/support/solutions/en/data/1-5C27B9/?solution=1-5C27B9`

[28] *C/C++ Source MEX-Files*
10.03.2010
`http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/f43721.html`

[29] *Rotation (mathematics)*
Wikipedia, 07.03.2010
`http://en.wikipedia.org/wiki/Rotation_(mathematics)`

[30] *Analytic Signal*
Wikipedia, 18.04.2010
`http://en.wikipedia.org/wiki/Analytic_signal`

[31] **Gady Agam**
Introduction to programming with OpenCV
Department of Computer Science, Illinois Institute of Technology, 2006
`http://www.cs.iit.edu/~agam/cs512/lect-notes/opencv-intro/index.html`

[32] **Robert Laganière**
Programming Computer Vision Applications
VIVA Lab, University of Ottawa, 2008
`http://www.site.uottawa.ca/~laganier/tutorial/opencv+directshow/cvision.htm`

[33] *OpenCV Installation Guide*
OpenCV Wiki, 05.05.2010
`http://opencv.willowgarage.com/wiki/InstallGuide`