FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	As-Rigid-As-Possible Optical Flow Estimation on the GPU using Dual Numbers
Student:	Bc. Boris Laskov
Supervisor:	prof. Ing. Daniel Sýkora, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2019/20

Instructions

Explore the state-of-the-art in the field of optical flow estimation [1, 2, 3, 4]. Focus namely on methods which can preserve local rigidity of the resulting deformation field [5]. Formulate non-linear energy minimization problem which allows estimating as-rigid-as-possible optical flow between two consecutive video frames. Solve the resulting non-linear energy using L-BFGS method [6] with automatic differentiation based on dual numbers [7]. Optimize the solution so that the optical flow can be estimated on HD resolution at interactive rates using currently available graphics cards. Verify functionality of the final implementation on a set of video sequences supplied by the thesis supervisor. Compare the results with several concurrent optical flow estimation methods.

References

[1] Lucas & Kanade: An Iterative Image Registration Technique with an Application to Stereo Vision, Proceedings of IUW, pp. 121-130, 1981.
[2] Glocker et al.: Dense Image Registration Through MRFs And Efficient Linear Programming, Medical Image Analysis 12(6):731-741, 2008.
[3] Liu et al.: SIFT Flow: Dense Correspondence across Scenes and its Applications, IEEE Transactions on Pattern Analysis and Machine Intelligence, 33(5):978-994, 2011.
[4] Sun et al.: PWC-Net: CNNs for Optical Flow Using Pyramid, Warping, and Cost Volume, Proceedings of CVPR, pp. 8934-8943, 2018.
[5] Sýkora et al.: As-Rigid-As-Possible Image Registration for Hand-drawn Cartoon Animations, Proceedings of NPAR, pp. 25-33, 2009.
[6] Liu & Nocedal: On the Limited Memory BFGS Method for Large Scale Optimization, Mathematical Programming 45(3):503-528, 1989.
[7] Piponi: Automatic Differentiation, C++ Templates, and Photogrammetry, Journal of Graphics Tools 9(4):41-55, 2004.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 7, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

As-Rigid-As-Possible Optical Flow Estimation on the GPU using Dual Numbers

Bc. Boris Laskov

Department of software engineering

Supervisor: prof. Ing. Daniel Sýkora, Ph.D.

April 21, 2019

Acknowledgements

I would like to thank Daniel Sýkora for all the guidance and valuable advice during the work on this thesis. I am also thankful to Ondřej Jamriška for the help in understanding the optical flow estimation algorithm he has developed. Last but not least, I am grateful to MAUR film studio for providing a video sequence on which we can test various optical flow estimation methods.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for non-profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on April 21, 2019 .....

Czech Technical University in Prague

Faculty of Information Technology

Citation of this thesis

Laskov, Boris. *As-Rigid-As-Possible Optical Flow Estimation on the GPU using Dual Numbers*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstract

The problem of optical flow estimation is among fundamental objectives in the field of computer vision. During the past three decades, various solutions to it have been proposed – from classic algorithms to neural nets.

In this thesis, we describe several methods and frameworks for optical flow estimation. We focus on the two algorithms that explicitly preserve local rigidity of the resulting flow field. In addition to providing the detailed description of the algorithms proper and a few related auxiliary concepts, we optimize the performance of a particular method – one of those that keep the flow locally rigid – by implementing it to run on the GPU.

In the theoretical part, we study several optical flow estimation algorithms and describe GPU-specific optimizations for one of them. In the practical part, we evaluate and compare the quality of results produced by each of the reviewed methods. We also measure computational speed of the GPU-based version of the selected algorithm and compare it with the performance of the default sequential implementation. The testing is done on a real production video.

Keywords Optical flow estimation, image registration, rigid transformation, dual numbers, GPGPU, CUDA

Abstrakt

Výpočet optického toku je jedním ze základních úkolů v oblasti počítačového vidění. Během posledních tří desetiletí byla navržena různá řešení – od klasických algoritmů až po neuronové sítě.

V této práci popisujeme několik metod k výpočtu optického toku. Zaměřujeme se na dva algoritmy, které explicitně zachovávají lokální tuhost výsledného vektorového pole. Kromě toho, že poskytujeme podrobný popis samotných algoritmů a souvisejících matematických nástrojů, optimalizujeme navíc výkon jedné vybrané metody s využitím GPU.

V teoretické části práce rozebíráme několik algoritmů pro výpočet optického toku a popisujeme specifické pro GPU optimalizací jednoho z nich. V praktické části hodnotíme a porovnáváme kvalitu výsledků získaných každou z prozkoumaných metod. Také měříme výpočetní rychlost GPU verze vybraného algoritmu a porovnáváme ji s rychlostí výchozí sekvenční implementace. Testování se provádí na reálném produkčním videu.

Klíčová slova Výpočet optického toku, registrace obrazu, rigidní transformace, duální čísla, GPGPU, CUDA

Contents

List of Figures

List of Tables

Introduction

Optical flow estimation is one of essential and highly practical objectives in the field of computer vision. Having two consecutive video frames with similar objects and overall context, we try to construct a flow field: for every pixel in the first frame, we want to find a displacement vector that defines its new position in the second one. Such a vector field represents dense correspondence between the frames. It can be used to accomplish various visual effects–related tasks: for instance, we can change the speed of the video or introduce new transitions for the filmed objects [1].

In this thesis, our primary goal is to map a separately created texture onto an object from a video. Painting every single frame by hand is a very time-consuming process, especially for complex scenes. But if we can capture the movement and deformations of the object across multiple frames, we can apply this knowledge to automatically warp the texture together with it. Even if we occasionally have to manually adjust the mapping or create several keyframes by hand, the ability to track the object at least for relatively short periods of time can still greatly reduce the amount of manual labor for the artists.

In various other applications of optical flow estimation, additional challenges often arise. For example, objects to be tracked in a scene may be unforeknowable. However, this is not our particular case: we can afford to create several keyframes, properly paint them, and extract the mask for the object so the algorithm knows what exactly it should compute optical flow for. We do not expect fast motion in the video. And, in most cases, we even have a green screen as the background. Moreover, we consider objects with local rigidity. On the one hand, this assumption limits the variety of transformations we can use, making some algorithms (to be covered in the following chapters) unsuitable for tracking e.g. fluids. But on the other one, these constraints may help to improve the overall quality of estimated optical flow for locally rigid bodies.

One of our main measures of quality is tracking precision. We should preserve the texture with its fine details even if it is warped with the flow estimated during longer video sequences. In our tests throughout this thesis, we use a sequence of about 200 frames. We try to map the texture from the first frame onto all subsequent ones. We take the last synthesized frame and see how well it looks (since the more frames we use to estimate optical flow, the more chances the flow field has to become corrupted). We compare it with the last real frame from the video.

Another important aspect is computational time required to complete the estimation. The faster the algorithm works, the simpler and more effectively one can work with it. That is why we will do our best to optimize the method producing the most visually satisfying results for maximum performance. We will try to do it on the GPU and see how well this problem is suited for a massively parallel implementation.

Chapter outline

In Chapter 1, we will provide a high-level overview of a few state-of-the-art methods for optical flow estimation. In Chapter 2, we will have a more in-depth look at several algorithms that do not explicitly preserve local rigidity of the flow field. We will describe how they work and evaluate their results on our test sequence. Chapter 3 will introduce a method that does preserve local rigidity but is not particularly suited for precise object tracking across longer video sequences. Its improved version will be presented in Chapter 4. In Chapter 5, we will list all the optimizations that make it run faster on the GPU and test its speed in comparison with the default CPU version. Finally, in Chapter 6 we will numerically evaluate the quality of results obtained with all algorithms from Chapters 2 to 4.

Chapter 1

Related work

This chapter is devoted to a brief overview of several optical flow estimation algorithms that can be currently considered to be state of the art. Their exploration is not part of the assignment, so we will omit a detailed discussion of them. Instead, we will provide a summary of the main ideas these methods are built upon.

We rely on the benchmark available at [2]. We have taken 5 published algorithms with the lowest average endpoint error, which represents the norm of the difference between ground-truth flow vectors and the ones computed by an algorithm.

We could not find the article with the top algorithm according to the benchmark. However, judging by its name and the list of authors, this missing study is very closely related to the method that is at the fifth place, which is described four paragraphs below. We will start with the algorithm holding the second place.

PM-PM: PatchMatch With Potts Model for Object Segmentation and Stereo Matching
The method proposed by Xu et al. in [3] aims to estimate correspondence field for a pair of images by jointly solving problems of image segmentation and stereo matching. The authors note that these tasks are mutually informative: abrupt changes in depth can indicate boundaries of objects and vice versa. Thus, solving one problem can help to find the solution for the other one. Image segmentation is performed with the multi-label Potts Model, and stereo matching is done through a modified PatchMatch algorithm. As the last step, an occlusion handling strategy is applied, which is based on minimum spanning trees.

Optical Flow via Locally Adaptive Fusion of Complementary Data Costs

When optical flow is estimated on the basis of only one dissimilarity measure, the quality of the resulting flow field can be insufficient. That is why Kim et al. in [4] propose to minimize the data term defined as a weighted combination of different discrepancy functions, which may be based on different assumptions (e.g. brightness or gradient constancy). Weights are locally adaptive, so the ratio of contributions of various measures can be unique at each pixel. The compound error function also has the data discriminability term which is used to evaluate the local "goodness" of individual measures. Both flow vectors and weights are subject to regularization. The article also contains a study about how to appropriately choose the set of dissimilarity functions.

Motion Detail Preserving Optical Flow Estimation

An important disadvantage of the traditional coarse-to-fine strategy (see Section 2.1.2) is that the removal of high-frequency information on coarser levels can prevent an algorithm from correctly estimating optical flow. Xu et al. in [5] proposes a refinement for this strategy that allows to capture both small and large motion more precisely. On each level of the pyramid, it performs sparse (SIFT-based) feature matching together with dense patch matching to improve the quality of the flow that comes from the previous (coarser) level. The approximated flow is refined by minimizing the error function. Its data term includes constraints based on both color and gradient constancy. Their weights are binary, so the algorithm can appropriately switch between them but not use them simultaneously. The study also contains the discussion of the efficient minimization strategy and occlusion handling.

Large Displacement Optical Flow from Nearest Neighbor Fields

The study by Chen et al. [6] appears to be very close to the missing one we have talked about earlier. It focuses on the approximate computation of the initial flow field which then serves as an input for another optical flow estimation algorithm. Many traditional methods start with initial vectors equal to zero. But the authors of [6] observe that a sufficiently good approximation of the flow field can help classic algorithms obtain much better results. In particular, it is useful when processing large displacements: other approaches often rely on the coarse-to-fine strategy, which removes high-frequency information that can be crucial for finding the right displacements. The solution is to precompute a better initial field. The authors construct a nearest neighbor field by taking patches from the first image and searching for their most similar counterparts in the second one. This allows to establish dense correspondence with approximately precise displacements. Based on this field, motion patterns are identified – they are used for motion segmentation. Finally, the quality of local flow is improved with a separate optical flow estimation framework.

Fast optical flow by component fusion
We could not find details on this algorithm (it holds the sixth place). The following method is from the seventh line of the benchmark.

Joint trilateral filtering for multiframe optical flow
Similarly to [5], the method developed by Stoll et al. [7] focuses on improving the quality of optical flow that comes from coarser levels when using coarse-to-fine strategy. But in this study, the authors employ a different approach: they use a trilateral filter to remove noise and outliers before setting the flow as the input for estimation process at a finer level. A trilateral filter takes into account spatial distance together with differences in colors and flow vectors. Moreover, a cross bilateral upsampling is used to scale the obtained flow to match the next pyramid level – it improves the accuracy of the upscaled vector field by preserving edges. Another aspect of this study is that the error function is minimized for five consecutive frames at once.

The rest of the present thesis will be written in accordance with its assignment. In the next chapter, we will focus on the more classic algorithms for optical flow estimation we are explicitly tasked to explore.

Chapter 2

Estimating non-rigid optical flow

In this chapter, we will review several methods that can be considered state of the art in optical flow estimation. We will describe them and provide examples of results that can be achieved with these algorithms and frameworks.

None of them explicitly try to keep the flow locally rigid. An algorithm that has been specifically designed to preserve the rigidity is discussed in Chapter 3.

2.1 Lucas–Kanade image registration technique

Initially, the technique developed by Lucas and Kanade [8] has been designed for image registration. Let us briefly describe what this task is about.

Suppose we have two images of an object. These images are not identical, e.g. they may be created under different lighting conditions, at different times, from different viewpoints, etc. The aim of image registration is to find a transformation that we can apply to the first image so that the object depicted in it will become properly aligned with itself in the second image [9]. In other words, we must find the object from the first image inside the second one together with parameters of the corresponding transformation, such as a relative offset by which the two locations of the object differ.

The Lucas–Kanade method can be used not only for this objective but it may also be easily extended for optical flow estimation. We will see this in action after going through the details of the original formulation proposed for the image registration task.

This section will be mainly based on [8] – the paper which this method has been proposed in.

2.1.1 Basic algorithm

We have two images: scene F and object G. We want to find offset $\boldsymbol{h}$ that shows the location of G inside F. In case of 2D images, $\boldsymbol{h}$ is a two-element vector*. Applying it should minimize the function of dissimilarity in the region where images overlap. To measure discrepancy, we use the sum of squared errors:

$$E(\boldsymbol{h}) = \sum_{\boldsymbol{x}\in\Omega} \left[F(\boldsymbol{x}+\boldsymbol{h}) - G(\boldsymbol{x})\right]^2, \tag{2.1}$$

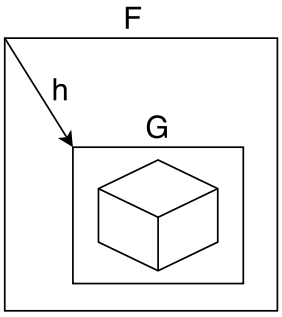


Figure 2.1: Locating object G in scene F

where Ω is the set of pixel coordinates that form our region of interest (either the whole G or a part of it with the object) and $\boldsymbol{x}^T = [x, y]$ is a vector denoting the current pixel location inside G. The actual "error" is simply the difference in colors (intensities) between the two pixels ($F(\boldsymbol{x}+\boldsymbol{h})$ and $G(\boldsymbol{x})$).

There are various ways to minimize this error function. We will follow the one from the original paper. We can take advantage of the following approximation:

$$F(\boldsymbol{x}+\boldsymbol{h}) \approx F(\boldsymbol{x}) + \boldsymbol{h}^T \nabla F(\boldsymbol{x}), \tag{2.2}$$

where $\nabla F(\boldsymbol{x})$ is the gradient of the scene. The gradient is a vector of partial derivatives in all directions of the given function – in case of 2D images, there are two directions (axes), x and y. With images being discrete functions, the derivatives cannot be computed exactly, but we can fall back on one of the many approximations. For example, the following difference of intensities of adjacent pixels works sufficiently well in most cases:

$$\nabla F(\boldsymbol{x}) = \begin{bmatrix} \nabla_x F(\boldsymbol{x}) \\ \nabla_y F(\boldsymbol{x}) \end{bmatrix} \approx \begin{bmatrix} F(\boldsymbol{x}+[1,0]) - F(\boldsymbol{x}) \\ F(\boldsymbol{x}+[0,1]) - F(\boldsymbol{x}) \end{bmatrix}.$$

To minimize the error function from Equation (2.1), we plug Equation (2.2) into it and differentiate it with respect to $\boldsymbol{h}$. Then we set it to be equal to 0.

*Note that for vectors we use bold font to distinguish them from scalar variables. We also assume column notation for vectors by default, so $\boldsymbol{x} = \left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right]$ and $\boldsymbol{x}^T = [x, y]$. While reading formulas, it may help, for instance, to determine the result of vector–vector multiplications: $\boldsymbol{x}^T\boldsymbol{x}$ is a scalar and $\boldsymbol{x}\boldsymbol{x}^T$ is a matrix.

We obtain:

$$\begin{aligned}
&\frac{\partial}{\partial \boldsymbol{h}} \sum_{\boldsymbol{x}\in\Omega} [F(\boldsymbol{x}+\boldsymbol{h}) - G(\boldsymbol{x})]^2 \\
\approx &\frac{\partial}{\partial \boldsymbol{h}} \sum_{\boldsymbol{x}\in\Omega} \left[F(\boldsymbol{x}) + \boldsymbol{h}^T \nabla F(\boldsymbol{x}) - G(\boldsymbol{x})\right]^2 \\
= &\sum_{\boldsymbol{x}\in\Omega} 2\left[F(\boldsymbol{x}) + \boldsymbol{h}^T \nabla F(\boldsymbol{x}) - G(\boldsymbol{x})\right] \nabla F(\boldsymbol{x}) \\
= &\, 0.
\end{aligned}$$

Simplifying and rearranging, as well as using properties of transposed matrix multiplication and the fact that $\nabla F(\boldsymbol{x}) \nabla F(\boldsymbol{x})^T$ ends up being a symmetric matrix, we get

$$\begin{aligned}
\sum_{\boldsymbol{x}\in\Omega} \nabla F(\boldsymbol{x}) \nabla F(\boldsymbol{x})^T \boldsymbol{h} &= \sum_{\boldsymbol{x}\in\Omega} \nabla F(\boldsymbol{x}) \left(G(\boldsymbol{x}) - F(\boldsymbol{x})\right) \\
\mathbf{A}\boldsymbol{h} &= \boldsymbol{b} \\
\boldsymbol{h} &= \mathbf{A}^{-1}\boldsymbol{b}.
\end{aligned} \tag{2.3}$$

For this system to have a solution, the inverse of 2×2 matrix $\mathbf{A}$ must exist. Usually, it is not a problem, but sometimes it is better to test this explicitly – using eigenvalues, for example.

This will give us offset $\boldsymbol{h}$. The algorithm is iterative, so we should repeat the process of finding the next $\boldsymbol{h}$ from the currently computed new position. We should proceed until convergence – that is, until the latest value of $\boldsymbol{h}$ we get becomes small enough. This can be tested with Euclidean (L^2) norm. All values of $\boldsymbol{h}$ in between the iterations are accumulated together to give us the new starting position.

The Lucas–Kanade method has the following important properties:

- The algorithm is iterative: we have to repeatedly compute and accumulate similar things. We have to detect convergence to know when to stop.
- It works well when the initial displacement of G in relation to the true object location in F is small enough. Otherwise, the window with G may "float away" – the algorithm may diverge.
- The more similar the object looks in both F and G, the more chances the algorithm has to reach convergence and find the right position.

2.1.2 Simple extensions

Weighting scheme
We can apply weights to improve the precision and convergence rate. The original article suggests taking values of F and G derivatives into account: when calculating the difference in colors $(G(\boldsymbol{x}) - F(\boldsymbol{x}))$, we can multiply each individual result by a scalar which shows how well the gradients of F and G match each other in the given point. The closer is the match, the greater value the color difference should be multiplied by. This will increase its significance in the sum. Every component of the right-hand-side sum (vector $\boldsymbol{b}$) should be multiplied accordingly. This will give us a weighted sum instead of an ordinary one we have had earlier.

We can calculate a certain weight w_i for the color difference between i-th F and G pixels in the region of their current overlap with the help of the following formula*:

$$w_i = \frac{1}{\|\nabla G(\boldsymbol{x}) - \nabla F(\boldsymbol{x})\|}.$$

Then, we should rewrite matrix $\mathbf{A}$ and right-hand-side vector $\boldsymbol{b}$ from Equation (2.3) as

$$\mathbf{A} = \sum_{\boldsymbol{x}\in\Omega} \nabla F(\boldsymbol{x}) \nabla F(\boldsymbol{x})^T \quad \rightarrow \quad \mathbf{A} = \sum_{\boldsymbol{x}\in\Omega} w_i \nabla F(\boldsymbol{x}) \nabla F(\boldsymbol{x})^T,$$

$$\boldsymbol{b} = \sum_{\boldsymbol{x}\in\Omega} \nabla F(\boldsymbol{x}) \left(G(\boldsymbol{x}) - F(\boldsymbol{x})\right) \quad \rightarrow \quad \boldsymbol{b} = \sum_{\boldsymbol{x}\in\Omega} w_i \nabla F(\boldsymbol{x}) \left(G(\boldsymbol{x}) - F(\boldsymbol{x})\right).$$

This will give us weighted sums, where color differences of pixels with similar gradient behavior will have greater impact on the value of offset $\boldsymbol{h}$.

Grayscale and RGB images
The described algorithm naturally supports multichannel images. For example, with grayscale images, the value of $F(\boldsymbol{x})$ is a scalar, while with 3-channel RGB images it is a vector of 3 elements. Similarly, the gradient of grayscale $F(\boldsymbol{x})$ is a two-element vector, while it is a 2×3 matrix in case of an RGB image. Multiplications inside each element of the sum that finally form matrix $\mathbf{A}$ and vector $\boldsymbol{b}$ involve computing dot product with vectors; due to this fact, they result in scalars, so the shapes of $\mathbf{A}$ and $\boldsymbol{b}$ remain the same (2×2 and 2×1 respectively).

*In practice, the result of this formula should be constrained by a fixed upper bound to prevent it from hitting infinity values in case of a perfect match.

Blurring images before computation
It is often a good idea to smooth F and G prior to running the Lucas–Kanade algorithm on them. Although this process can suppress some important high-frequency information, it can help to "spread" otherwise large but isolated gradient values over some neighborhood. Then, the algorithm can properly capture the right direction earlier and start moving along it with each iteration.

Multiresolution (also known as multigrid or a pyramid)
While blurring the image may be required for the algorithm to converge, the final position may turn out to be less exact than it could be. We can mitigate this issue by implementing a so-called coarse-to-fine strategy:

1. We create a set (a pyramid) for each image by decreasing its resolution several times. Alternatively, instead of changing the resolution, we can use Gaussian blur with different values of parameter σ.
2. We start with the blurriest F and G we have and run the algorithm. We acquire an approximate value of $\boldsymbol{h}$.
3. Having obtained this approximation, we refine it by using images with more high-frequency information – the next ones from their pyramids.
4. We repeat this process until we reach the images of original size or level of detail.

In this way, we can achieve precise results with more confidence that the algorithm will correctly converge.

2.1.3 Example

In this example, we are trying to find the neon sign of Radio City Music Hall on its exterior. The orange frame shows the current position of sign G in scene F. Figure 2.2a illustrates the initial location that we start with: it partially overlaps with the desired region but is still not as exact as we would like it to be. Figure 2.2b presents the result of running the Lucas–Kanade algorithm: it has converged after only 14 iterations and found the desired location precisely.

This example is rather simple; we show it to demonstrate what results of the algorithm can look like. The object (neon sign) has been cut out from the scene, so the only thing that differs is its location; other parameters of the scene and the object are the same (lighting, viewpoint, noise, etc.). The only extension we have used here is smoothing the image with Gaussian blur – no weighting schemes or pyramids have been applied. Studying the impact of different parameters and extensions on the Lucas–Kanade algorithm goes beyond the scope of this thesis.

(a) (b)

Figure 2.2: Aligning "Radio City" neon sign. (a) Initial position of the sign in the scene. (b) Position refined by the Lucas–Kanade technique.

2.1.4 Transformation extension

Previously, we have estimated only offset $\boldsymbol{h}$ for the object inside the scene. But the Lucas–Kanade method can be extended to find more complex transformations. In this section, we will show how to improve it to handle affine transformations, which may include translation, rotation, scaling and shearing [1]. An affine transformation includes 6 parameters in total and can be represented with a 2×3 matrix

$$\mathbf{M} = \begin{bmatrix} 1+p_1 & p_3 & p_5 \\ p_2 & 1+p_4 & p_6 \end{bmatrix}. \tag{2.4}$$

Here, we have added 1 to p_1 and p_4 to receive an identity transformation when all parameters are equal to zero.

To apply an affine transformation to a 2D point $\boldsymbol{x}$, we should extend it with the third element equal to 1 and left-multiply it by matrix $\mathbf{M}$:

$$\boldsymbol{x} = \begin{bmatrix} x \\ y \end{bmatrix} \quad \rightarrow \quad \boldsymbol{x} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\mathbf{M} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

Let us take Equation (2.1) and replace the translation with an affine transformation. This will be the new error function we should minimize:

$$E(\mathbf{M}) = \sum_{\boldsymbol{x}\in\Omega} \left[F(\mathbf{M}\boldsymbol{x}) - G(\boldsymbol{x})\right]^2.$$

We can then differentiate this equation with respect to our 6 parameters. We use the way described in [10]. Omitting the major part of math approximations and transformations, the brief sequence of steps is:

$$\begin{aligned}
&\frac{\partial}{\partial \boldsymbol{p}} \sum_{\boldsymbol{x}\in\Omega} \left[F(\mathbf{M}\boldsymbol{x}) - G(\boldsymbol{x})\right]^2 \\
&\approx \frac{\partial}{\partial \boldsymbol{p}} \sum_{\boldsymbol{x}\in\Omega} \left[F(\boldsymbol{x}) + \nabla F(\boldsymbol{x})^T \frac{\partial W}{\partial \boldsymbol{p}}\boldsymbol{p} - G(\boldsymbol{x})\right]^2 \\
&= \sum_{\boldsymbol{x}\in\Omega} 2\left(\nabla F(\boldsymbol{x})^T \frac{\partial W}{\partial \boldsymbol{p}}\right)^T \left[F(\boldsymbol{x}) + \nabla F(\boldsymbol{x})^T \frac{\partial W}{\partial \boldsymbol{p}}\boldsymbol{p} - G(\boldsymbol{x})\right] \\
&= 0,
\end{aligned} \tag{2.5}$$

where $\boldsymbol{p}$ is a vector of all our parameters ($\boldsymbol{p}^T = [p_1, p_2, p_3, p_4, p_5, p_6]$), W is a warping function, which simply applies an affine transformation $\mathbf{M}$ to a given 2D point ($W(\boldsymbol{x}) = \mathbf{M}\boldsymbol{x}$) and $\frac{\partial W}{\partial \boldsymbol{p}}$ is its derivative with respect to 6 parameters, which ends up being

$$\frac{\partial W}{\partial \boldsymbol{p}} = \begin{bmatrix} x & 0 & y & 0 & 1 & 0 \\ 0 & x & 0 & y & 0 & 1 \end{bmatrix}.$$

Expressing $\boldsymbol{p}$ from Equation (2.5) and obtaining its value makes one iteration. We then add all values from $\boldsymbol{p}$ to our current affine transformation matrix, set the new starting position with it and continue iterating. Similarly to the case of a simple offset we have described earlier, we can detect convergence by taking the L^2 norm of the freshly acquired "delta" parameters and deciding whether it is sufficiently small or not.

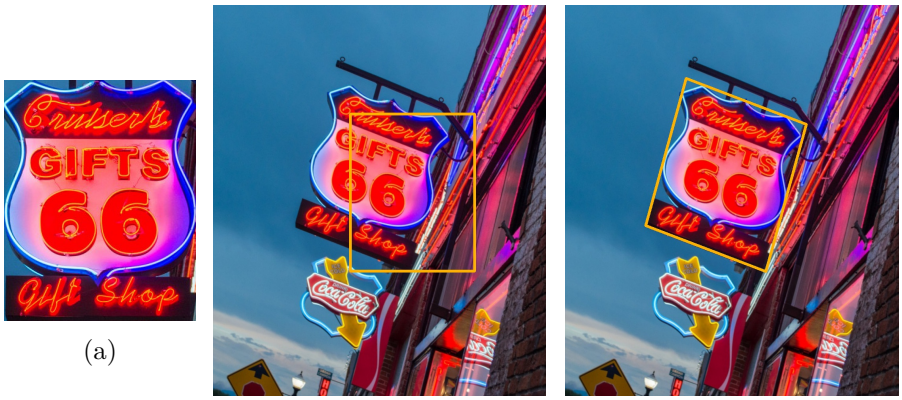
2.1.5 Example

Let us take a look at another example of registration with the Lucas–Kanade technique. After cutting out the sign we want to locate, we rotated and sheared the scene. We have also started with some offset from the correct position, so all parameters of the resulting affine transformation will have to

be altered. The initial matrix had p_5 and p_6 set to the values of the starting translation, p_1–p_4 were set to zero. Since we have added 1 in Equation (2.4) explicitly, we had the initial transformation with translation only:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & h_x \\ 0 & 1 & h_y \end{bmatrix}.$$

As you can see in Figure 2.3, the algorithm has correctly computed not only the translation, but the complete affine transformation. Input images have been pre-blurred (the same way as in Section 2.1.3).



(a) (b) (c)

Figure 2.3: Aligning "Gifts 66" neon sign. (a) Original sign we have been looking for. (b) Initial position of the sign in the scene. (c) Position refined by the Lucas–Kanade technique.

2.1.6 Estimating optical flow

Let us finally move from the task of image registration to the main purpose of studying the Lucas–Kanade technique. In order to estimate optical flow by using this algorithm, we should perform registration for every pixel from the region of interest:

- For every pixel, we create a square window around it.
- We register this window in the other frame.
- The obtained transformation is valid only for the current pixel, which is in the middle of the window.

Thus, we use image registration to estimate the optical flow by simply repeating the steps from the original algorithm.

To enhance the quality of the resulting flow field, we can additionally apply a weighting scheme to each of the search windows. By assigning the central pixel the biggest weight, we will increase its importance during the process of registration. All other pixels will get smaller values: the farther away they are located, the smaller their weights should be. We can use a 2D Gaussian kernel (or rather a discrete approximation of it) to get a weighted sum.

2.1.6.1 Computing optical flow in the video

To estimate optical flow for the whole video sequence, we create a **Flow** matrix with the size of one frame. Every element is defined as $Flow[i, j] = [i, j]$. This matrix represents initial flow field where every pixel stays in its original place.

We compute optical flow "backwards": between the second and the first frame, then between the third and the second one, etc. Once we obtain the flow field for another pair of images (Δ**Flow**), we look where its pixel offsets point to inside the "main" **Flow** matrix. We then replace the offsets in this two-frames-local ("delta") optical flow with global values by taking 4 corresponding locations from the **Flow** matrix and bilinearly interpolating between them:

$$\Delta Flow[i, j] \leftarrow \text{interpolate}(Flow[\Delta Flow[i, j]]), \quad \forall i, j \in \Delta\mathbf{Flow}$$

By doing this, we create a flow matrix for the current frame relative to the very first one.

With this newly computed flow, we reconstruct the texture for the current frame from the first one in our sequence. Then, the previous "global" **Flow** matrix gets discarded and replaced by the current one (with interpolated values):

$$\mathbf{Flow} \leftarrow \Delta\mathbf{Flow}$$

We are ready to continue applying the Lucas–Kanade method to the next pair of video frames.

2.1.7 Example

Let us show an example of estimating optical flow with the help of the Lucas–Kanade method. We have run it on our test video sequence. Figure 2.4a is the last frame from the original video and Figure 2.4b presents its synthesized version which was obtained by mapping the texture taken from the first frame – we have used the optical flow estimated with the Lucas–Kanade algorithm to warp it. The size of the window has been set to 17, the differences in colors have been weighted with a 2D discrete approximation of Gaussian kernel. We have also applied multiresolution extension.

(a) (b)

Figure 2.4: Estimating optical flow with the Lucas–Kanade method. (a) The last frame from the original video. (b) The same frame synthesized with the texture from the first frame warped by optical flow. Original frame (a): © MAUR film.

We can see that, on the one hand, the algorithm has performed well: we can still distinguish original shapes, and the details have been preserved (at least to some extent). But on the other hand, the texture looks "washed out" and is distorted far more than we would like it to be. This clearly shows the main issue with the conventional Lucas–Kanade technique: it does not take mutual positioning of individual pixels into account. Unfortunately, we cannot fix it with *any* of the parameters we have mentioned throughout this section like blurring, weighting, etc.

Skipping ahead, we will try to correct it and preserve rigidity in Chapter 4.

2.1.8 Summary

According to the results of our tests, the Lucas–Kanade technique cannot produce sufficiently good results when applied to extended video sequences, so we will continue studying other algorithms.

However, there exist various cases in which the Lucas–Kanade method can be successfully applied. For instance, it performs well when tracking the "coarse" position of an object in a scene (without precisely describing its inner transformations). Together with the support for various extensions, this algorithm can serve as an inspiration or even as a building block for more sophisticated and complicated image processing approaches.

2.2 Image registration through MRFs and LP*

This image registration technique was originally proposed by Glocker et al. [11] It was developed with a view to process medical images; however, it is not limited to this particular field.

As its name suggests, the algorithm uses the concept of Markov random fields to formulate the minimization task. We will provide a short summary of the related statistical terminology. A detailed discussion of image registration through random fields is omitted since it goes beyond the scope of this thesis. Nevertheless, the introduction below should clarify the most important aspects.

This section is based mainly on [11] and [12] – the latter is a good source for those who wish to get to the level of fine details.

2.2.1 A brief introduction to the terminology

In order to better understand the principles of the algorithm, we provide a short list of the main terms and definitions:

Random experiment, outcome, event
Every experiment has a predefined set of its possible outcomes. An event is defined by a subset of outcomes and has a certain probability to occur. If an event is based on a set of a single outcome, we can call it an elementary event.

To illustrate, suppose that we have a fair dice. The experiment is to roll it and see which value comes up. The set of outcomes consists of integers from 1 to 6. The experiment of rolling a dice can lead to some events we should define beforehand such as:

- the result is less than or equal to 3 (based on {1, 2, 3} outcomes);
- the result is equal to 6 (based on {6} outcome – an elementary event).

Random variable
It is a variable associated with a set of possible outcomes (or elementary events) that can happen as a result of a random experiment. Once we conduct the experiment, its result (or the numeric equivalent which the outcome is mapped to) will become the value of the variable. We assume that we know the set of all possible outcomes beforehand, as well as the fact that all of them are theoretically possible (have non-zero probability).

If we represent a roll of a dice with a random variable, the result is naturally an integer from 1 to 6 (unless we remap the resulting numbers in some other way). If we flip a coin, then strictly speaking we should assign numbers to the

*These acronyms will be explained in Section 2.2.1.

"heads" and "tails" outcomes, one of which will end up being taken by the random variable.

Label
A label is the second name for an elementary event. Finding out which outcome a random variable will take is the same as finding a label to assign to this variable.

Random fields
A random field is a graph-like structure. Like every graph, it has a set of nodes and a set of edges between them. The key points are:

- every node is a random variable;
- an edge between two nodes shows that the two random variables depend on each other in terms of the values they will finally take;
- all nodes that are connected to the current node with an edge form its neighborhood.

Labeling of a random field
When every random variable in a field has a label assigned, then the set of all those labels together with corresponding random variables is called a labeling.

Markov random field (MRF)
It is a random field that satisfies the following:

$$p(\boldsymbol{x}) > 0, \quad \forall \boldsymbol{x} \in \mathcal{X}, \tag{2.6}$$

$$p(x_i \mid \{x_j : j \in \mathcal{N}(i)\}) = p(x_i \mid \{x_j : j \in V \setminus \{i\}\}) \tag{2.7}$$

Equation (2.6) ensures that all possible labelings we can construct have non-zero probability ($\boldsymbol{x}$ is a concrete labeling from the set of all possible ones $\mathcal{X}$). Equation (2.7) shows that a variable can depend only on its neighbors – if two variables are not connected with an edge, they are independent ($\mathcal{N}(i)$ is the neighborhood of node i and V is the set of all nodes).

Linear programming
Linear programming is a method of optimization that helps to minimize the given cost function, where additional constraints on the solution are expressed as linear functions. Linear programming is used in [11] as a way to find the best deformation by minimizing the error function defined with the help of an MRF.

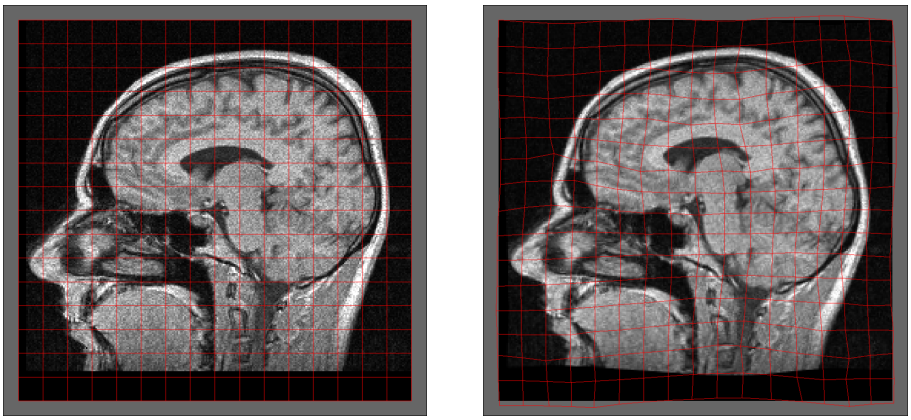
Fast-PD
This method is described in [13]. It is a specific algorithm that is used to minimize the energy of the given MRF. PD stands for primal–dual approach:

we solve two problems, the primal one and the dual one. The primal problem deals with minimization – with each iteration it gets closer to the optimal minimum providing a new upper bound for it. The dual problem is also solved iteratively, but it it approaches the solution from the other side shifting the lower bound higher and higher. We know that the real solution lies within these bounds. Thus, when the upper and lower bounds come close enough to each other, we can stop iterating and take the current primal problem solution as a good approximation to the real minimum.

2.2.2 The algorithm

One of the key ideas of this image registration technique is to reduce the dimensionality of the problem. Instead of letting every pixel have an independent offset, we move only a set of control points. This approach leaves us with fewer parameters to optimize.

We construct a two-dimensional grid over the image (depicted in red in Figure 2.5). It has significantly fewer vertices than the image has pixels. Each vertex becomes a control point, and each pixel depends on the combination of the displacements of nearby vertices. We look for an optimal offset for every vertex so that the total error function over the image is minimized.



(a) (b)

Figure 2.5: An example of a grid created to reduce dimensionality of a registration problem (picture from the reference implementation available at [14]). (a) Original grid. (b) Deformed grid after registration.

At the same time, this grid can be viewed as a Markov random field. Vertices then become random variables; they can take values from a predefined discrete set of displacement vectors. Our task is to find an optimal labeling.

The transformation of an individual pixel can be written as

$$\mathcal{T}(\boldsymbol{x}) = \boldsymbol{x} + \mathcal{D}(\boldsymbol{x}), \quad \mathcal{D}(\boldsymbol{x}) = \sum_{\boldsymbol{p} \in G} \eta(|\boldsymbol{x} - \boldsymbol{p}|)\boldsymbol{d}^{l_{\boldsymbol{p}}}.$$

Here, η is a weight function that helps to select only nearby control points for the given pixel $\boldsymbol{x}$, $\boldsymbol{p}$ is a control point and $\boldsymbol{d}^{l_{\boldsymbol{p}}}$ is a displacement vector currently associated with it.

The error function to minimize depends on the chosen labeling $\boldsymbol{l}$ (the set of displacement vectors assigned to the nodes of the grid):

$$E_{MRF}(\boldsymbol{l}) = \underbrace{\sum_{\boldsymbol{p} \in G} V_{\boldsymbol{p}}(l_{\boldsymbol{p}})}_{\text{data term}} + \underbrace{\sum_{\boldsymbol{p} \in G} \sum_{\boldsymbol{q} \in \mathcal{N}(\boldsymbol{p})} V_{\boldsymbol{pq}}(l_{\boldsymbol{p}}, l_{\boldsymbol{q}})}_{\text{smoothness term}}. \tag{2.8}$$

Data term represents the sum of differences between colors of pixels with the current grid deformation. It can be estimated as

$$V_{\boldsymbol{p}}(l_{\boldsymbol{p}}) \approx \int_{\Omega} \hat{\eta}(|\boldsymbol{x} - \boldsymbol{p}|) \cdot \rho_h(G(\boldsymbol{x}), F(\mathcal{T}(\boldsymbol{x}))),$$

where Ω is the region of interest (the whole image G or a part of it), $\hat{\eta}$ is a weight function to select the area around given control point $\boldsymbol{p}$, and ρ_h is a dissimilarity function: for instance, we can choose it to be the SSD (the sum of squared differences, just what we have seen in Section 2.1.1), the SAD (the sum of absolute differences), or anything else suitable in the particular case.

Smoothness term (spatial regularization) is based on the assumption that neighbor control points should have similar displacement and is defined as

$$V_{\boldsymbol{pq}}(l_{\boldsymbol{p}}, l_{\boldsymbol{q}}) = \lambda_{\boldsymbol{pq}} \left| \boldsymbol{d}^{l_{\boldsymbol{p}}} - \boldsymbol{d}^{l_{\boldsymbol{q}}} \right| \tag{2.9}$$

with $\boldsymbol{p}$ and $\boldsymbol{q}$ being neighbor control points (defined as $\boldsymbol{q} \in \mathcal{N}(\boldsymbol{p})$ in Equation (2.8) and shown in Figure 2.6) and λ is a weight, which should be chosen to suit a particular image registration problem and a dissimilarity function. The term is intended to prevent too destructive grid deformations (provided that pieces of the scene may change in appearance, and ideal mapping with zero error is not always possible). This is where the fact about dependencies between adjacent random variables of an MRF becomes meaningful.

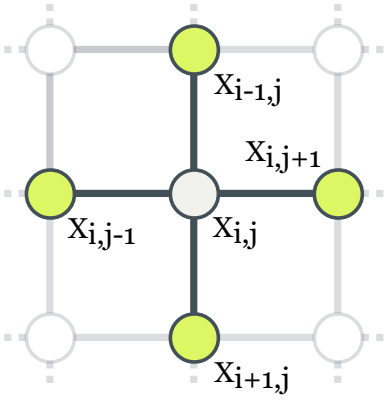


Figure 2.6: Four-node neighborhood $\mathcal{N}(X_{i,j})$ of random variable $X_{i,j}$

2.2.3 Minimization

The minimization of Equation (2.8) is done with the help of linear programming and the Fast-PD algorithm [11, 13]. One of the key advantages of this strategy is that it is gradient-free. It means we do not have to estimate gradient values, which can be difficult with discrete functions and the variety of available approximations. We will omit a detailed discussion of this topic since it goes beyond the scope of this thesis.

An important limitation of this approach is the fact that we have only a discrete set of displacement vectors to choose from. Its size can be adjusted with a separate parameter, which denotes the number of samples taken between the zero offset and the maximum one. Adding a new sample causes a new outer loop to appear. This can negatively impact the performance of the method. So, instead of estimating gradient values, we have to find an optimal maximum for all labels and a suitable number of discrete offsets in between.

2.2.4 Extensions

The behavior of the algorithm can be adjusted in many ways – from choosing different values of parameters to plugging in various functions. The latter are used, for instance, to measure dissimilarity between colors in the data term and offset vectors in the smoothness term. The complete list of parameters the reference implementation uses can be viewed in its user guide at [14].

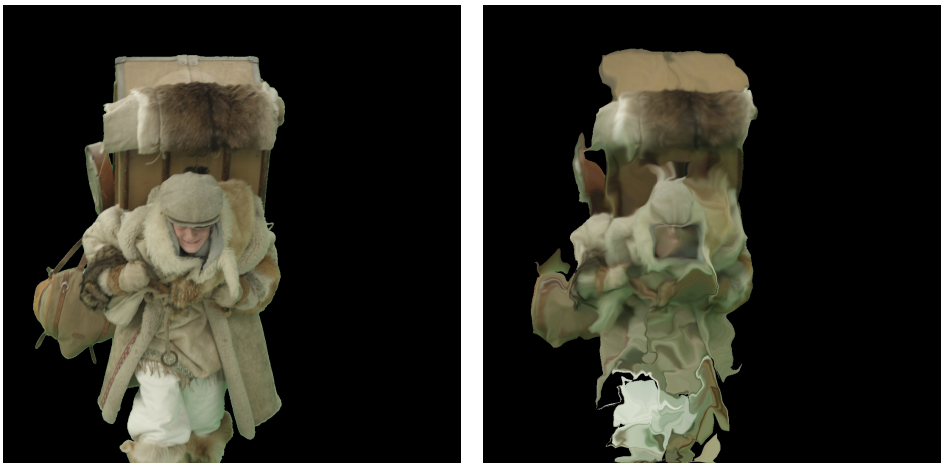
Coarse-to-fine strategy is also present but is not limited to the image resolution. The size of grid cells can also be consecutively reduced. Although the number of control points to label increases, it helps to settle the deformation grid more precisely. And earlier iterations do not have to deal with a large number of nodes.

2.2.5 Example

Because of the complexity of this image registration method, we have not coded it ourselves; instead, we have used a reference implementation available at [14]. The approach of combining "global" optical flow from two-frame-local values has been the same as the one described in Section 2.1.6.1: we have computed deformations for every pair of video frames and then combined them to reach the coordinates in the very first frame.

As you can see in Figure 2.7, in our object-in-video-tracking task the results reveal the same problems as those obtained with the Lucas–Kanade technique. The texture looks "washed out" and over-smoothed with not so many details preserved. Larger deformations are also clearly visible. Playing with different parameters has not changed the situation in a noticeably positive way: the algorithm does not explicitly preserve rigidity "by design".

On the bright side, the result may appear slightly better than after flow estimation with the Lucas–Kanade method. This is probably because of the regularization term.



(a) (b)

Figure 2.7: Estimating optical flow via MRF-based image registration method by Glocker et al. (a) The last frame from the original video. (b) The same frame synthesized with the texture from the first frame warped by optical flow. Original frame (a): © MAUR film.

2.2.6 Summary

Although image registration through MRFs does not perform best in our particular task, it does not mean it cannot be successfully applied in many other cases. When local rigidity is not compulsory, it can show impressive results. Thanks to the grid, the algorithm takes into account some information about

pixel mutual positions and leaves us with far fewer parameters to optimize, which positively affects computational speed. It can be modified with many various parameters and dissimilarity measures. The gradient-free approach is an interesting alternative to more classic gradient-based methods.

However, we will try to find something else to solve our texture-mapping task.

2.3 SIFT flow

The SIFT flow algorithm was proposed by Liu et al. [15] Unlike the two methods we looked at earlier, it adds an extra layer of abstraction into the matching process: instead of directly comparing colors of pixels, it operates on scale-invariant feature transform (SIFT) descriptors created from them. We will introduce the basic idea of features and descriptors shortly.

This section is based on [15] – the original article describing SIFT flow and [1] – a book where one can find more information regarding features and their various descriptors.

2.3.1 Features and descriptors

Let us go through some concepts to better understand the topic:

Features
A feature is a region inside an image. It is centered at some pixel and captures a certain area around it. A good feature should satisfy the following properties:

- It should represent some locally-unique information – in other words, it should clearly stand out in its neighborhood.
- It should be easily and reliably trackable across different images of the same scene: that is, if we have detected it in an image, we should have no trouble also locating it in all the other images of the given scene.

Features can be also called points of interest. They can be used to find correspondence between images or to track certain points across frame sequences (in motion capture, for instance). There exist a lot of algorithms designed specifically to locate good features – they are commonly referred to as feature detectors. They include, for example, Harris corner detector, Difference-of-Gaussians (DoG) detector, and many others.

Descriptors
A descriptor is a structure used to conveniently represent a feature; usually, it is a vector of numbers. Descriptors should preserve information about features they encode. However, they should also be (at least to some extent) invariant to the transformations that can change the appearance of the given feature in different images of the scene, such as overall lighting, rotation, scale, or even

some deformations. This property enables us to efficiently match descriptors and to find a mapping between features from the first image and the second one.

When we were estimating optical flow in Section 2.1.7, we created a window around each pixel we wanted to track. Then, we were registering it in the previous image by minimizing the sum of squared differences of intensities. Ironically, the window we had can be viewed as a trivial descriptor – perhaps, the most trivial one. It neither abstracted the raw colors away from the matching process nor took possible transformations into account.

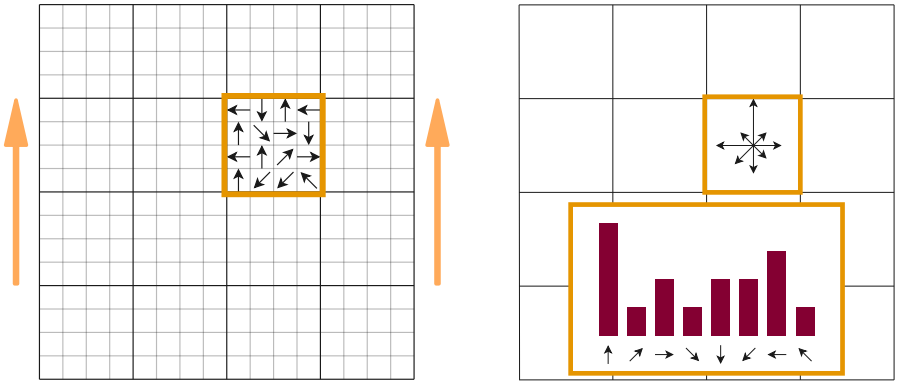
SIFT descriptors
A SIFT descriptor is a particular histogram-based descriptor proposed by Lowe in [16]. As inputs, it takes the center of the feature to describe, its estimated scale and approximate dominant orientation. It is constructed as follows:

1. We create a square image region from the center position of the feature and a multiple of its scale. The top side of the square should be aligned with the dominant orientation.
2. We rotate the given image region for the top side of the square to be up.
3. We divide the region into 16 squares (each side becomes 4 squares long).
4. For each of 16 squares, we create an 8-bin histogram. Each bin corresponds to the one of 8 gradient directions. Naturally, there are many more than just 8 gradient directions, but we coarsely quantize them to fit those 8 ones.
5. We compute values of the gradient for all pixels in every smaller square and fill their histograms with appropriate values. Prior to gradient estimation, we may additionally apply Gaussian blur to the image.
6. As the last stage, we may apply some normalizations to obtained histograms.

In the end, we have 16 squares with 8-bin gradient orientation histogram each. All these values form a SIFT descriptor – a vector of 128 elements. A simplified example can be seen in Figure 2.8.

2.3.2 The algorithm

As we have mentioned earlier, SIFT flow matches descriptors instead of raw pixel intensities. If we run feature detection first and then perform registration of the obtained features, the resulting correspondence will be *sparse* image correspondence. It is called sparse because after classic feature detection we are left with a relatively small number of good features scattered across the images. Although it can handle greater lighting changes and displacements, we



(a) (b)

Figure 2.8: An example of a SIFT descriptor. For brevity, only one square is filled with gradient directions. (a) Estimated gradient directions at every pixel. Arrows at the sides represent the dominant orientation – top side of the square is already aligned with it. (b) A histogram of gradient orientations.

will never get pixel-level correspondence using this approach. This is why in SIFT flow we entirely skip the process of feature detection. We simply create a SIFT descriptor for every pixel that we want to track and find mapping between them, thus obtaining dense correspondence.

The function we try to minimize is

$$E(w) = \underbrace{\sum_{\boldsymbol{p}\in\Omega} (\|s_1(\boldsymbol{p}) - s_2(\boldsymbol{p} + w(\boldsymbol{p}))\|_1)}_{\text{data term}} + \underbrace{\sum_{\boldsymbol{p}\in\Omega} \eta(|w_x(\boldsymbol{p})| + |w_y(\boldsymbol{p})|)}_{\text{small displacement term}} + \underbrace{\sum_{\boldsymbol{p}\in\Omega} \sum_{\boldsymbol{q}\in\mathcal{N}(\boldsymbol{p})} \alpha(\|w(\boldsymbol{p}) - w(\boldsymbol{q})\|_1)}_{\text{smoothness term}} .$$

As you can see, it consist of 3 terms:

Data term denotes the total difference accumulated across all pairs of descriptors. Here, Ω is the region of interest (all pixels we want to match), $\boldsymbol{p}$ is a two-element vector with coordinates of the current pixel, s_1 and s_2 are functions that return descriptors centered at given positions in the first and the second image respectively, and $w(\boldsymbol{p})$ is a function that returns a displacement vector for the given point.

Small displacement term should force the desired displacement vectors to be small if data and smoothness term do not provide sufficient information; η is a weight, w_x and w_y are individual elements of a displacement vector.

Smoothness term is based on the assumption that adjacent pixels should have similar offsets. It is almost identical to what we have seen in Section 2.2.2, Equation (2.9). Even the neighborhood system is the same (see Figure 2.6).

Unlike other algorithms we have discussed earlier, in SIFT flow a different norm is used to compute the difference: truncated L^1 norm. Basically, it is a standard L^1 norm constrained with an additional upper bound t:

$$\text{Truncated } L^1 \text{ norm} \coloneqq \min(t, \|\boldsymbol{a} - \boldsymbol{b}\|_1).$$

Authors state that truncated L^1 norm deals better with outliers and flow discontinuities.

2.3.3 Minimization

Authors minimize the error function with the help of the dual-layer loopy belief propagation algorithm. One of its important disadvantages (with respect to our texture-mapping task) is the fact that resulting pixel displacements can be only integers. This means that we cannot expect subpixel-level precision with SIFT flow. We will omit a complete description of this minimization strategy since in this thesis we focus more on optical flow algorithms proper than on various numerical optimization techniques.

2.3.4 Extensions

As with many other algorithms, the multigrid strategy can be applied to SIFT flow too. After several tests, the authors [15] point out that it not only speeds the minimization process up but also helps to reach lower error values (find more correct displacements) in the end.

As part of the optimization, it is possible to divide the smoothness term into two separate functions for x and y directions:

$$\sum_{\boldsymbol{p}\in\Omega}\sum_{\boldsymbol{q}\in\mathcal{N}(\boldsymbol{p})} \alpha(\|w(\boldsymbol{p}) - w(\boldsymbol{q})\|_1) =$$

$$\sum_{\boldsymbol{p}\in\Omega}\sum_{\boldsymbol{q}\in\mathcal{N}(\boldsymbol{p})} \alpha(|w_x(\boldsymbol{p}) - w_x(\boldsymbol{q})|) + \sum_{\boldsymbol{p}\in\Omega}\sum_{\boldsymbol{q}\in\mathcal{N}(\boldsymbol{p})} \alpha(|w_y(\boldsymbol{p}) - w_y(\boldsymbol{q})|).$$

This leads to a noticeable decrease of the computational complexity: from $O(L^4)$ to $O(L^2)$ for one message passing iteration (during minimization), where L is a set of possible displacement vectors.

2.3.5 Example

We have used the reference implementation of SIFT flow (written in C++ and Matlab), which is provided together with the article and can be downloaded at [15]. Once again, we have taken advantage of the composition technique described in Section 2.1.6.1 to compute optical flow.

In Figure 2.9, you can see that resulting image looks quite blocky and distorted. This is due to the fact that the algorithm produces only integer-valued displacement vectors, so the flow field does not exhibit subpixel-level precision. In addition, while the error function has a smoothness term to help regularize the flow, there is still no explicit support for keeping the field locally rigid. Thus, we are left with these unsatisfactory results.

(a) (b)

Figure 2.9: Estimating optical flow with SIFT flow. (a) The last frame from the original video. (b) The same frame synthesized with the texture from the first frame warped by optical flow. Original frame (a): © MAUR film.

2.3.6 Summary

SIFT flow does not help us to produce visually pleasing texture mapping. However, it does not mean that the algorithm is useless: it works perfectly in different optical flow–related tasks. In has some interesting properties unavailable to many other methods: it can better handle large object displacements across images, it is less sensitive to varying lighting conditions, etc.

For example, it can be used to find the most matching image to the given one (a query image). Its authors have conducted the following test: suppose, we have a database of pictures and we want to find an image that corresponds best to the query. The database contains no other images of the same scene that is depicted in the query. We retrieve the first 20 nearest neighbors with

the help of some other algorithm. Then, we apply SIFT flow, and declare the image with the lowest final value of the error function to be the best match. In this scenario, we can find a different scene with a very close semantic meaning (similar types of objects present). The results of this test are impressive (see the original article for details). Hence, the error function can be used not only to find the flow by minimizing it – its final value can be viewed as a dissimilarity measure.

2.4 PWC-Net

PWC-Net is a framework proposed by Sun et al. in [17] (also the main source of information for this section). It combines CNNs (convolutional neural networks) with classic optical flow–related routines. By the time of publishing in 2018, it had surpassed all other methods in MPI Sintel (final pass) and KITTI 2015 benchmarks, which are commonly used in evaluation of optical flow estimation algorithms.

2.4.1 The framework

PWC stands for a pyramid, warping and a cost volume – the main (but not all) components PWC-Net is assembled from. Let us go through them briefly.

In order to produce better results in less time, PWC-Net constructs a *pyramid* from each of the two given images. This helps to both capture larger motions more accurately and speed the computation up by reducing the search region for every patch. Unlike many other approaches, PWC-Net processes inputs in the feature domain: that is, it operates on feature maps generated by a CNN instead of raw color values at every pixel. When downsampling, the number of features is reduced by a factor of two. There are 6 levels in total.

Once a flow field is estimated at the coarsest level, one of the images (or, more precisely, its feature representation) on the next level is *warped* according to it. Only then the process of flow estimation is repeated, now with one of the finer images warped by an upscaled version of the flow field from the previous (coarser) level.

When we try to find the best location for the given patch from the image, we test different positions inside some neighborhood in the warped version of the other image. This produces a *cost volume* – a construct showing how well the central pixel from the patch maps to some location in the other image. It is calculated as correlation between features. The cost volume is then used as the input to CNN-based optical flow estimator, which computes the new flow field.

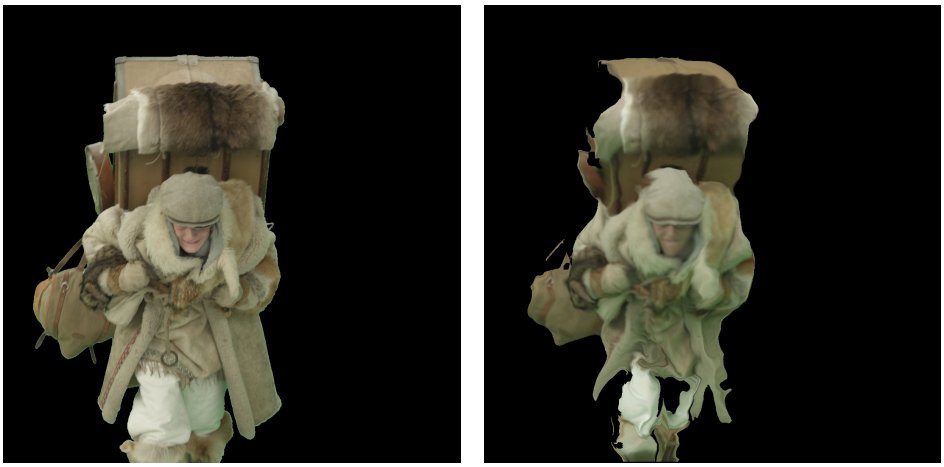
Once optical flow is estimated, it can be optionally adjusted with a special CNN – the context network. It is designed to refine the flow field with the help

of contextual information that enables better semantic segmentation (simply stated – dividing the scene into distinct objects).

Note that warping and cost volume computation are designed without any use of CNNs. Neural networks are applied to create feature pyramids of images, to extract flow from the cost volume and to optionally refine the result.

2.4.2 Example

We have tested the reference implementation of PWC-Net [18] on our video sequence. Since it works for only two consecutive frames at a time, we have once again used the approach of combining flow fields described in Section 2.1.6.1.



(a) (b)

Figure 2.10: Estimating optical flow with PWC-Net. (a) The last frame from the original video. (b) The same frame synthesized with the texture from the first frame warped by optical flow. Original frame (a): © MAUR film.

The results we have obtained (depicted in Figure 2.10) are a little controversial. On the one hand, PWC-Net has managed to preserve the texture very well in comparison to the other algorithms we have tested: it exposes many more original details and is no longer over-smoothed. On the other hand, the overall shape of the object being tracked is deformed and "squeezed". Outer boundaries are no longer in their places, and the more there are frames across which we track the object, the smaller it becomes. We have attempted to fix this in several ways: by computing and adding a "static flow" (optical flow computed by PWC-Net from a pair of identical images), blurring sharp boundaries outwards with Poisson image editing (see [1] for details), and simply applying additional hard-coded constants. Unfortunately, none of these hotfixes have worked. Even if some of them had improved the situation, we could not have called it a proper solution anyway.

2.4.3 Summary

In the original article, the authors state several facts about strengths and weaknesses of PWC-Net. For example, let us take MPI Sintel. This benchmark has two passes: the clean one and the final one. Objects in the clean pass have sharper edges, which are more accurately aligned between consecutive frames, while in the final pass these edges can be to a certain extent destroyed by motion blur, atmospheric effects, etc. The authors say that while outperforming its competition on the final pass, on the clean one, PWC-Net performs worse than some other, more traditional algorithms. However, in practice we cannot always rely on the sharp clear boundaries, so the results of the final pass are more important and representative. This might be true in real-world scenarios, but not in our "sterile" case, where we have green screen, masks and slow motion between frames. The authors also explicitly state that PWC-Net delivers better results for parts of images with large motion and away from edges.

Although being a very promising solution, PWC-Net cannot help us in achieving results of quality that we want in our specific task.

Chapter 3

As-rigid-as-possible image registration

As-rigid-as-possible (ARAP) image registration is an algorithm developed by Sýkora et al. and originally proposed in [19]. Unlike all the other methods we have overviewed, it is meant to explicitly preserve local rigidity of the object being registered between frames. At each iteration, it performs regularization with the help of multiple rigid transformations applied to a grid that is constructed over the object.

Firstly, we will briefly go over the concept of rigid transformations, and then we will discuss the algorithm.

3.1 Rigid transformation

A rigid transformation consists of rotation and translation – is does not involve any scaling or shearing. To represent it, we need rotation matrix $\mathbf{R}$ and translation vector $\boldsymbol{t}$. We transform a given point $\boldsymbol{x}$ as follows:

$$T(\boldsymbol{x}) = \mathbf{R}\boldsymbol{x} + \boldsymbol{t}. \tag{3.1}$$

Matrix $\mathbf{R}$ can be viewed as an orthogonal transformation – it preserves lengths of vectors and angles between them [20]. Thus, it can be represented with an orthogonal matrix. Such matrices have this property [21]:

$$\mathbf{A}^{-1} = \mathbf{A}^T \quad \rightarrow \quad \mathbf{A}\mathbf{A}^T = \mathbf{I}.$$

Keeping that in mind, consider the case: suppose, we have two sets of control points $\{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_n\}$ and $\{\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n\}$ and we want to find the best

single rigid transformation to map between them. In other words, we want to minimize

$$E(\mathbf{R}, \boldsymbol{t}) = \sum_{i=1}^{n} \|\mathbf{R}\boldsymbol{p}_i + \boldsymbol{t} - \boldsymbol{q}_i\|_2^2$$

$$\text{s.t. } \mathbf{R}\mathbf{R}^T = \mathbf{I}$$

with respect to both $\mathbf{R}$ and $\boldsymbol{t}$. With the help of this sum of squared differences, we want to find a rigid transformation that will bring $\boldsymbol{p}_i$ points as close to their desired $\boldsymbol{q}_i$ locations as possible.

According to [22], there exists an analytical solution for 2D case. Firstly, we compute centroids $\boldsymbol{p}_c$ and $\boldsymbol{q}_c$ from both sets of control points as

$$\boldsymbol{p}_c = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{p}_i, \qquad \boldsymbol{q}_c = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{q}_i. \tag{3.2}$$

Secondly, we define $\hat{\boldsymbol{p}}_i$ and $\hat{\boldsymbol{q}}_i$ to be

$$\hat{\boldsymbol{p}}_i = \boldsymbol{p}_i - \boldsymbol{p}_c, \qquad \hat{\boldsymbol{q}}_i = \boldsymbol{q}_i - \boldsymbol{q}_c. \tag{3.3}$$

Then, we are ready to compute matrix $\mathbf{R}$ (symbol $\perp$ denotes a perpendicular vector; if $\boldsymbol{x}^T = [x,\, y]$, then $\boldsymbol{x}^{\perp T} = [-y,\, x]$):

$$\begin{aligned} \mathbf{R} &= \frac{1}{\mu}\sum_{i=1}^{n} \begin{bmatrix} \hat{\boldsymbol{p}}_i^T \\ \hat{\boldsymbol{p}}_i^{\perp T} \end{bmatrix} \begin{bmatrix} \hat{\boldsymbol{q}}_i & \hat{\boldsymbol{q}}_i^{\perp} \end{bmatrix}, \\ \mu &= \sqrt{\left[\sum_{i=1}^{n} \hat{\boldsymbol{p}}_i^T \hat{\boldsymbol{q}}_i\right]^2 + \left[\sum_{i=1}^{n} \hat{\boldsymbol{p}}_i^{\perp T} \hat{\boldsymbol{q}}_i\right]^2}. \end{aligned} \tag{3.4}$$

Once the rotation matrix is acquired, we can easily get translation vector $\boldsymbol{t}$:

$$\boldsymbol{t} = \boldsymbol{q}_c - \mathbf{R}\boldsymbol{p}_c. \tag{3.5}$$

Note that this will give us a single rigid transformation, which most likely will not be able to map every point $\boldsymbol{p}_i$ to $\boldsymbol{q}_i$ exactly. But we do not want an exact mapping since it will (in a general case) break the rigidity of the transformation. Some sources also provide a weighted version of the equations above: these weights should be computed for every pixel individually because

they show how far different control points are situated with respect to a given pixel. Using weighted equations will effectively yield a distinct transformation valid only for the pixel the weights have been computed for. As you will see later, we will use rigid transformations for cells of a grid, where every cell is defined by 4 control points, and all pixels inside it should be transformed in the same way. This is why we do not require any weighting scheme for our particular task.

3.2 The algorithm

We take an image and construct a 2D grid over the object we want to register. Later, for its every cell we will be repeatedly computing rigid transformations. In addition to reducing the number of parameters to optimize, the grid also controls the level of rigidity: the bigger its cells are, the more rigid the overall transformation of the object will be.

ARAP image registration consists of two phases listed below.

Pushing phase
We search for a better location for every vertex of the grid based on the similarity of its neighborhood with the second image. In this stage, we take a patch – a window of a predefined size with a vertex being in the center of it – and use block matching to register it by minimizing a sum of absolute differences of pixel intensities (see Figure 3.1). The area which we are looking for the minimum SAD in is not a whole image: it is also a fixed-size area, slightly bigger than a patch (in the original article, it could fit 9 patches).

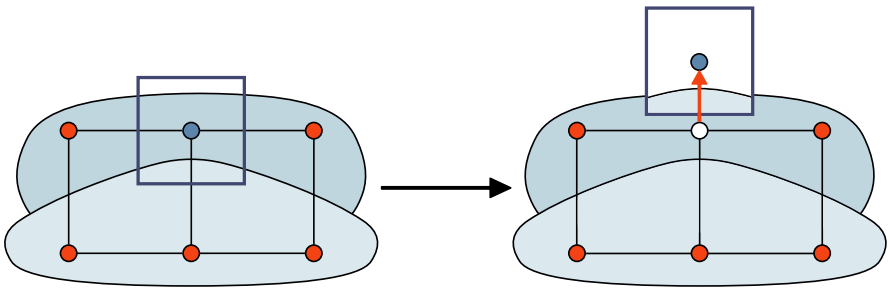


Figure 3.1: Pushing phase. The lighter and the darker shapes represent the same object in two different images. We have constructed a grid over the lighter shape that is to be aligned with its darker counterpart. Initially capturing a top center part of the object in the first image, the patch becomes roughly aligned with top center of its deformed version in the second one.

Block matching is essentially a brute force approach (which can still make use of some optimizations such as early termination or winner-update strategy). Thus, it does not get stuck in local minimums – in contrast to gradient-based

optimization algorithms. The size of the search area is rather restricted. This limitation is mitigated by the fact that the ARAP image registration method is iterative: pushing phase is repeated multiple times and the search area moves together with the corresponding vertex at each iteration. This process allows it to eventually find the most suitable location for the patch far beyond the relatively small region around the initial position of the vertex.

Regularization phase

After we have pushed all vertices to more suitable locations during the previous phase, our grid becomes arbitrarily deformed: we have allowed its nodes to float independently of each other without any respect to the geometry of our object. This is where the process of regularization comes in. For every (quadrilateral) cell of the gird, we take the coordinates of its 4 vertices before ($\boldsymbol{p}_i$) and after ($\boldsymbol{q}_i$) the pushing phase. We find a rigid transformation between them and apply it to the initial positions ($\boldsymbol{p}_i$). Since it cannot always map the points exactly, the vertices of the grid become split into several instances (by the number of cells that have been sharing any given vertex). We fix this by computing the average of all two, three or four of them. After that, not only the grid is restored, but the rigidity of the object is improved. New vertices adjust their after-pushing-phase locations now with respect to multiple rigid transformations (see Figure 3.2). We can repeat the process of regularization more times before starting with another outer algorithm iteration, which will mark the beginning of a new pushing phase.

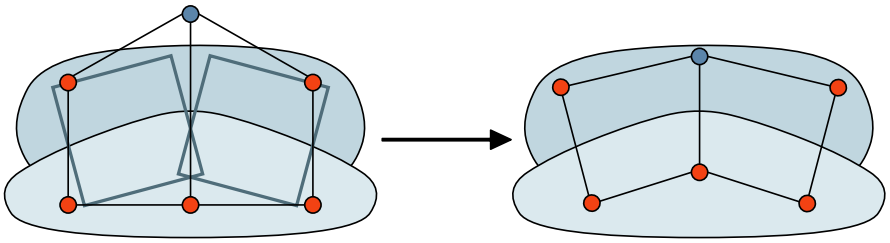


Figure 3.2: Regularization phase. Based on the new positions of control points obtained after the pushing phase, we compute a rigid transformation for every cell of the grid. When some vertices are split into several instances (like the two in the middle), we find their average position.

Since the algorithm is iterative, we should correctly detect convergence to stop iterating. There may be many conditions to choose from; the authors propose to identify the moment when the average distance between initial and moved vertices stops increasing significantly and remains almost the same for the most recent N iterations.

3.3 Drawing resulting images

When registration completes, we obtain a deformed grid. We need to compute an appropriate transformation for every cell – we have averaged many vertices, so we cannot simply apply a rigid transformation because it will not map every pixel inside a deformed cell precisely. There are a few possible ways we can take advantage of.

3.3.1 Using third-party software

We can delegate the rendering task to dedicated third-party software – something like OpenGL. When making a production-ready solution, this is the most appropriate way to do it. Nevertheless, there are two issues with this approach:

1. It will introduce additional dependencies to our code and make it less portable.

2. It will require more time to find a suitable framework, study it thoroughly and integrate with the algorithm.

In this thesis, we are viewing the ARAP image registration technique as a proof of concept and not trying to code the final production version. So, we are going to use a simpler ad hoc solution described below.

3.3.2 Implementing a simple rendering routine

To render our deformed grid, we need to do the following:

1. For every pixel of the resulting image, we need to determine which cell of the grid it belongs to. Every cell has its own transformation, so we should choose the right one to apply.

2. After that, we need to compute a transformation between the original form of the cell and its deformed version. We know the current location of the pixel we want to get the color for inside of the resulting image, so it is much more convenient to construct the transformation backwards: we will obtain real-valued coordinates for the original image and we can interpolate them to get the final intensity.

Let us go through both steps in more details.

3.3.2.1 Choosing an appropriate cell

When we draw the resulting image, we iterate over all cells we have. For each of them, we find minimum and maximum x and y values, thus determining a rectangle that has all sides parallel to either x or y axis – it contains our deformed quadrilateral. Then, in a nested loop we start to go over all pixels inside it. This is where we need to decide which pixels belong to the cell and which do not. Below, we provide two possible ways to determine it.

Edge function
As stated in [23], we can use the edge function. It is given by the formula

$$E(\boldsymbol{v}_0) = (\boldsymbol{v}_{0,x} - \boldsymbol{v}_{1,x})(\boldsymbol{v}_{2,y} - \boldsymbol{v}_{1,y}) - (\boldsymbol{v}_{0,y} - \boldsymbol{v}_{1,y})(\boldsymbol{v}_{2,x} - \boldsymbol{v}_{1,x}), \tag{3.6}$$

where $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are two vertices defining an edge of the polygon and $\boldsymbol{v}_0$ is a point we want to test. The output value of this function can be positive, negative and zero. If it is zero, then the given point lies precisely on the edge. Otherwise, the sign determines whether it is to the left or right of the edge vector. In our case, we traverse edges of the polygon in the clockwise direction, and our coordinate system starts at the top left corner of the image. Under these conditions, a positive value indicates that the point lies to the left of the edge and a negative one shows that it lies to the right (Figure 3.3). Note that this is different in comparison to the example from the article because "left" and "right" side labels depend on the coordinate system.

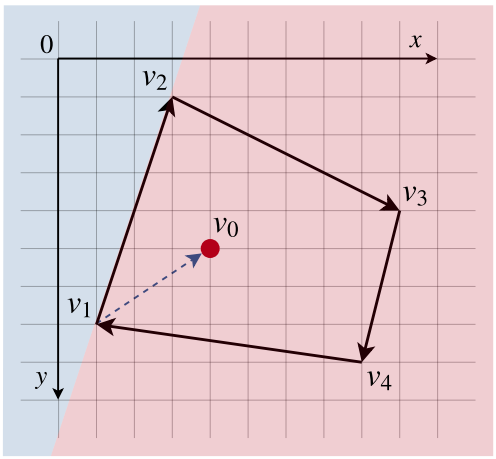


Figure 3.3: The $\boldsymbol{v}_2 - \boldsymbol{v}_1$ edge function splits the plane into the left and right sides. If the point lies to the right of every edge vector, then it is inside the polygon.

The function in Equation (3.6) can be viewed as a cross product between $\boldsymbol{v}_0 - \boldsymbol{v}_1$ and $\boldsymbol{v}_2 - \boldsymbol{v}_1$ vectors. Although cross product by itself is undefined in 2D, we can add the third dimension to every point and set it to zero. We

will obtain a vector with 3 elements where only the third one is non-zero: its absolute value is equal to the area of the parallelogram between the two vectors and its sign shows the direction of rotation we need to perform to align them (right-hand rule).

To see if a point lies inside of a polygon, we need to apply the edge function to every edge together with the given point. If all 4 values appear to be negative, then the point is inside. It is important to traverse the edges in a consistent direction (clockwise in our case).

This method has one fatal flaw that makes it unsuitable for our grid: it can correctly handle only convex polygons. But our quadrilaterals can end up being deformed too much, so testing them with the edge function can yield false negative results. To address this issue, we will use the technique described next.

PNPOLY

PNPOLY (point inclusion in polygon test) is an algorithm developed by Franklin [24]. It is based on the idea of ray casting: starting with the point we want to test, we "cast a ray" from its position along a certain direction and see how many times it crosses the edges. The point lies inside the polygon if the number is odd and outside it if the number is even. This is illustrated in Figure 3.4.

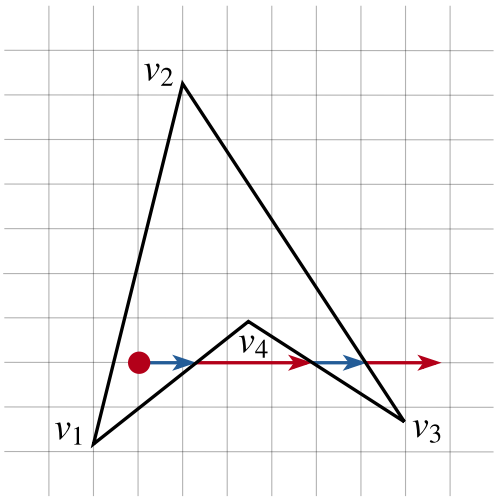


Figure 3.4: The ray crosses three edges. Since the number is odd, we conclude the point lies inside the polygon. The edge function will produce a false negative here because of $\boldsymbol{v}_4 - \boldsymbol{v}_3$ edge.

There exist many implementations of ray casting algorithms. We have chosen PNPOLY because of its following properties:

- It is concise: the original implementation in C is only 7 lines of code.

- It correctly handles concave polygons (unlike the edge function described above).
- It automatically deals with point-on-the-boundary cases. In particular, if the plane is divided into polygons, every point will be assigned to exactly one of them. In other words, we will not face situations such as coloring a pixel twice (when it is classified as a part of more than one grid cell) or filling in gaps (when a pixel is not classified at all).

This algorithm is not perfect though. In fact, there is even a study of how unreliable such methods can be [25]. But its performance is more than sufficient for our task, and it is much better than the edge function. This is why we choose PNPOLY to do grid rasterization for us.

3.3.2.2 Constructing transformations

Once we determine which cell of the grid a given pixel belongs to, we should construct an appropriate transformation. We can do it, for example, with one of the following approaches.

Projective transformation [1]
Cells in our grid are all quadrilaterals. Having two shapes with 4 vertices each, we can construct a projective transformation to map between them. It is based on 4 2D points; thus, is has 8 degrees of freedom – 8 variables that constitute it. If $[x, y]$ is a source point, we can obtain its target counterpart $[x', y']$ as

$$x' = \frac{a_{11}x + a_{12}y + t_x}{f_1 x + f_2 y + 1}, \qquad y' = \frac{a_{21}x + a_{22}y + t_y}{f_1 x + f_2 y + 1}.$$

If we multiply both sides by the denominator and rearrange the equalities to have x' and y' in the right-hand side and everything else in the left-hand side, we will get this equation:

$$\boldsymbol{v}^T = [a_{11}, a_{12}, a_{21}, a_{22}, f_1, f_2, t_x, t_y],$$

$$\begin{bmatrix} x_1 & y_1 & 0 & 0 & -x_1x_1' & -y_1x_1' & 1 & 0 \\ 0 & 0 & x_1 & y_1 & -x_1y_1' & -y_1y_1' & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_4 & y_4 & 0 & 0 & -x_4x_4' & -y_4x_4' & 1 & 0 \\ 0 & 0 & x_4 & y_4 & -x_4y_4' & -y_4y_4' & 0 & 1 \end{bmatrix} \boldsymbol{v} = \begin{bmatrix} x_1' \\ y_1' \\ \vdots \\ x_4' \\ y_4' \end{bmatrix}.$$

Each of the 4 corner points generates two rows for the matrix and two elements for the right-hand-side vector. By solving this linear system, we will get 8 parameters to build a projective transformation from. In our case, the

source points are the vertices of the grid we have on the latter frame and the target ones are the vertices of the grid from the earlier frame. This way, we map backwards in the frame order.

Mean value coordinates

As an alternative to computing projective transformations, we can make use of mean value coordinates [26]. The idea is to find a weight λ_i for each of the 4 vertices of the quadrilateral containing the point so that

$$\sum_{i=1}^{4} \lambda_i \boldsymbol{v}_i = \boldsymbol{v}_0, \qquad \sum_{i=1}^{4} \lambda_i = 1. \tag{3.7}$$

Essentially, we are expressing coordinates of a pixel through a weighted sum (a convex combination) of the vertices that define its enclosing cell. We can then apply those weights to any other set of 4 vertices to get a new location that will relate to them in a similar way the initial location relates to initial vertices. This method scales well to polygons with arbitrary number of sides which do not even have to be convex.

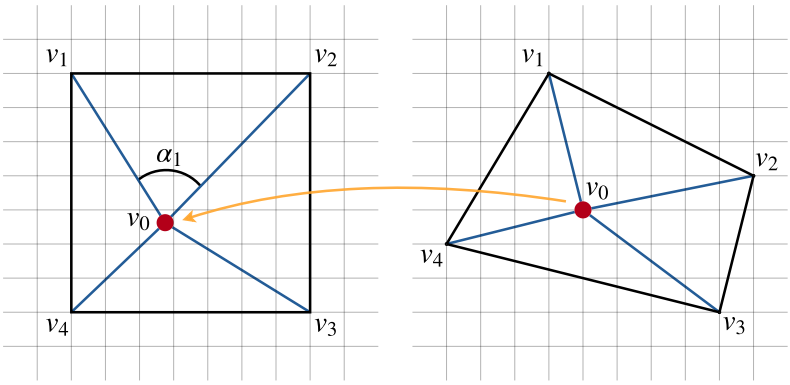


Figure 3.5: Finding source pixel position via mean value coordinates

Suppose that we have a quadrilateral with $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_4$ vertices and a point $\boldsymbol{v}_0$ inside it (Figure 3.5). We can find the weights $\lambda_1, \ldots, \lambda_4$ as

$$\lambda_i = \frac{w_i}{\sum_{j=1}^{4} w_j}, \qquad w_i = \frac{\tan\left(\alpha_{i-1} / 2\right) + \tan\left(\alpha_i / 2\right)}{\|\boldsymbol{v}_i - \boldsymbol{v}_0\|}.$$

We compute them for every pixel inside a deformed cell to apply to the vertices of the original version of the cell. Thus, we get the desired source location in the keyframe to pick color from.

We cannot easily determine which of the two approaches is better. We have chosen mean value coordinates since this method is slightly easier to implement.

3.4 Estimating optical flow

At first, we have tried to estimate optical flow via ARAP image registration in the pairwise manner, which is described in Section 2.1.6.1 and has been used for all previously discussed algorithms. This way, the results appeared to be unsatisfactory: the synthesized image was unacceptably deformed. This can be a consequence of the low precision of the algorithm: because of block matching, we have only integer offsets for every cell. In the article, the authors state one can compensate the absence of subpixel-level precision by refining the final result with some other algorithm such as the MRF-based image registration by Glocker et al. [11]

We have then redesigned the flow estimation part: now, we use the single instance of the grid throughout the entire video sequence. That is, when the process of registration completes for another pair of frames, we do not reset the deformed grid. Instead, we use it as a start for the next pair. This gives us much better results as shown in Figure 3.6.

(a) (b)

Figure 3.6: Estimating optical flow with ARAP image registration. (a) The last frame from the original video. (b) The same frame synthesized with the texture from the first frame warped by optical flow. Original frame (a): © MAUR film.

As you can see, the texture looks much better. It is not as smooth as it was when using the Lucas–Kanade and the Glocker et al. techniques. It now has more sharp elements transferred directly from the first frame. Having the

same grid on every image in a sequence allows us to construct transformations from a given frame straight to any other one, so we use this property to get the colors from the very first frame.

Unlike the results of SIFT flow and PWC-Net, the silhouette appears to be preserved much better. Still, the lack of precision is noticeable.

3.5 Summary

In the next chapter, we will present the final algorithm we have. We will see if it can solve the problems of all the other methods we have reviewed so far and produce even better results.

Chapter 4

LK-ARAP optical flow

The LK-ARAP image registration method has been developed by Ondřej Jamriška and is yet to be published (personal communication, October 5, 2018). As the tests will show in Chapter 6, in our particular task this algorithm produces the most visually pleasing results compared to all other previously reviewed techniques.

4.1 The key idea

The key idea behind the LK-ARAP algorithm is to enhance the regular as-rigid-as-possible image registration method described in Chapter 3 with subpixel-level precision. Letters LK in its name stand for "Lucas–Kanade" since

- it uses sum of squared differences in the data term instead of the absolute ones; and
- its error function is minimized with the help of a gradient-based approach (in contrast to gradient-free block matching).

Positions of grid vertices can be real-valued – there are no hard constraints on lengths of offset vectors. In the base ARAP algorithm, they can also be real-valued but only thanks to the regularization phase: block matching that is used to minimize visual differences of grid cells can initially produce new positions with only integer coordinates. With an approach similar to the one in the Lucas–Kanade method, we can smoothly move the grid adjusting it as needed based on both the value and the gradient of the dissimilarity function. We also use bilinear interpolation to obtain color values.

The LK-ARAP algorithm has a single compound error function (as opposed to the two independent phases of the base ARAP registration). This allows

us to control the influence of the data and rigidity dissimilarities individually by assigning different weights to them.

These adjustments might seem insignificant but in fact they play a crucial role in achieving the best results.

4.2 The algorithm

The error function of the algorithm is as follows:

$$E(w) = \underbrace{\sum_{(\mathcal{C},\mathcal{C}_w)\in(\mathcal{G},w(\mathcal{G}))} \sum_{(\boldsymbol{s},\boldsymbol{s}_w)\in(\mathcal{C},\mathcal{C}_w)} \|F(\boldsymbol{s}) - G(\boldsymbol{s}_w)\|_2^2}_{\text{data term}} + \underbrace{\sum_{(\mathcal{C},\mathcal{C}_w)\in(\mathcal{G},w(\mathcal{G}))} \sum_{(\boldsymbol{v},\boldsymbol{v}_w)\in(\mathcal{C},\mathcal{C}_w)} \alpha \left\|r\left(\mathcal{C},\mathcal{C}_w,\boldsymbol{v}\right) - \boldsymbol{v}_w\right\|_2^2}_{\text{rigidity term}}. \tag{4.1}$$

Let us describe it in every detail.

We have two consecutive frames from the video: the earlier one is F and the latter one is G. We also have a grid* $\mathcal{G}$ constructed over the object in F and fixed in it. Our aim is to find the best transformation w for this grid in G to minimize the error function. Its symbol can be interpreted as "warping": this function simply takes a grid as an input and changes positions of its vertices.

The transformation w is applied to the grid as a whole – we do not allow its cells to be warped independently. That is, unlike classic ARAP registration, there is no averaging of vertices. The grid never gets decomposed into freely floating quadrilaterals. Every non-corner vertex is *shared* between 2, 3, or 4 cells.

A cell is denoted with $\mathcal{C}$ or $\mathcal{C}_w$ in case it belongs to the transformed version of the grid. In our formula, every cell has two sets of elements we can iterate over: the first one contains 4 vertices $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_4$ and the other one – an arbitrary number of samples, or to be more precise *color sampling points* $\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_k$. Both of them are represented as vectors of two elements. The vertices are stored "as is"; the sampling points, however, are expressed through mean value coordinates (see Figure 4.1 and Equation (3.7)). This is necessary for preserving their relative positions inside the cell when the grid inevitably gets deformed. Sampling points are the locations at which we fetch pixel intensities and compute their differences (between every $\boldsymbol{s}_i$ from the current version of cell $\mathcal{C}$ and its $\boldsymbol{s}_{w,i}$ counterpart from the newly deformed $\mathcal{C}_w$). We store them as

*Note that we use calligraphic letters for the grid and its cells. This should help to distinguish between grid $\mathcal{G}$ and "target" image G.

sets of 4 weights each and recompute their exact positions every time we want to fetch a color value. The weights remain the same in every frame regardless any grid transformations.

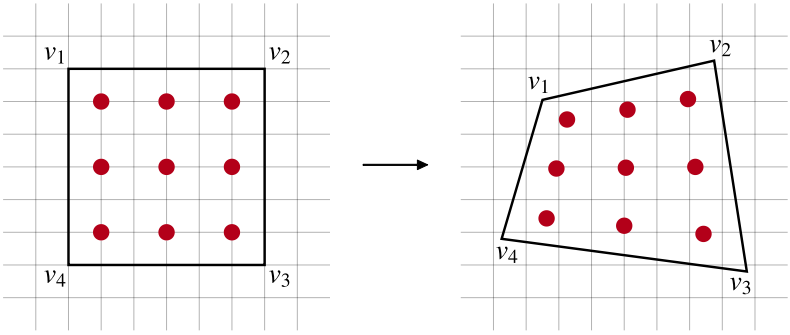


Figure 4.1: A simplified example of a cell with 9 sampling points inside it. Thanks to mean value coordinates, their relative positions remain similar however deformed the cell is.

The **data term** should now be straightforward: we simply accumulate the squared differences of pixel intensities between the two images using two sets of the same cardinality filled with real-valued coordinates. We do it across all cells in the grid.

In the **rigidity term**, for every cell we take its current vertices in F and their altered versions in G and compute a rigid transformation between them. Then, we apply it to the vertices of the cell in F to see how close they will move to their desired positions inside G. If they happen to map exactly, then the transformation is rigid and we have zero error here. If not, we should penalize the function since we want the transformations to be as rigid as possible. We take Euclidean distances between every vertex of the cell in F warped by the obtained rigid transformation and its desired location in G. The values of these distances are squared and added to the overall error.

The function r computes a rigid transformation between the vertices of two input cells (its first two arguments) and applies it to the point given as the last argument. It is a shortcut for Equations (3.1) to (3.5).

The weight α is introduced to adjust the influence of the rigidity component. The bigger it is, the more rigid grid transformation we will get at the cost of more precise color-based non-rigid mapping.

The other parameters we can freely choose values for are the length of the side of a cell and the number of color sampling points inside it. Larger cells can prevent us from finding a good color-based object mapping by taking away elasticity of the grid and its ability to adapt to tiny local deformations, while smaller ones can fail to cope with preserving local rigidity. The number of

sampling points, in most cases, should be equal to the area of a cell, so we neither waste any pixels nor fetch too many mixtures of the same colors in different proportions.

The entire error accumulation algorithm can be expressed with the pseudocode provided in Listing 1.

```
1  energy = 0
2  for cellF, cellG in zip(grid, warp(grid)):
3      # Color differences
4      for point in cellF:
5          sourcePos = meanValue(point.weights, cellF.vertices)
6          targetPos = meanValue(point.weights, cellG.vertices)
7          colorDiff = F[sourcePos] - G[targetPos]
8          colorDiffSquared = colorDiff.norm().pow(2)
9          energy += colorDiffSquared
10     # Rigidity differences
11     R, t = rigidTransformation(cellF.vertices, cellG.vertices)
12     warpedSourceVertices = [R * v + t for v in cellF.vertices]
13     rigidityDiff = warpedSourceVertices - cellG.vertices
14     rigidityDiffSquared = rigidityDiff.norm().pow(2)
15     energy += alpha * rigidityDiffSquared
```

Listing 1: Error calculation pseudocode. In line 4, it makes no difference whether to iterate over `cellF` or `cellG` because the weights of sampling points are the same regardless of grid transformations.

4.3 Specifics of the realization

The reference implementation of the algorithm has some interesting specifics:

Caching colors from the keyframe
When we calculate the difference in colors between images F and G, the grid in F is effectively "frozen". When we want to try out and evaluate various positions of vertices inside G, we do not have to repeatedly fetch the intensities from F. Instead, we can read them once and cache them. At first, this might seem to work only during the error minimization between a given pair of images. However, if the lighting of the scene does not change much, we can cache color samples only once – in the keyframe – and reuse them throughout the whole video sequence. This will also help us to preserve more color information to match against – provided that the object in the keyframe can be seen more clearly and has less deformations than in the following images.

Keeping two separate grids for the data and rigidity terms
When the object moves and changes its shape from frame to frame, the grid can get badly deformed. This can lead to loss in efficiency of the rigidity term. To overcome the issue, we can keep two separate versions of the grid for image F. The first one is the fully deformed version – the final state that has been registered inside F from the frame previous to it. When registering between F and G, we will use it to compute the data term. The second one is the grid from the frame previous to F that is slightly interpolated towards the final fully registered version (say, by 10%). This grid can be used to calculate the rigidity term between F and G. As practice shows, this adjustment helps to achieve better results – most probably because of preserving some information about the original shape of the object. It can always be turned off by setting the interpolation coefficient to 100%.

Extending the mask
If the background of the object we want to track is homogeneous (a green screen, for instance), we can try to extend the mask – to make it a few pixels thicker. It can help the grid to capture the boundaries of the object and better hold on them.

4.4 Minimization

As we have seen in the other previously reviewed algorithms, there can exist numerous ways of how to minimize a given function. But according to the assignment of this thesis, we should concentrate on using only one particular method – the L-BFGS one, which we will describe in this section. We will also cover the concept of dual numbers because they play a crucial role in providing the minimization algorithm with necessary inputs – values of the function and its gradient.

The main source of information for the L-BFGS part has been chosen to be [27]: this book provides many detailed insights into how various optimization algorithms work. In the assignment of the thesis you may find [28], which is an article with a brief summary of the main steps of the L-BFGS method; it is devoted to a comparison with some other minimization approaches.

4.4.1 The L-BFGS method

In its name, letter L stands for limited-memory, and BFGS is an acronym for Broyden, Fletcher, Goldfarb, and Shanno – the creators of the algorithm. We will firstly provide a high-level overview of the basic BFGS method and then see what is different in its limited-memory variation.

4.4.1.1 The basic BFGS algorithm

The BFGS method is an iterative quasi-Newton technique. It starts at some point x_0 and follows the "downhill direction" of the function until it reaches the minimum. At the beginning of each iteration, it requires us to evaluate the function and its gradient. Using this information, it updates its knowledge about the function and decides where to go next, so that the value of the function should further decrease. Let us describe the individual steps of one of its iterations.

Step 1: Determining the direction
Standing at $\boldsymbol{x}_k$, the first thing the BFGS algorithm determines is the direction of the next step, which we express with a vector $\boldsymbol{p}_k$. Some other methods (gradient descent, for example) rely on the direction opposite to the gradient of the function $-\nabla f(\boldsymbol{x}_k)$ (it is also called the steepest descent direction). Our algorithm, however, puts it to question and additionally takes into account the curvature information.

Provided that the given (often called objective) function is twice continuously differentiable, we can expand it with the help of Taylor's theorem:

$$f(\boldsymbol{x}_k + \boldsymbol{p}_k) = f(\boldsymbol{x}_k) + \boldsymbol{p}_k^T \nabla f(\boldsymbol{x}_k) + \frac{1}{2}\boldsymbol{p}_k^T \nabla^2 f(\boldsymbol{x}_k + t\boldsymbol{p}_k)\boldsymbol{p}_k, \quad t \in (0,\, 1).$$

It is important to notice that to reduce the value of the function one does not have to follow the steepest descent direction exactly. In fact, we can pick any $\boldsymbol{p}_k$ which makes an angle smaller than $\pi/2$ with it. Under this condition, the angle between $\boldsymbol{p}_k$ and the original gradient direction will be strictly larger than $\pi/2$. This way, the second term in the expanded form in the above equation will still remain negative (this can be checked by substituting the formula of the dot product in place of the regular multiplication), and the value at $\boldsymbol{x}_k + \boldsymbol{p}_k$ will be less than at the current $\boldsymbol{x}_k$. Detailed proofs considering both steepest descent and downhill directions can be found in [27].

Let us approximate the value of $f(\boldsymbol{x}_k + \boldsymbol{p}_k)$ by setting t to be always zero:

$$f(\boldsymbol{x}_k + \boldsymbol{p}_k) \approx f(\boldsymbol{x}_k) + \boldsymbol{p}_k^T \nabla f(\boldsymbol{x}_k) + \frac{1}{2}\boldsymbol{p}_k^T \nabla^2 f(\boldsymbol{x}_k)\boldsymbol{p}_k. \tag{4.2}$$

If we differentiate it with respect to $\boldsymbol{p}_k$, we will see that

$$\boldsymbol{p}_k = -(\nabla^2 f(\boldsymbol{x}_k))^{-1} \nabla f(\boldsymbol{x}_k). \tag{4.3}$$

This $\boldsymbol{p}_k$ is called the Newton direction. We can assume that by following it we will obtain smaller function values, when the approximation in Equation (4.2) does not differ much from the original function and the size of the step is small enough.

This approach, however, has one important property that can potentially cause problems: the Hessian $\nabla^2 f(\boldsymbol{x}_k)$ must necessarily be positive definite, and we cannot guarantee it with our objective function. Luckily, the BFGS method actually uses quasi-Newton direction: it does not require the knowledge of the real second derivative. Instead, the Hessian gets approximated and continuously refined with the help of the values of the function and its gradient. If we start with a positive definite matrix ($\mathbf{I}$, for example), then the BFGS update formula (in step 3) together with the Wolfe conditions (in step 2) will make sure that its updated version will also be positive definite even if the objective function is not convex.

To sum up, in this step of the current (k-th) iteration the BFGS algorithm multiplies the latest approximation of the inverted Hessian ($\mathbf{H}_k$) by the steepest descent direction and obtains a search direction $\boldsymbol{p}_k$ (similarly to Equation (4.3)):

$$\boldsymbol{p}_k = -\mathbf{H}_k \nabla f(\boldsymbol{x}_k).$$

We write $\mathbf{H}_k$ instead of $(\nabla^2 f(\boldsymbol{x}_k))^{-1}$ because we do not evaluate the Hessian at $\boldsymbol{x}_k$ but rather use its fixed inverse approximated specifically at this ($\boldsymbol{x}_k$) point. The BFGS method stores only the inverse of the approximated Hessian and applies all updates straight to it. This way, we do not need to compute the inverse every time we need it.

If the norm of the gradient is bigger than a certain threshold, we can terminate the algorithm rather than computing $\boldsymbol{p}_k$. The output will be $\boldsymbol{x}_k$ – the current position we are standing at.

Step 2: Calculating the length of the step
Having determined the direction $\boldsymbol{p}_k$, we have to also decide how far along it we will go. The goal of this step is to get

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$$

by appropriately choosing the best step length α_k (a positive scalar). We also want to do it fast, without too many evaluations of the objective function, so we cannot minimize $f(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k)$ w.r.t. α_k exactly. This is why we fall back to the inexact linear search.

We impose the Wolfe conditions on the desired α_k. The first one is called the sufficient decrease condition. It has the form of

$$f(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k) \leq f(\boldsymbol{x}_k) + c_1 \alpha_k \nabla f(\boldsymbol{x}_k)^T \boldsymbol{p}_k.$$

It implies that the step length should be proportional to the decrease in the function value we will obtain – the farther we go, the smaller value we should get. And, naturally, the value cannot increase – moving to a point with a

larger value would make no sense in this deterministic algorithm. Constant c_1 should be from (0, 1) interval.

The second inequality is called the curvature condition. It is formulated as

$$\nabla f(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k)^T \boldsymbol{p}_k \geq c_2 \nabla f(\boldsymbol{x}_k)^T \boldsymbol{p}_k.$$

According to it, the derivative of the function at $\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$ should be sloping either less steeply or in the opposite direction (compared to the initial position $\boldsymbol{x}_k$). It indicates that we will not likely get further decrease of the function value along the given direction. Otherwise, if this condition does not hold, we can continue moving along $\boldsymbol{p}_k$ to get an even better result. Constant c_2 should belong to $(c_1, 1)$.

All in all, in this step the BFGS method moves along the previously obtained direction $\boldsymbol{p}_k$ and calculates the next position $\boldsymbol{x}_{k+1}$, at which the value of the function is smaller than at $\boldsymbol{x}_k$.

Step 3: Updating the approximation of the Hessian
The last step is to update the current approximation of the inverse of the Hessian with the new knowledge we have obtained from the values of the function and its gradient. We impose certain conditions on the new $\mathbf{H}_{k+1}$:

- It should be symmetric – similarly to the inverse of the true Hessian (and the true Hessian itself).
- It should be positive definite for the downhill direction to be calculated correctly. When the inverse is positive definite, so is the original matrix.
- It should satisfy the secant equation.

To define the last condition more precisely: the secant equation has the form of

$$\mathbf{B}_{k+1} \boldsymbol{s}_k = \boldsymbol{y}_k,$$

where $\mathbf{B}_{k+1}$ is the "normal" Hessian, $\boldsymbol{s}_k$ is the difference in function arguments $\boldsymbol{x}_{k+1} - \boldsymbol{x}_k$, and $\boldsymbol{y}_k$ is the difference in the gradient values $\nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k)$. When considering the inverse of the Hessian $\mathbf{H}_{k+1}$, we can write the secant equation as

$$\mathbf{H}_{k+1} \boldsymbol{y}_k = \boldsymbol{s}_k.$$

This condition originates from the requirement that the gradient (w.r.t. $\boldsymbol{p}_k$) of the approximation given in Equation (4.2) (with $\nabla^2 f(\boldsymbol{x}_k)$ substituted with the estimate of the Hessian) should be equal to the one of the objective function for the two most recent iterations. It is satisfiable when $\boldsymbol{y}_k^T \boldsymbol{s}_k > 0$: we have this covered thanks to the curvature condition from the second step.

These three prerequisites defined above will give us infinitely many solutions. This is why we add the fourth one: the new approximation of the inverse Hessian $\mathbf{H}_{k+1}$ should be close to the current $\mathbf{H}_k$: $\|\mathbf{H}_{k+1} - \mathbf{H}_k\|$ should be minimal. Omitting the math, the solution to this minimization problem is given by

$$\mathbf{H}_{k+1} = \mathbf{V}_k^T \mathbf{H}_k \mathbf{V}_k + \rho_k \boldsymbol{s}_k \boldsymbol{s}_k^T, \quad \rho_k = \frac{1}{\boldsymbol{y}_k^T \boldsymbol{s}_k}, \quad \mathbf{V}_k = \mathbf{I} - \rho_k \boldsymbol{y}_k \boldsymbol{s}_k^T.$$

As the initial $\mathbf{H}_0$, we can take, for example, the identity matrix or a multiple of it.

To sum up, at this step the BFGS method updates its estimation of the inverted Hessian for the next $(k + 1)$ iteration based on the newly learned values of the objective function and its gradient. We are ready to start again with step 1, now at $\boldsymbol{x}_{k+1}$ position obtained during the second step.

4.4.1.2 The limited-memory version

The classic version of the BFGS method requires us to store the approximation of the inverted Hessian as a full dense $n \times n$ matrix. When our function has many variables (n is large), $\mathbf{H}_k$ can take too much space. To overcome this issue, the L-BFGS method does not store it explicitly. Instead, it keeps m most recent pairs of $\boldsymbol{s}_i$ and $\boldsymbol{y}_i$. The inverse of the Hessian is reconstructed using them together with some $\mathbf{H}_0$, which can vary from iteration to iteration. The core idea is that the latest information about the function (in the form of $\boldsymbol{s}_i$ and $\boldsymbol{y}_i$) has the greatest impact on the approximation of $\mathbf{H}_{k+1}$, while the older bits of knowledge may be safely discarded. Besides, there exists a special efficient routine for computing $\mathbf{H}_k \nabla f(\boldsymbol{x}_k)$ value from those $\{\boldsymbol{s}_i, \boldsymbol{y}_i\}$ pairs (it can be found again in [27]). Restricting m to be much smaller than n can greatly reduce the memory usage.

4.4.1.3 Limitations

Employing the L-BFGS method involves facing certain limitations:

- Our error function is not guaranteed to be convex. It means that we will likely end up with a local minimizer rather than a global one.
- Our error function relies on the values from images, which are discrete functions. We have to approximate the gradient using one of the many available approaches; this process can be error-prone.

The first issue should not be very important: we do not expect large motion between every two consecutive frames. In general, to completely solve it,

we should abandon gradient-based approaches in favor of something totally different. The truth is that the other methods also have their disadvantages.

As for the second one – in practice, many image-processing algorithms rely on some gradient approximations, yet they work perfectly fine. We can also try various techniques (other than getting the difference between the adjacent pixels).

Anyway, if we succeed in making the whole algorithm fast enough, the user will be able to occasionally adjust the grid position by hand (in an interactive manner).

4.4.2 Dual numbers

For the BFGS method to work, we have to supply it with values of both the error function we want to minimize and its gradient. Evaluating the function is straightforward: we can use the pseudocode from Listing 1. With the gradient, it is not that simple. Of course, we can differentiate the function by hand, but this process is tedious and error-prone. Besides, we will have to manually write and maintain the code calculating partial derivatives. Luckily, there is a totally different and much easier way.

Piponi in [29] suggests using the concept of dual numbers. A dual number is defined as $a + bd$, where a is the real part, b is the imaginary part, and d is a special element with the property that $d^2 = 0$. This notation can appear confusing at first, but we can draw parallels with much more popular complex numbers. They are written as $a + bi$, where i is also a special element turning into -1 when squared. In contrast, the second power of d in dual numbers return 0.

Dual numbers are commutative and associative. We can define basic operations on them:

- Addition:

$$(a_1 + b_1 d) + (a_2 + b_2 d) = (a_1 + a_2) + (b_1 + b_2)d.$$

- Subtraction:

$$(a_1 + b_1 d) - (a_2 + b_2 d) = (a_1 - a_2) + (b_1 - b_2)d.$$

- Multiplication:

$$(a_1 + b_1 d)(a_2 + b_2 d) = a_1 a_2 + (a_1 b_2 + a_2 b_1)d.$$

- Division (by multiplying both the numerator and the denominator by $a_2 - b_2 d$ and simplifying):

$$\frac{a_1 + b_1 d}{a_2 + b_2 d} = \frac{a_1}{a_2} + \frac{a_2 b_1 - a_1 b_2}{a_2^2} d.$$

In order to find the value of the derivative, all we have to do is to evaluate a function using dual numbers instead of the "ordinary" ones. Suppose that we have $f(x) = 4x^3$ and we want to get the value of its derivative at $x = 2$. Let us substitute the input value 2 with a dual $2 + d$ (we set b to 1 whenever we want to compute $\frac{df}{dx}$). The function gets expanded into*:

$$4((2 + d)(2 + d)(2 + d)) = 4((4 + 4d)(2 + d)) = 4(8 + 12d) = 32 + 48d. \quad (4.4)$$

We can verify that the value of derivative is indeed 48 – it is equal to the coefficient in front of d in the result. We have also received the value of the initial function at $x = 2$: it is 32 – the real part of the answer.

To understand why this actually works, let us have a look at the Taylor series expansion of some function $f(a)$ when we replace its argument with a dual number:

$$f(a + bd) = f(a) + bf'(a)d + \frac{1}{2} b^2 f''(a)d^2 + \cdots + \frac{1}{n!} b^n f^{(n)}(a)d^n + \cdots .$$

Thanks to the $d^2 = 0$ property, all the terms in the right-hand side except the two first ones will turn into zeros, so we can write

$$f(a + bd) = f(a) + bf'(a)d. \quad (4.5)$$

Having this b in the right-hand side should now clarify why we have used 1 as the seed value for the coefficient in front of d in Equation (4.4).

We can define the majority of operations on dual numbers with the help of object-oriented programming and operator overloading: we will create a corresponding class with a and b variables and specify custom logic for addition, multiplication, etc. If we have to deal with an arbitrary function, we can construct its dual result by hand (with Equation (4.5) in mind):

$$\sin(a + bd) = \sin(a) + b \cos(a)d.$$

*For the sake of completeness, the constant 4 in front of x^3 could have been written as $4+0d$.

In our case, we need to pick the color value from image I. As inputs, we have 2 variables $x = a_x + b_x d_x$ and $y = a_y + b_y d_y$ specifying the floating-point coordinates. Firstly, we will take their real parts a_x and a_y, read the (grayscale) color intensity $I(x, y)$ and the gradient $\nabla I(x, y)$ as usual (through bilinear interpolation). Then, we will construct the answer as

$$I(x, y) + \nabla_x I(x, y) b_x d_x + \nabla_y I(x, y) b_y d_y. \tag{4.6}$$

We can extend the above formula for RGB images by applying it on every channel separately. We will obtain a vector of three dual numbers (instead of three real ones).

You may have noticed that we have used d_x and d_y elements instead of the single d. That is because we have a function of two variables: x and y. The concept of dual numbers works well with partial derivatives and scales naturally to handle functions of more than one variable. All we have to do is to create an individual d_i element for every variable the function depends on. In case of $f(x, y)$, we have dual numbers in the form of $a + b_x d_x + b_y d_y$, or, more generally, $a + \sum_{i=1}^{n} b_i d_i$, where $n = 2$. In seed imaginary values, we will put 1 near the d_i corresponding to the particular variable i and 0 near all others. In other words, we will compute

$$f(x + 1d_1 + 0d_2,\, y + 0d_1 + 1d_2) = f(x + d_1,\, y + d_2).$$

When we obtain the result, the partial derivative w.r.t. x will be given by a coefficient near d_1 and the one w.r.t. y – near d_2. The property $d_i d_j = 0$ holds for any i and j in any order.

4.4.2.1 Relation to automatic differentiation

The algebra of dual numbers is only a tool, which helps us to implement the forward mode of automatic differentiation. Automatic differentiation is a collection of methods created to evaluate derivatives of functions.

Every function consists of individual subexpressions, which are evaluated in a specific order. Let us show it using an example. Suppose, we have a function

$$f(x, y) = 4x^3 + xy. \tag{4.7}$$

To evaluate it for given x and y, we have to perform a specific sequence of steps. In the forward mode, on every step, we not only compute the value of the current subexpression but also its derivative. The process is illustrated in Figure 4.2.

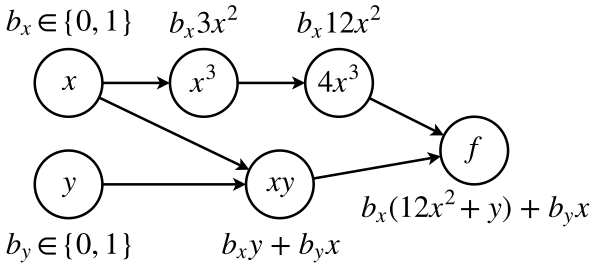


Figure 4.2: Using the forward mode on Equation (4.7). Initial subexpressions are in circles, and their derivatives are near them. Substitute (b_x, b_y) with $(1, 0)$ in the final formula to get $\frac{\partial f}{\partial x}$ and $(0, 1)$ for $\frac{\partial f}{\partial y}$. In practice, only numerical values are stored in each node.

Moving from the most basic subexpressions to the whole function, we step-by-step construct both the value of $f(x, y)$ and its partial derivatives. If we recall the chain rule:

$$\frac{d(g(f(x)))}{dx} = \frac{dg}{df}\frac{df}{dx},$$

we may notice that the forward mode traverses it from right to left – the natural order in which we compute the "usual" value of the function. By providing more combinations of seed values (more d_i elements with corresponding coefficients for input dual numbers) we can calculate more different partial derivatives in the same evaluation run (in case we have a function of more than one variable). This will naturally add extra computational complexity and increase the amount of memory required to store intermediate results.

One can learn more about the forward and the reverse modes of automatic differentiation in [27]. The forward mode through dual numbers is not the only way to compute derivatives, but, according to the assignment of this thesis, we should use exactly this concept.

4.5 Example

Now when we know how the algorithm works, let us finally take a look at its performance on our test video sequence. It is not perfect, but, subjectively, its result is the best one among all the others we have reviewed. It has preserved the texture with most of its fine details and introduced fewer deformations to it. The overall picture may still look slightly blurred due to the fact that we have mapped it from a frame where the object is located farther away from us. This is why even with a perfect mapping it will always appear stretched (in our particular test).

(a) (b)

Figure 4.3: Estimating optical flow with the LK-ARAP image registration. (a) The last frame from the original video. (b) The same frame synthesized with the texture from the first frame warped by optical flow. Original frame (a): © MAUR film.

4.6 Summary

We have described the LK-ARAP image registration algorithm in much detail and showed what we can achieve by using it. In Chapter 5, we will talk about the GPU version of the method and go through the complete list of all changes and optimizations we have made in it. In Chapter 6, we will once again show the results of various algorithms we have tested – now side-by-side – and measure their quality numerically.

Chapter 5

LK-ARAP method GPU optimization

In this chapter, we will discuss the implementation of the LK-ARAP method. We will provide a list of optimizations that help the CUDA version run faster. At the end, we will test and compare the speed of CPU and GPU implementations.

5.1 General details

We have implemented the algorithm using C++. The GPU version is written with the help of the CUDA platform. For operations with matrices and vectors, we have chosen Eigen: it is a header-only library for linear algebra which is compatible with CUDA `nvcc` compiler.

For minimizing the error function with the L-BFGS method, we use libLBFGS available at [30]. We perform minimization directly on the vertices of the grid. As initial values, we take the positions of the vertices we currently have in the previous frame F. Every vertex is represented by its x and y coordinates, so in total we have $2n$ variables to optimize, where n is the number of vertices in the grid.

Dual numbers with a minimal required set of operations have been implemented by us. We have done it to better control and optimize them. We have also exploited the fact that each individual error in the sums from Equation (4.1) depends only on a subset of variables the function is minimized with respect to. This is why we have implemented two classes:

- `SparseDual` stores the imaginary part as two arrays: the first one is for indices i of non-zero coefficients b_i, and the second one – for the actual b_i values. Not only this greatly reduces memory consumption – it also

prevents the arithmetic operations to unnecessarily process zero imaginary coefficients. We use instances of `SparseDual` class everywhere – except for the final sum, which depends on all the variables.

- `DenseDual` is used as a container being filled with individual errors from the data and rigidity term. It stores b_i values in a single array allocated dynamically to accomodate all of them.

For drawing resulting images, we have used the approaches from Section 3.3.2. Now, with the information from Chapter 4, the basic sequential implementation should become straightforward.

5.2 GPU optimizations

We have chosen the CUDA platform developed by NVIDIA to implement a GPU-based version of the LK-ARAP method. At first, this may seem as a vendor lock-in because programs written in CUDA cannot be ported on GPUs of other companies. But in fact, this choice allows us to focus on optimizing and fine-tuning the algorithm for a widespread family of both professional- and consumer-grade graphics cards with less boilerplate code. Every GPU platform has its own specifics, which makes writing and maintaining cross-platform programs much harder. Moreover, if we want to achieve a good speedup everywhere, we will probably have to design and code separate versions of the algorithm for hardware of different platforms and vendors. However, we believe that the optimizations we have used in the CUDA version may still be valid for other GPUs, so they may help in porting or reimplementing our code for other graphics chips.

5.2.1 Applied optimizations

To make the most out of the GPU, we have to deal with its many peculiar properties. Let us go through the most important ones in detail.

5.2.1.1 Identifying regions suitable for GPU parallelization

CUDA programs work best when the data can be split into many individual chunks to be processed independently (with minimal synchronization) in a similar way. In our case, there are two functions that meet these requirements:

- The first one computes the color differences (the data term from Equation (4.1)). We will let every GPU thread compute one difference.
- The second one computes a rigid deformation, applies it, and measures distances between the transformed source vertices and their target counterparts (the rigidity term). One thread can handle one quadrilateral.

A function that is called by the host (ordinary CPU) code to be executed on the device (GPU) is denoted as a kernel. According to the list above, we will have two kernels to implement and run.

The only bottleneck here is the addition of each individual error to the final sum. We have to use an atomic operation to compute the result correctly. Later in this section, we will see how we can reduce the degree of collisions to speed this addition up.

5.2.1.2 Transferring only essential data

To process the data on the GPU, we have to firstly allocate a chunk of space for it inside the GPU memory and transfer it there – it is a time-consuming yet necessary procedure. Ideally, we want to move only the data we have to work with, without unrelated extra dependencies to both save time on transfers and ensure effective cached read. The latter should become clearer from one of the following sections devoted to global memory and access patterns. Right now, let us simply remember that in a kernel we should have only the data essential for processing.

5.2.1.3 Choosing suitable data types

GPU is known to process certain data types faster than others. So we have chosen `float` over `double` for all floating-point operations. As our tests show, the precision is still sufficient. The integer values remain unchanged – they are used to represent indices, for example.

5.2.1.4 Removing heap allocations

CUDA heap allocations are very slow – much slower than in ordinary CPU code. Thus, it is much better to get rid of them completely.

For example, in `SparseDual` class the imaginary part is represented with two dynamically allocated arrays: one for indices and another one for actual elements. The only solution here is to know the upper bound beforehand and hardcode it in. Ours is equal to 8: locations of every color sample depend on 4 vertices of a cell, where each vertex is represented as two dual numbers (for x and y coordinates). Each of these 8 dual numbers have exactly one $b_i = 1$ with all the other coefficients being zero. Constructing the target x and y values through a weighted sum (mean value coordinates) yields two dual numbers with 4 non-zero imaginary coefficients. Later, when we compute gradient by hand and add its parts up (see Equation (4.6)), we obtain a single dual number with 8 non-zero b_i values. The remaining operations with it (subtraction of a fixed cached intensity from F, squaring results from individual channels and adding them together) do not change the number of non-zero imaginary coefficients.

Exactly the same upper bound is also valid when computing rigidity errors since we do it with the help of the same 4 vertices represented with 8 dual numbers in total.

Exploiting these observations, we can fix the upper bound and completely remove the reallocation logic from `SparseDual` class.

Another good case to consider is the `VectorXf` class from the Eigen library. It represents an arbitrarily-sized vector of `float` numbers. We may use it to store the color values extracted from the current frame G. However, is it better to replace it with a fixed-size `Vector3f` that can hold exactly 3 elements. It is enough for RGB images, and no heap allocations are used in this case.

We can also use templates to create arrays of various sizes for different stages of the algorithm as needed.

5.2.1.5 Removing pointers from structures

Suppose that we have an array of size n, where each element is a structure with a pointer as one of its members. To copy it to the GPU, we will have to do $n + 1$ memory allocations and copies: one for the array itself and n for its elements (because we need to copy the additional data that can be reached through a pointer inside every structure separately). This approach has two issues:

- It is slow.
- It requires much more memory (because of all the additional data which GPU needs when it allocates space for the pointer).

To solve this problem, we can use static allocation with a fixed upper bound. If the array of structures is allocated dynamically, every its element will still be managed on the heap of the host (and in global memory on the GPU), so there will be no shortage of space.

For instance, we have already determined the upper bound for `SparseDual` numbers. Now, we can rewrite its arrays for both imaginary indices and coefficients as

```
int*   bIndices  = new int  [8]; -> int   bIndices [8];
float* bElements = new float[8]; -> float bElements[8];
```

5.2.1.6 Using registers – the fastest GPU memory type

To get good performance, it is necessary to carefully choose memory types to use for various pieces of data being addressed and processed. The fastest

memory we have is the memory of registers of the GPU. The amount of it is restricted, so we have to use it carefully and efficiently.

Small arrays allocated on the stack can end up in registers too (provided that we have a sufficient number of free ones). Generally, however, they will be placed inside local memory, which is a part of global memory – the slowest GPU memory type we have.

For example:

- All kernel-local variables of primitive types are stored inside registers by default.
- Local instances of `SparseDual` class are also put into registers together with their two stack-allocated arrays of size 8. We try to use as few sparse dual numbers as we can to prevent "register spilling" (usage of local memory instead when there are not enough free registers).

5.2.1.7 Ensuring coalesced (or cached repetitive) access

When we transfer arrays of data from the host to the device, we effectively copy it into global memory of the GPU. This memory type is the biggest and the slowest among all other types. It is called global because it can be accessed by every thread from every streaming multiprocessor – in contrast to the registers which are thread-local. Since we cannot load the data into registers directly, we have to put it into global memory from the host code and read it on the device.

Fortunately, accesses to global memory are cached: when a thread reads, for instance, a 4-byte floating-point number from an array, several other numbers immediately following it are read and transferred to the cache as well. We can make threads with consecutive IDs read consecutive numbers from an array, so only the first thread makes a request to global memory, and several other threads read the values from the cache instead. This pattern of reading contiguous chunks of memory is called coalesced access [31].

For this reason, we have stated earlier that only the data essential for processing should be found in the arrays transferred to the GPU. The more unrelated things are located in between the computationally-important data pieces, the more chunks of memory we have to access wasting the bandwidth.

Sometimes, even if the data access is strided (not coalesced), it does not necessarily mean that we use the bandwidth inefficiently. Consider the following case: we have a quadrilateral with 36 color samples inside, each one is processed by one thread. These 36 threads simultaneously read the coordinates of the 4 vertices stored in global memory. Although this access is strided (every vertex can be shared between 2 to 4 cells in the grid and stored in

the array only once), it is still fast. This is due to the fact that 36 threads simultaneously read the same pieces of data which get cached after the first access. This wastes the cache itself since other unrelated vertices get also loaded into it, but when its size is sufficient, such a way of reading memory does not cause any problems. This case (in general) can be completely fixed by introducing redundant copies of vertices and storing them near each other, but our experiments have shown that this is not required.

5.2.1.8 Using a texture for random 2D access

When computing differences in colors, we have to access their values in the current frame G. The grid can be deformed in an arbitrary way, so we cannot foresee exact positions obtained through mean value coordinates. This leads to uncoalesced memory access. Due to the fact that every thread reads intensities at unique non-repeating locations, we cannot expect any cache hits.

To fix this problem, we bind the data of image G to a texture. In CUDA, a texture object provides a special caching strategy for the underlying data based on spatial locality [31]: when an element is accessed, not only its neighbors to the left and right get cached – elements from upper and lower locations are also retrieved. This works even if the underlying data has been initially linear, and logically adjacent pixels have been stored with a row-sized stride in between.

5.2.1.9 Reduce the number of texture accesses

We can combine values of 3 channels of an RGB image into one integer. The biggest value will be $255+255*256+255*256^2 = 2^{24}-1$. There is even a reserve for the fourth channel. This procedure will reduce the number of texture accesses 3 times in exchange for 3 extra division and/or modulo operations. Each intensity then has to be normalized to become a floating-point value from $[0, 1]$ interval with an additional division. In many cases with CUDA, exchanging memory accesses for computational operations makes a positive impact on the speed (provided how slow the memory can be).

Also, this will reduce the amount of data being transferred to the GPU. Now, one 4-byte integer can hold all 3 channels, so we do not need to encode each individual channel with its own 4-byte floating-point number.

5.2.1.10 Reducing real part locally at block level

When we have computed an individual squared difference, we need to add it to the total error represented with an instance of `DenseDual`. We do it with the help of atomic operations from the CUDA toolkit. While sets of imaginary indices are unique to every cell of the grid, the real part is always present and is always added to the real part of the total error. To reduce the degree of

collisions for atomic additions, we precompute a local block-of-threads-level sum, and only then the first thread adds it to the result. This is done with the help of the warp shuffle–based reduction method described in [31].

5.2.1.11 Using static indexing

When we have a few small stack-allocated arrays, and they happen to perfectly fit into registers, there is still one hidden problem: if we use dynamic indexing to access their elements, they will be copied to local memory first (which is as slow as global memory). This is because registers are not indexable [31]. Fortunately, we can work around this issue by using either shared memory or static indexing [32]. We have decided for the latter approach because with lots of `SparseDual` instances encapsulating small arrays we find it inconvenient to use shared memory.

The essence of static indexing is the usage of indices known at compile time. Consider the following loop:

```
for (int i = 0; i < 8; ++i) { sum += a[i]; }
```

We know the upper bound, the lower bound and the step size for `i` beforehand – they do not depend on any other variables and will always remain the same. The compiler can optimize this loop by unrolling it: instead of the actual loop there will be only a sequence of operations with fixed indices:

```
sum += a[0];
sum += a[1];
...
```

We can add `#pragma unroll` directive above the loop to explicitly state that we want this behavior.

If we cannot hardcode constants in a simple manner but still precisely know the logic of the algorithm, we can use a template with an integer argument. A function with such a `for` loop can be called in another (outer) `for` loop:

```
for (int i = 0; i < 8; ++i) { doSomething<i>(); }

template <int COUNT>
void doSomething() {
    for (int i = 0; i < COUNT; ++i) { ... }
}
```

In this situation, we can use the concept of template metaprogramming. Just like in [33], we can write our static for loop, which will be compiled into a straightforward sequence of operations (see Listing 2).

```
struct PrinterFunc {
    template <int I>
    void launch(string& text) {
        cout << text << " " << I << endl;
    }
};

template <int I>
struct StaticForLoop {
    template<typename FUNC, typename ...ARGS>
    void launch(ARGS&... args) {
        StaticForLoop<I - 1>().launch<FUNC>(args...);
        FUNC().launch<I>(args...);
    }
};

template <>
struct StaticForLoop<-1> {
    template<typename FUNC, typename ...ARGS>
    void launch(ARGS&... args) {}
};

StaticForLoop<5>().launch<PrinterFunc>("Hello from iteration");
```

Listing 2: A static `for` loop implemented with template metaprogramming

5.2.1.12 Relaxing consistency of sparse dual numbers

In this context, the term "consistency" refers to the increasing order of indices in the imaginary part of sparse dual numbers. This order can be altered when we, for instance, add two dual numbers up. Normally, this operation can be done with a `while` loop which adds two sparse vectors and ensures the result also preserves the increasing order. With this approach, it is impossible to use static indexing – we do not have a `for` loop. But we can still find the solution, in this case – the other way around: we can do without this consistency. We will eventually add our sparse dual number to the total (dense) error. So it will change nothing if, prior to this addition, we have indices in a random order.

Crucially, we have to add up dual numbers which do not share any indices. So, we can rewrite the addition to stack new indices with their coefficients after the existing ones. This is used, for example, when we compute the gradient as

in Equation (4.6) and add two dual numbers – their sets of imaginary indices are disjoint.

In another part of the algorithm, we have to compute the squared error across 3 color channels. The 3 dual numbers we have to add up share all indices in the identical relaxed order (effectively – disorder). So we add the imaginary values up and copy the array of indices unchanged.

5.2.2 Other possible optimizations

In this section, we list a few optimizations that are also, in theory, applicable to our code, but we have decided not to implement them.

5.2.2.1 Reducing imaginary part at cell level

To reduce the degree of collisions even further, we could have reduced the imaginary part inside every cell prior to adding it to the total sum. This approach involves two synchronization points – one when zero-initializing a container for the partial sum and the other one when waiting for all threads processing the same cell to add their contribution to it. When we implemented this optimization and experimented with it, we noticed a performance drop. It seems that simple atomic operations in CUDA are well-optimized, and this kind of partial reduction is not required.

5.2.2.2 Sorting the imaginary part of sparse dual numbers

Instead of relaxing the consistency, we could have used shared memory to correctly sort the imaginary part of dual numbers. While it is totally unnecessary in our case, in another scenario involving sparse dual numbers this optimization may prove to be important and convenient.

5.2.3 Parts left CPU-only

Except for the functions that compute color and rigidity differences, we have left other parts of the algorithm unchanged, mainly:

- Various simple `for` loops: when we have to do something simple with every vertex (for instance, to interpolate the "rest pose" of the grid towards the last registered result), we had better leave this loop to run on the host: firstly, it does not process a sufficient amount of data to saturate the GPU*, and secondly, the number of computational operations is not much bigger than the number of memory accesses. We will lose time transferring data to and from the GPU and manipulating it in global memory.

*At least with HD resolution, a grid has only several tens of thousands elements, which is small by the standards of a GPU and does not help in hiding its latencies.

- The L-BFGS method: optimizing this minimization algorithm goes beyond the scope of the thesis.
- Drawing results: if this code goes into production, our simple drawing routines will not be used at all (see Section 3.3.1).

5.3 CPU version

For a fair comparison, as the sequential version we will use the CUDA one with kernels wrapped in `for` loops and the texture substituted with a simple image. To test the parallel CPU version, we have added OpenMP directives to `for` loops and ensured the correct atomic addition where needed.

5.4 Speed tests

We have tested our algorithm on two inputs with different number of vertices and color samples. They are listed in Table 5.1.

	The smaller instance (#1)	The larger instance (#2)
# of cells	2067	16847
# of vertices	2263	17221
# of color samples	74412	606492

Table 5.1: Parameters of test task instances

Every frame in our test sequence has a resolution of 1200×1080 pixels. In the smaller task instance, the object being tracked is located far away from the camera taking roughly 1/15 of the viewport. In the larger task instance, the object is close to the camera and occupies approximately half of the screen. We have measured the average time based on 10 consecutive frames.

Our test PC has an Intel Core i7-8750H with 6 cores and 12 threads and a GeForce GTX 1070 Max-Q graphics card – an underclocked laptop version of GTX 1070.

All tests have been run on Ubuntu 18.04 with `g++` 7.3.0 and CUDA 10.0.

5.4.1 Raw error computation speed test

In this test, we look at how fast the algorithm can calculate one sum of squared errors together with its gradient. We do not consider any allocations, transfers, or anything else besides the two functions that compute color and rigidity differences. We have averaged the results across both all iterations of

the L-BFGS method when registering one individual frame and all 10 frames we have processed. Timespans are given in milliseconds.

		Sequential	OpenMP	CUDA
#1	Error evaluation	7.584 ms	1.476 ms	0.133 ms
	Speedup	1x	5.14x	57.02x
		0.19x	1x	11.10x
#2	Error evaluation	66.673 ms	14.417 ms	0.784 ms
	Speedup	1x	4.62x	85.04x
		0.22x	1x	18.39x

Table 5.2: Measuring average time required to evaluate the error function and its gradient

The results are shown in Table 5.2. The OpenMP version cannot be exactly 6 times faster because of the presence of atomic addition and lower frequencies of CPU cores (when only one core is busy, Intel's Turbo Boost increases its speed).

We can see that the CUDA version provides a much more noticeable speedup when it has to process a larger amount of data. We assume it happens because with the smaller task instance the GPU is not saturated enough with data to effectively hide its latencies.

5.4.2 Frame processing speed test

Let us see how long it takes to register an entire frame. Here, we consider the following timespans:

- Per-iteration time mainly consists of copying a set of current vertices to the GPU, calculating the error with respect to them in kernels, copying the result back to host, and waiting for the L-BFGS algorithm to process it.

- Per-frame time includes interpolation of the grid based on the previous iteration, copying this updated data to CUDA, binding new image G to the texture object, and initializing and transferring L-BFGS variables.

- Per-video time includes allocations and deallocations of arrays related to CUDA and the L-BFGS algorithm; it also contains the time required to transfer to the GPU the data that will not change during the entire video sequence.

We have not measured time for:

- Grid construction: for the sake of simplicity, we use a very straightforward algorithm. If the LK-ARAP method goes into production, it should be replaced with either a more sophisticated routine for better object segmentation or an interactive tool for the user.
- Reading and writing images: these processes vary depending on the libraries we use and do not have a direct relation to the LK-ARAP algorithm.
- Drawing resulting images: see Section 3.3.1.

One of the problems we have encountered is that the L-BFGS method performs different number of iterations in CUDA and both CPU implementations. This is probably caused by accumulated errors in floating-point numbers: we use fast (imprecise) math mode for both versions, but the one provided with CUDA platform likely differs from the one supplied with plain `g++`. Moreover, we use lots of threads on the GPU (and 12 in case of OpenMP) and cannot guarantee the order in which numbers are added up. This can be relevant since some floating-point operations do not have to be associative due to roundoff errors.

To make the speed comparison fair, we have divided every error minimization timespan with its corresponding number of iterations and multiplied it by the average number of iterations across all 3 versions of the code. This should give a comparable approximation.

The speedup has been computed based on the time required for error minimization plus per-frame operations. The results are listed in Table 5.3. Per-video time intervals are presented separately in Table 5.4. They do not scale with the length of the video. We will discuss them shortly.

When benchmarking the algorithm together with sequential parts of code, we cannot expect results to be as good as they are in Table 5.2. Yet the CUDA version still delivers a noticeable speedup. Its per-frame overhead is much bigger when compared to the one of the CPU counterparts because of all the required data transfers. Although it is not entirely correct to add per-frame overhead (because it does not scale with the number of iterations of the L-BFGS method), these timespans provide a good example of what to expect performance-wise in a real scenario.

Finally, in Table 5.4 there are per-video overheads. The more frames we register while estimating optical flow, the less impact they will have on the speedup. You can see that even if we add these values to the time of every individual frame, the CUDA version will still be faster than parallelized CPU code. If the L-BFGS method performs much fewer iterations than in our current tests,

		Sequential	OpenMP	CUDA
#1	Preparation	0.01 ms	0.01 ms	1.23 ms
	Error minimization	272.11 ms	62.61 ms	9.59 ms
	Speedup	1x	4.35x	25.15x
		0.23x	1x	5.79x
#2	Preparation	0.07 ms	0.08 ms	2.28 ms
	Error minimization	8059.62 ms	1833.04 ms	203.45 ms
	Speedup	1x	4.40x	39.18x
		0.23x	1x	8.91x

Table 5.3: Measuring average time required to register a frame

the overhead of the CUDA version can potentially have a more negative impact on the speedup, though. The OpenMP version has slightly larger time values than the fully sequential one since it has to allocate additional space for CPU-thread-local partial sums.

	Sequential	OpenMP	CUDA
Overhead (#1)	0.01 ms	0.02 ms	1.07 ms
Overhead (#2)	0.07 ms	0.49 ms	3.25 ms

Table 5.4: One-time-per-video overhead

5.5 Summary

We have partially reimplemented the LK-ARAP method for NVIDIA GPUs and managed to achieve a sufficient performance boost. The question of whether dual numbers are suitable for doing math on graphics cards is still open, though. We have been lucky enough to have an algorithm that uses relatively few dual numbers with few non-zero imaginary indices, which fit the registers perfectly and leave us with enough warps per multiprocessor. Deterministic behavior of the algorithm at each step allows us to implement specific optimized versions of operations on dual numbers. Since the consistency of the imaginary part is not required, we can take advantage of static indexing by carefully using template programming. Overall, our results prove that with some certain problems one can definitely use dual numbers on graphics cards and even obtain a noticeable speedup. However, we cannot be sure that the situation will be the same in the general case.

In the assignment, we were tasked to optimize the algorithm so it should be able to estimate optical flow "at interactive rates". We think this statement is too vague to give an unambiguous answer of whether we have achieved this goal or not. We have done our best to make the algorithm as fast as possible, but can $\approx$200 ms per frame be considered interactive? We have to leave this question to the readers. The required time also heavily depends on the input video sequence or, more precisely, on the number of iterations the L-BFGS method decides to perform.

Chapter 6

Quality evaluation

In this chapter, we will present side-by-side the resulting images obtained with all algorithms we have reviewed in the thesis; we will also numerically evaluate and compare their quality.

6.1 Ways to evaluate the quality

There exist different ways of how we can evaluate the quality of estimated optical flow. The most obvious way to do it is to compare our flow results with the correct (ground-truth) values. The problem is that we do not have ground truth for our sequence. In fact, it is very hard to find *any* video with correctly precomputed optical flow to compare with. We could have tried [2], for example, but their datasets are not suitable for us for the following reasons:

- Sequences are rather short – they contain only 8 frames. Since our task is to be estimating optical flow for as long as possible, with minimum number of manual corrections, these videos cannot really show how well (or not) our algorithm performs. The sample video used in all our tests consists of more than 200 frames.

- These sequences are not segmented. For our task, we assume that we have the mask for the object we want to track and, most likely, a green screen behind it. Detection of various objects prior to estimating optical flow for them can generally be a part of the challenge, but not in our case.

Due to our assumptions of mostly unchanged lighting conditions and texture look throughout the whole sequence, we can compare the final synthesized frame with the corresponding real one from the video – this will be our test. We will also present the resulting images you have seen in different chapters

and sections of this thesis side by side – this should help to evaluate them subjectively.

6.2 Comparison with the real frame

We compare only the region of the image where the object is located according to the mask of this frame. We have also rescaled pixel intensities to be from the $[0, 1]$ interval. As dissimilarity measures, we have used the sum of absolute differences (SAD):

$$\sum_{\boldsymbol{x}\in\Omega} \left|F(\boldsymbol{x}) - G(\boldsymbol{x})\right|,$$

the sum of squared differences (SSD):

$$\sum_{\boldsymbol{x}\in\Omega} \left[F(\boldsymbol{x}) - G(\boldsymbol{x})\right]^2,$$

and normalized cross-correlation (NCC):

$$\sum_{\boldsymbol{x}\in\Omega} \widehat{F}(\boldsymbol{x})\widehat{G}(\boldsymbol{x}), \quad \widehat{F}(\boldsymbol{x}) = \frac{F(\boldsymbol{x}) - \overline{F}}{\sqrt{\sum_{\boldsymbol{y}\in\Omega}\left[F(\boldsymbol{y}) - \overline{F}\right]^2}}, \quad \widehat{G}(\boldsymbol{x}) = \frac{G(\boldsymbol{x}) - \overline{G}}{\sqrt{\sum_{\boldsymbol{y}\in\Omega}\left[G(\boldsymbol{y}) - \overline{G}\right]^2}},$$

where $\overline{F}$ and $\overline{G}$ represent mean intensities of pixels from the region of interest Ω given by the mask.

The results are shown in Table 6.1.

	SAD	SSD	NCC
The LK alg.	177497	64700	0.435
The MRF-based alg.	171034	56599	0.461
SIFT flow	130493	33344	0.609
PWC-Net	196174	73160	0.400
The ARAP alg.	109827	27527	0.668
The LK-ARAP alg.	**94608**	**21947**	**0.751**

Table 6.1: Numerical quality evaluation

The LK-ARAP image registration algorithm shows the best scores according to all 3 measures: it has the lowest dissimilarity and the highest correlation with the real frame. The base ARAP registration holds the second place.

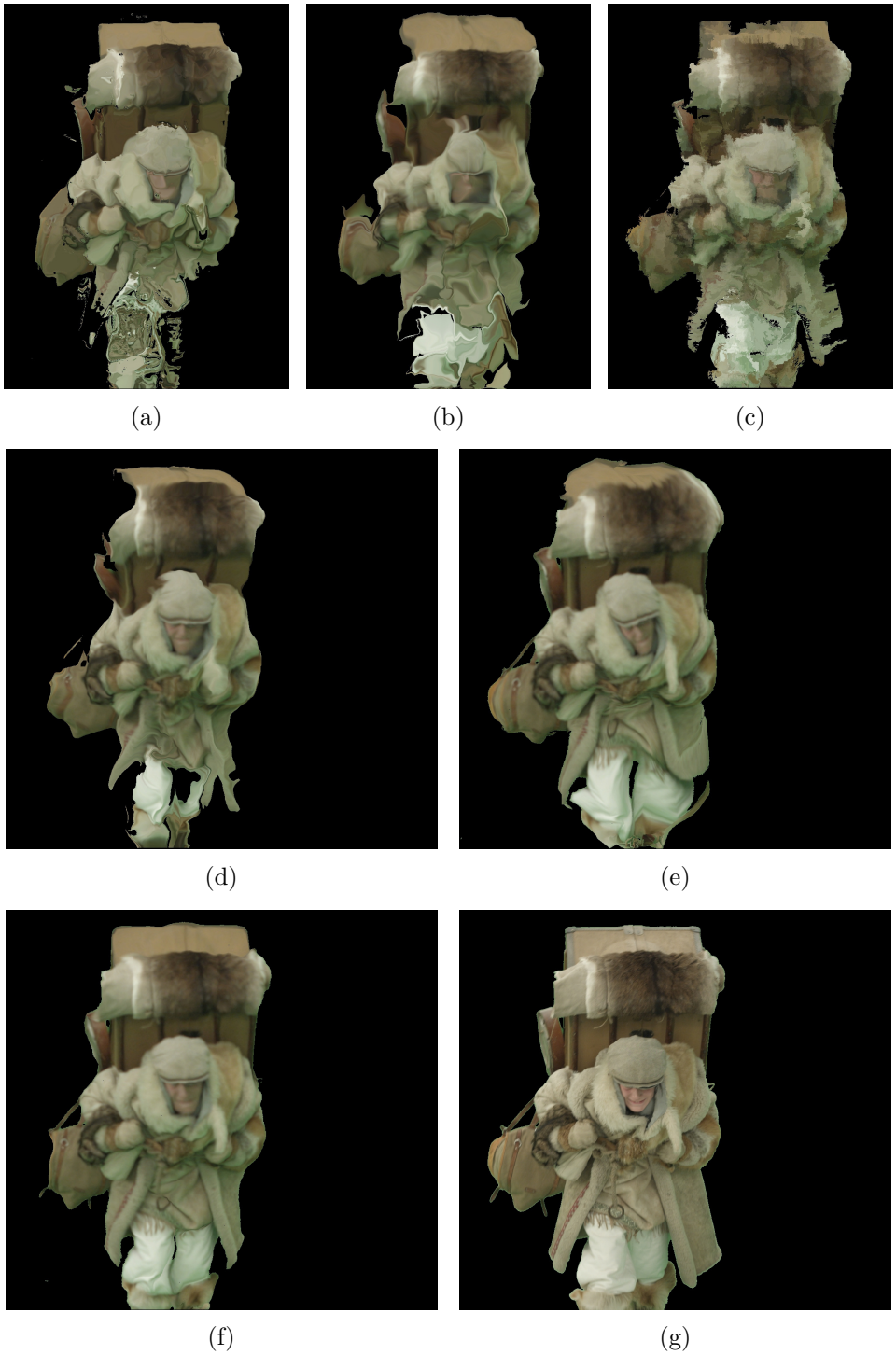


Figure 6.1: Results of all reviewed algorithms. (a) The Lucas–Kanade method. (b) Registration through MRFs. (c) SIFT flow. (d) PWC-Net. (e) ARAP registration. (f) LK-ARAP registration. (g) The real (reference) frame (© MAUR film).

It is hard to explain the poor scores of PWC-Net results. We suppose it is because of the fact that the overall size of the object was reduced; thus, we have many black pixels on the sides, which significantly increase the total dissimilarity score. Otherwise, the texture is well preserved – much better than with the three previous algorithms.

6.3 Subjective comparison

Subjectively, we think that the LK-ARAP image registration algorithm produces the most accurate results. You can have a look at Figure 6.1 and pick the winner according to your own taste.

Conclusion

During the course of this thesis, we have familiarized ourselves with the problem of optical flow estimation and a number of possible solutions to it. We have reviewed 6 vastly different methods and tested them on our video sequence.

Most attention has been paid to the LK-ARAP image registration algorithm – one of the new techniques that shows the best results among all other methods explored in this thesis. We have described in detail not only the algorithm proper but also the minimization strategy for its error function, which involves the L-BFGS quasi-Newton method and automatic differentiation with the help of dual numbers. We have also extensively described the two other methods the LK-ARAP algorithm is based on – the Lucas–Kanade and the ARAP image registration ones.

We have partially rewritten the LK-ARAP algorithm for CUDA platform. According to our speed tests, the new GPU-based code runs noticeably faster even when compared to the parallelized CPU version. We have made a list of all GPU-specific optimizations applied.

And last but not least, we have numerically evaluated the quality of results obtained with all 6 reviewed methods. The LK-ARAP algorithm has won with all three dissimilarity measures considered in Section 6.2.

On our way to implementing the ARAP and the LK-ARAP methods, we have looked into several auxiliary rendering-related concepts: the edge function, the PNPOLY algorithm, mean value coordinates, and affine and projective transformations.

The author considers that all goals from the assignment have been achieved. Yet, further improvements are still possible. For example, we can experiment with the LK-ARAP algorithm and its integral parts. The speed of the CUDA version may also be open for improvement with even more sophisticated approaches.

Bibliography

[1] Radke, R. J. *Computer Vision for Visual Effects*. Cambridge University Press, 2013, ISBN 978-0-521-76687-6.

[2] Baker, S.; Scharstein, D.; et al. Optical flow. In: *The Middlebury Computer Vision Pages*. Available from: `http://vision.middlebury.edu/flow/`

[3] Xu, S.; Zhang, F.; et al. PM-PM: PatchMatch With Potts Model for Object Segmentation and Stereo Matching. *IEEE Transactions on Image Processing*, volume 24, no. 7, 2015: pp. 2182–2196.

[4] Kim, T. H.; Lee, H. S.; et al. Optical Flow via Locally Adaptive Fusion of Complementary Data Costs. In *Proceedings of IEEE International Conference on Computer Vision*, 2013, pp. 3344–3351.

[5] Xu, L.; Jia, J.; et al. Motion Detail Preserving Optical Flow Estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 34, no. 9, 2012: pp. 1744–1757.

[6] Chen, Z.; Jin, H.; et al. Large Displacement Optical Flow from Nearest Neighbor Fields. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 2443–2450.

[7] Stoll, M.; Volz, S.; et al. Joint trilateral filtering for multiframe optical flow. In *Proceedings of IEEE International Conference on Image Processing*, 2013, pp. 3845–3849.

[8] Lucas, B. D.; Kanade, T. An Iterative Image Registration Technique with an Application to Stereo Vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, volume 2, 1981, pp. 674–679.

[9] Goshtasby, A. A. *Image Registration: Principles, Tools and Methods*. Springer-Verlag London, 2012, ISBN 978-1-4471-2458-0.

[10] Baker, S.; Matthews, I. A. Lucas–Kanade 20 Years On: A Unifying Framework. *International Journal of Computer Vision*, volume 56, no. 3, 2004: pp. 221–255.

[11] Glocker, B.; Komodakis, N.; et al. Dense image registration through MRFs and efficient linear programming. *Medical image analysis*, volume 12, 2008: pp. 731–741.

[12] Glocker, B. M. *Random Fields for Image Registration*. Dissertation thesis, Technical University of Munich, 2011.

[13] Komodakis, N.; Tziritas, G.; et al. Fast, Approximately Optimal Solutions for Single and Dynamic MRFs. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2007, pp. 1–8.

[14] Glocker, B.; Komodakis, N. mrf-registration.net. Mathématiques Appliquées aux Systèmes (MAS), Ecole Centrale de Paris, France, © 2008–2013. Available from: `http://www.mrf-registration.net/deformable/index.html`

[15] Liu, C.; Yuen, J.; et al. SIFT Flow: Dense Correspondence Across Scenes and Its Applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 33, no. 5, 2011: pp. 978–994. Available from: `https://people.csail.mit.edu/celiu/SIFTflow/`

[16] Lowe, D. G. Object Recognition from Local Scale-Invariant Features. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, 1999, pp. 1150–1157.

[17] Sun, D.; Yang, X.; et al. PWC-Net: CNNs for Optical Flow Using Pyramid, Warping, and Cost Volume. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 8934–8943.

[18] NVIDIA Research Projects. PWC-Net: CNNs for Optical Flow Using Pyramid, Warping, and Cost Volume. © 2018 NVIDIA Corporation. Available from: `https://github.com/NVlabs/PWC-Net`

[19] Sýkora, D.; Dingliana, J.; et al. As-Rigid-As-Possible Image Registration for Hand-Drawn Cartoon Animations. In *Proceedings of the 7th International Symposium on Non-photorealistic Animation and Rendering*, 2009, pp. 25–33.

[20] Rowland, T. Orthogonal Transformation. From MathWorld – A Wolfram Web Resource, created by Eric W. Weisstein. Available from: `http://mathworld.wolfram.com/OrthogonalTransformation.html`

[21] Rowland, T. Orthogonal Matrix. From MathWorld – A Wolfram Web Resource, created by Eric W. Weisstein. Available from: `http://mathworld.wolfram.com/OrthogonalMatrix.html`

[22] Schaefer, S.; McPhail, T.; et al. Image Deformation Using Moving Least Squares. *ACM Transactions on Graphics*, volume 25, no. 3, 2006: pp. 533–540.

[23] Pineda, J. A Parallel Algorithm for Polygon Rasterization. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, 1988, pp. 17–20.

[24] Franklin, W. R. PNPOLY – Point Inclusion in Polygon Test. 2018, W. Randolph Franklin (WRF), © 1994–2006. Available from: `https://wrf.ecse.rpi.edu/Research/Short_Notes/pnpoly.html`

[25] Schirra, S. How Reliable Are Practical Point-in-Polygon Strategies? In *Proceedings of the 16th Annual European Symposium on Algorithms*, 2008, pp. 744–755.

[26] Floater, M. S. Mean value coordinates. *Computer Aided Geometric Design*, volume 20, no. 1, 2003: pp. 19–27, ISSN 0167-8396.

[27] Nocedal, J.; Wright, S. J. *Numerical Optimization*. Springer-Verlag New York, second edition, 2006, ISBN 978-0-387-40065-5.

[28] Liu, D. C.; Nocedal, J. On the Limited Memory BFGS Method for Large Scale Optimization. *Mathematical Programming*, volume 45, no. 1-3, 1989: pp. 503–528.

[29] Piponi, D. Automatic Differentiation, C++ Templates, and Photogrammetry. *J. Graphics, GPU, & Game Tools*, volume 9, 2004: pp. 41–55.

[30] Okazaki, N.; Nocedal, J. libLBFGS: C library of limited-memory BFGS (L-BFGS). © 1990 Jorge Nocedal, © 2007–2010 Naoaki Okazaki. Available from: `https://github.com/chokkan/liblbfgs/`

[31] Cheng, J.; Grossman, M.; et al. *Professional CUDA C Programming*. Wrox Press, 2014, ISBN 978-1-118-73932-7.

[32] Milakov, M. GPU Pro Tip: Fast Dynamic Indexing of Private Arrays in CUDA. In: *NVIDIA Developer Blog*. 11 February 2015. Available from: `https://devblogs.nvidia.com/fast-dynamic-indexing-private-arrays-cuda/`

[33] Savas, N. Template Metaprogramming: Compile time loops over class methods. In: *Medium*. 23 December 2016. Available from: `https://medium.com/@savas/template-metaprogramming-compile-time-loops-over-class-methods-a243dc346122`

Appendix A

Acronyms

2D Two-dimensional

ARAP As-rigid-as-possible

BFGS Broyden, Fletcher, Goldfarb, Shanno

CNN Convolutional neural network

CPU Central processing unit

CUDA Compute Unified Device Architecture

DoG Difference-of-Gaussians

Fast-PD Fast primal-dual

GPGPU General-purpose computing on graphics processing units

GPU Graphics processing unit

HD High-definition

L-BFGS Limited-memory BFGS

LK-ARAP Lucas–Kanade as-rigid-as-possible

LP Linear programming

MRF Markov random field

NCC Normalized cross-correlation

OpenGL Open Graphics Library

OpenMP Open Multi-Processing

PNPOLY Point inclusion in polygon test

PWC Pyramid, warping, cost volume

RGB Red, green, blue

SAD Sum of absolute differences

SIFT Scale-invariant feature transform

SSD Sum of squared differences

Appendix B

Contents of enclosed CD

- `thesis`
  - `src` ..... the source code of the thesis in LaTeX
  - `DP_Laskov_Boris_2019.pdf` ..... the thesis in PDF
- `programs`
  - `LK_translation` ..... the code for the "Radio City" example
  - `LK_affine` ..... the code for the "Gifts 66" example
  - `LK_optical_flow` ..... flow estimation with the LK method
  - `ARAP_optical_flow` ..... flow estimation with the ARAP method
  - `LK-ARAP-CUDA` ..... CUDA impl. of the LK-ARAP method
  - `LK-ARAP-sequential` ..... sequential impl. of the LK-ARAP method