Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Autonomous Systems Lab
Prof. Roland Siegwart

Bachelor Thesis

Embedded Photometric Visual Odometry

Spring Term 2015

Supervised by:
Jörn Rehder
Pascal Gohl

Author:
Samuel Bryner

Declaration of Originality

I hereby declare that the written work I have submitted entitled

Embedded Photometric Visual Odometry

is original work which I alone have authored and which is written in my own words.[1]

Author

Samuel	Bryner

Student supervisors

Jörn	Rehder
Pascal	Gohl

Supervising lecturer

Roland	Siegwart

With the signature I declare that I have been informed regarding normal academic citation rules and that I have read and understood the information on 'Citation etiquette' (`https://www.ethz.ch/content/dam/ethz/main/education/rechtliches-abschluesse/leistungskontrollen/plagiarism-citationetiquette.pdf`). The citation conventions usual to the discipline in question here have been respected.

The above written work may be tested electronically for plagiarism.

Place and date	Signature

[1]Co-authored work: The signatures of all authors are required. Each signature attests to the originality of the entire piece of written work in its final form.

Contents

Abstract

This work investigates how an FPGA can assist a general purpose processor to run demanding algorithms on a computationally constrained platform. A photometric visual odometry approach is implemented on a Xilinx Zynq-7020 ARM CPU and integrated with an existing semi-global matching core on the FPGA to provide accurate visual odometry at around 5 Hz. It is shown that there is still potential in offloading work onto an FPGA or multiple cores.

Symbols

Symbols

$I(\boldsymbol{x})$	intensity image
$D(\boldsymbol{x})$	disparity image

stereo camera intrinsics:

f	focal length
$\boldsymbol{c}$	principal point
b	baseline

warping:

π^{-1}	back-projection operator, mapping a pixel with disparity value into 3D space
π	projection operator, mapping a point in 3D space onto the camera plane
$\boldsymbol{T}$	6-DOF transformation (rotation and translation)
$\boldsymbol{J_T}$	3×6 Jacobian of the transformation operator $\boldsymbol{T}$
$\boldsymbol{J_\pi}$	2×3 Jacobian of the projection function π
$\boldsymbol{J_I}$	1×2 Jacobian of the intensity image sampling

Indices

C	current frame
P	previous frame
x, y, z	world coordinates ($\in \mathbb{R}^3$, meters)
u, v	coordinates in camera plane ($\in \mathbb{R}^2$, pixels)

Acronyms and Abbreviations

ETH	Eidgenössische Technische Hochschule
ASL	Autonomous Systems Lab at ETH
IMU	Inertial Measurement Unit
SGM	Semi-Global Matching
SLAM	Simultaneous Location And Mapping

Chapter 1

Introduction

1.1 Motivation

Robots are generally constrained in their computational power and energy usage. A powerful method is to offload computation onto an FPGA, which is usually a much more efficient than a general purpose processor. However, programming an FPGA is not straightforward and integtrating it with code running on a CPU can be tricky.

In this work, a novel example of such an integration is provided by running a semi-global stereo matching [1] core developed in [2] on the FPGA and using its output for a photometric visual odometry algorithm [3] running on the CPU.

This approach of photometric odometry does not track a sparse set of features, as is usuallty done in visual odometry, but instead warps the full image to find a perspective where the warped image matches the previous frame.

Photometric odometry is very robust against various perturbations, such as blur, occlusions or changes in lighting [4]. It is also more accurate than feature-based methods, as it incorporates every pixel instead of throwing out most of the data. This also increases computational load however. Visual odoemtry in general requires a well structured environment, as for example homogenous walls are basically impossible to track.

This "dense" approach is well suited for offloading to an FPGA, as most parts are highly parallelizable. Note tough, that this is not the most efficient way to do embedded odometry, as a lot more data has to be processed. The main goal was to explore how an FPGA and a general purpose processor can be integrated on an embedded device and to ascertain the potential for further optimizations by transfering more parts to the FPGA.

A visual-inertial sensor developed by the ASL [5] is used which features a Xilinx Zynq 7020 SoC consisting of a dual-core ARM Cortex A9 and an ARTIX-7 FPGA. The sensor has a wide-angle stereo camera with a resolution of 752×480 pixels with syncronized global shutters as well as a high-precision inertial measurement unit, which was not used here. This vi-sensor is not only small and lightweight (133×57mm, 130 g), it is also power-efficient, consuming around 5 W.

1.2 Related Work

Photometric odometry as initially developed by Comport et al. in [3] has been implemented in [6] to use data from the SGM core but running on a powerful PC instead.

In [7], feature-based odometry running on the vi-sensor is developed, which uses the FPGA for corner detection. While very similar in result and method, a completely different and much more 'sparse' algorithm is used. A real-time perfomance of 10 Hz is achieved by using vector instructions and fusing inertial measurements. A similiar implementation [8] uses a Texas Instrument SoC, developed for smartphones.

Other approaches to photometric odoemtry rely on different means to get depth data, such as an RGB-D sensor [9]. Usually, active systems are used which increase power consumption, are either expensive (Laser), low resolution (time of flight) or work only in constrained scenery (structured light). Howevery, RGB-D sensors can be more accurate, cover longer ranges and consume less computing power.

Chapter 2

Method

2.1 Overview

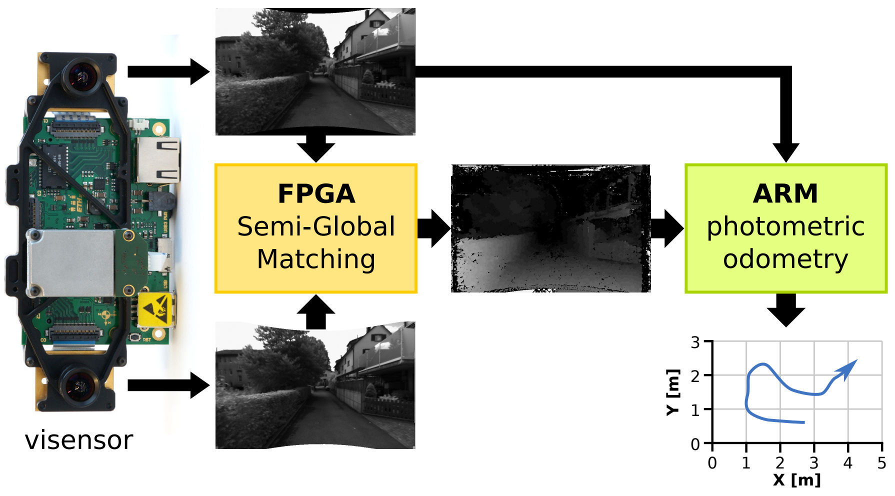


Figure 2.1: schematic overview of the whole system

The visensor provides a stream of frames, each of which consists of a stereo pair of grayscale intensity data [1]. A semiglobal stereo matching core developed in [2] running on the FPGA processes these and produces a disparity image which assigns every pixel the disparity between the two cameras. The FPGA also provides a rectified camera image.

This pair of intensity and disparity images is conceptually equivalent to a three dimensional point-cloud, as we can calculate the distance to the camera for every pixel from the disparity data. This is in turn makes it possible to render the point-cloud from an arbitrary perspective, allowing us to look at an image as if it was recorded from a different angle.

To estimate the ego-motion between two frames we can thus look for a perspective

[1]Grayscale instead of full-color images are used, because the information gain from colors is offset by the loss of resolution. However, the approach described here would work similarly with colored images.

where the re-rendered image looks the same as the previous frame. The movement of the virtual camera will then correspond to the actual, physical movement of the sensor.

The problem is formulized as a minimization problem, by subtracting the intensities from the previous frame with the intensities of the current frame sampled at the warped pixel locations. This way, a photometric error is calculated, measuring the similarity of the warped current frame with the previous one.

Note that this approach assumes Lambertian reflectance and constant illumination (also known as photo-consitency), which implies that points should have the same brightness, regardless of viewing angle.

2.2 Warping Pipeline

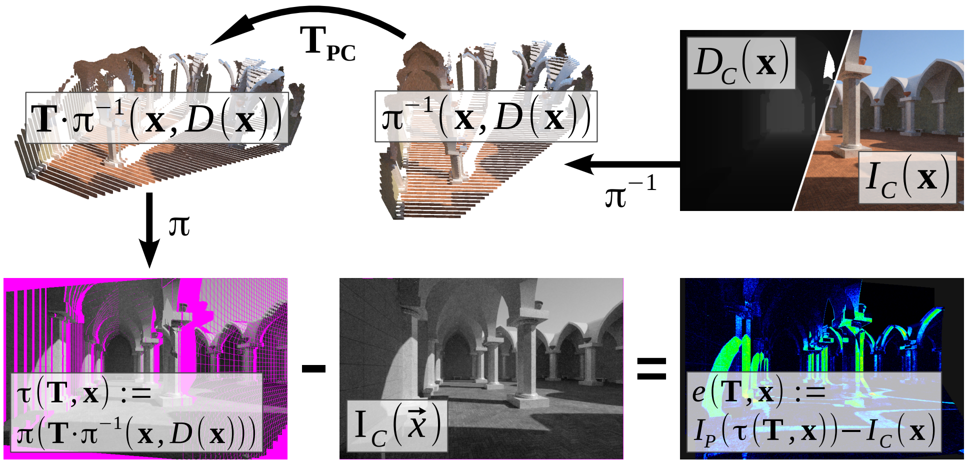

$\mathbf{T}_{PC}$
$\mathbf{T}\cdot\pi^{-1}(\mathbf{x}, D(\mathbf{x}))$
$\pi^{-1}(\mathbf{x}, D(\mathbf{x}))$
$D_C(\mathbf{x})$
$I_C(\mathbf{x})$
π^{-1}
π
$\tau(\mathbf{T},\mathbf{x}) := \pi(\mathbf{T}\cdot\pi^{-1}(\mathbf{x}, D(\mathbf{x})))$
-
$I_C(\vec{x})$
=
$e(\mathbf{T},\mathbf{x}) := I_P(\tau(\mathbf{T},\mathbf{x})) - I_C(\mathbf{x})$

Figure 2.2: the full warping pipeline (pink pixels are not sampled by any of the warped points)

This section closely follows [6].

Using an back-projection derived from the standard pinhole camera model, a

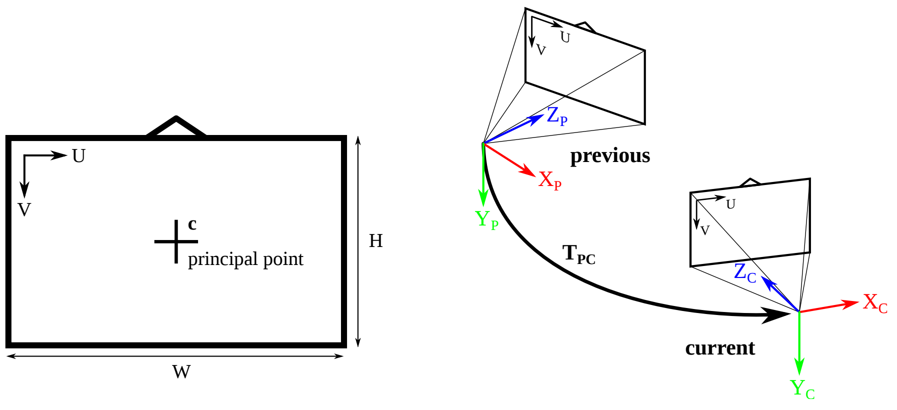


Figure 2.3: The coordinate systems used in this work.

point $\boldsymbol{x} = [x_u x_v]^\intercal$ in the camera image plane is be back-projected into a point $\boldsymbol{p_C} \in \mathbb{R}^3$ (in the current frame of reference; written using homogeneous coordinates) by using the pixel's disparity value $D_C(\boldsymbol{x})$:

$$\boldsymbol{p_C} = \pi^{-1}(\boldsymbol{x}, D_C(\boldsymbol{x})) := \frac{b}{D_C(x)} \begin{bmatrix} x_u - c_u \\ x_v - c_v \\ f \\ 1 \end{bmatrix} \tag{2.1}$$

where b is the stereo baseline, f the focal length and $\boldsymbol{c} = [c_u c_v]^\intercal$ the principal point of the camera.

This point $\boldsymbol{p_C}$ can now be moved into the previous camera frame by translating and rotating it, by writing the transformation from the previous into the current camera frame $\boldsymbol{T_{PC}}$ as a 4×4 homogeneous transformation matrix:

$$\boldsymbol{p_P} = \boldsymbol{T_{PC}} \cdot \boldsymbol{p_C} \tag{2.2}$$

A point in 3D space can be projected back onto the (now moved) camera image plane:

$$\boldsymbol{x}' = \pi(\boldsymbol{p_P}) := \frac{f}{p_z} \begin{bmatrix} p_x \\ p_y \end{bmatrix} + \boldsymbol{c} \tag{2.3}$$

This whole warping operator is summarized in a warping operator τ:

$$\boldsymbol{x}' = \tau(\boldsymbol{x}, \boldsymbol{T_{PC}}) := \pi(\boldsymbol{T_{PC}} \cdot \pi^{-1}(\boldsymbol{x}, D_C(\boldsymbol{x}))) \tag{2.4}$$

2.2.1 Representation of Transformation

While translations are mathematically very straightforward, rotations can be represented in numerous ways (Euler-angles, quaternions, rotation matrices, etc.) and proper derivation of the Jacobians can be tricky.

Fortunately, odometry works in a relative fashion without any absolute orientation and steps are very incremental as photometric odometry cannot handle more than a few degrees of rotation. This implies we do not have to deal with gimbal-lock and other mathematical hurdles of working in $SO(3)$. In this work, Tait-Bryan angles were used, following the ZYX convention (yaw, pitch, roll).

2.3 Minimization

Using τ, the error between the previous frame and the warped current frame is be defined as:

$$e(\boldsymbol{x}, \boldsymbol{T_{PC}}) := I_P(\tau(\boldsymbol{x}, \boldsymbol{T_{PC}})) - I_C(\boldsymbol{x}) \tag{2.5}$$

To estimate the motion between two frames, this photometric error term should be minimized for every pixel:

$$\hat{\boldsymbol{T}} = \underset{\boldsymbol{T}}{\operatorname{argmin}} \sum_{\boldsymbol{x} \in I_p} e(\boldsymbol{x}, \boldsymbol{T})^2 \tag{2.6}$$

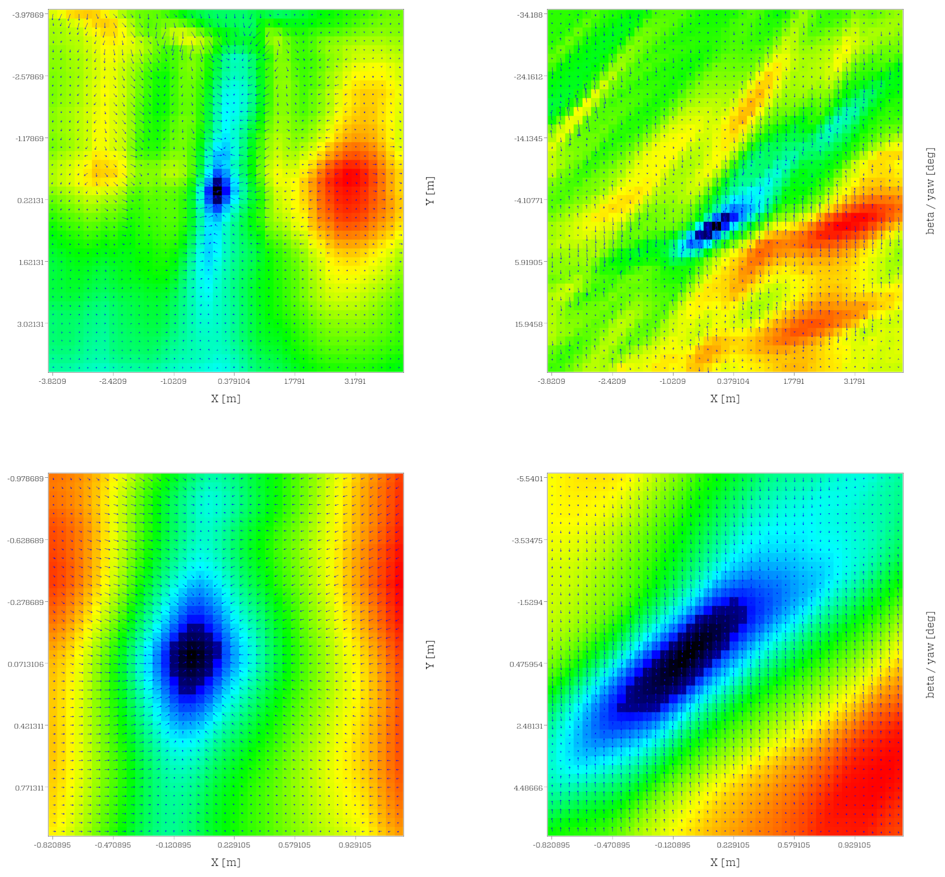


Figure 2.4: A visualization of the photometric error surface described by e generated from simulated camera data. Note how displacements of about half a meter or about 5° still lead to the correct minimum. Also note, that there are many local minima further out.

This equation can be solved for the estimated motion by applying standard optimization techniques, for example Gauss-Newton:

$$\boldsymbol{J}^\intercal \boldsymbol{J} \boldsymbol{\Delta T} = -\boldsymbol{J}^\intercal \boldsymbol{e}(\boldsymbol{T}) \tag{2.7}$$

Here, $\boldsymbol{J} \in \mathbb{R}^{N\times 6}$ are the stacked Jacobian matrices and $\boldsymbol{e}(\boldsymbol{T}) \in \mathbb{R}^N$ the vector of the error terms of all N pixels. This equation is iteratively solved for the increment $\boldsymbol{\Delta T}$ of the motion estimation after recalculating the error term and Jacobians from the new estimation.

The Jacobian is derived by applying the chain rule to the photometric error term 2.5. For a single pixel, we get a 1×6 Jacobian:

$$\boldsymbol{J} := \boldsymbol{J_I} \boldsymbol{J_\pi} \boldsymbol{J_T} \tag{2.8}$$

where $\boldsymbol{J_I} \in \mathbb{R}^{1\times 2}$ is the image derivative of the warped previous frame and is approximated using the image's gradient:

$$\boldsymbol{J_I} := \left.\frac{\partial I_p(\boldsymbol{x})}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\tau(\boldsymbol{x},\boldsymbol{T})} \approx \begin{bmatrix} \nabla_x I_p & \nabla_y I_p \end{bmatrix} \tag{2.9}$$

The term $\boldsymbol{J_\pi}$ is the 2×3 Jacobian of the projection function 2.3, evaluated at the warped 3D point:

$$\boldsymbol{J_\pi} := \left.\frac{\partial \pi(\boldsymbol{p})}{\partial \boldsymbol{p}}\right|_{\boldsymbol{p}=\boldsymbol{T}\cdot\pi^{-1}(\boldsymbol{x},D(\boldsymbol{x}))} = \begin{bmatrix} f/\boldsymbol{p_z} & 0 & -f\boldsymbol{p_x}/\boldsymbol{p_z}^2 \\ 0 & f/\boldsymbol{p_z} & -f\boldsymbol{p_y}/\boldsymbol{p_z}^2 \end{bmatrix} \tag{2.10}$$

$\boldsymbol{J_T}$ is the 3×6 Jacobian of the transformation operator T and the most costly term to compute:

$$\boldsymbol{J_T} := \left.\frac{\partial (\boldsymbol{T}\boldsymbol{p})}{\partial \boldsymbol{T}}\right|_{\boldsymbol{p}=\pi^{-1}(\boldsymbol{x},D(\boldsymbol{x}))} \tag{2.11}$$

2.4 Measures for Runtime Reduction

The algorithm described in the chapter 2 works well but suffers from slow performance and is not nearly realtime capable on an embedded device. Therefore, some optimization strategies are applied:

2.4.1 Image Pyramids

A common optimization technique is the use of multiple resolutions: Images are repeatedly filtered and downscaled by a factor of two (effectively quartering the number of pixels) by averaging over a 2×2 pixel block to generate a stack of increasingly smaller images (a 'pyramid').

The minimization is run on the smallest set of images and the resulting value is used as an initial value for the next bigger set of images. This greatly reduces the number of iterations required and enhances the convergence radius.

The image pyramid can also be used to trade a bit of accuracy for even more performance gain by simply aborting early and not using the full resolution at all. Throwing out the one or two lowermost levels usually incurs negligible loss of accuracy (see also section 3.1).

2.4.2 Pixel Selection by Image Gradient

We can further optimize away pixels which do not strongly influence the minimization such as points in homogenous image regions where $\|\nabla \mathbf{I}\| \approx 0$ and therefore $\|\boldsymbol{J_I}\| \approx 0$.

This is already provided to some extent by the semi-global matching alorithm, as pixels without strong gradients are usually hard to match and therefore often do not provide a disparity value.

Unfortunately, we still have to perform warping before we can sample the image gradient. Using the unwarped image as a further approximation of $\|\boldsymbol{J_I}\|$ turns out to bee too far off, as a motion of even a few degrees moves the image content by dozens of pixels and gradients do not overlap much in warped and unwarped images.

2.4.3 Handling Outliers

By robustly weighting the photometric error terms (here, Huber weights were used) outliers can be dampened to reduce the influence of occlusions, moving scenery or other noise, such as errors from the semi-global matcher [3]. This improves quality and stability with negligible performance penalty.

Another idea (which was not fully investigated) to handle occlusions is to use a Z-buffer to eliminate points which are behind others when warped. However, as photometric odometry works very incrementally, large occlusions are scarce.

2.4.4 Further Possible Optmiziations

A few measures which have not been investigated in this work but which provide good avenues for further optimizations:

Integration of an IMU

Modern visual odometry and SLAM systems such as [10] incorporate data from an inertial measurement unit using various complicated filtering schemes to increase accuracy. In contrast, using integrated acceleration values for pohotometric odometry would be easy by simply providing a good initial guess for the minimization process and could greatly speed up perfomance by reducing the number of required optimization iterations.

Keyframes

Instead of always matching the previous frame, keyframes can be used instead which are only updated when the relative motion grows above some threshold. This way, drift can be reduced and even completely eliminated when being more or less stationary.

Offload more work to FPGA or multi-core

Large parts of the photometric odometry algorithm are highly parallelizable and would profit from an implementation running on the FPGA. As this is one of the main point of this work, it is further discussed in section 3.2.

Chapter 3

Results

3.1 Qualitative Assessment

To assess the quality of photometric approach to visual odometry, an indoor circular trajectory was processed offline with OpenCV's SGM to ensure high-quality disparity data. It was also run trough the very accurate ASLAM algorithm [10] to provide something akin to ground-truth.

After a traveled distance of about 12 m a drift of about 20 cm has been accumulated.

The trajectory was recorded in an office environment, using a vi-sensor as described in section 1.1 with a baseline of 11 cm and a 112 ° diagonal field of view, providing a resolution of 752×480 pixels. To ensure photoconsistency, the shutter speed was fixed. The image gradient threshold was tuned to use at least 25 % of all pixels. The lowest resolution used in the image pyramid was 94×60 pixels, downscaling the orignial image three times. No error weighting was used, as the trajectory did contain few occlusions. The code was compiled for maximum performance, using `-O3 -mfpu=neon -mfloat-abi=softfp -ffast-math -fno-finite-math-only`.

Movement was restricted to a slow walking speed and was recorded with 30 FPS. On faster trajectories with larger movement between frames, the Gauss-Newton optimization diverges in badly lit or sparse parts with few image gradients, resulting in a completely wrong trajectory.

An important finding, which was already noted in [3], is that processing the camera images in their full resolution is not a necessity. As can be seen in figure 3.1, even using only a twice downscaled image still results in precise tracking while drastically improving performance.

Downscaling usually implies filtering and doing so alters the 3D structure. For this reason, [3] does not downscale the disparity values and samples them at the full resolution. Though when matching pose on the camera plane instead of in 3D space, downscaling the disparity values works as well.

It might be worth investigating how much of an effect on performance and quality downscaling the disparity images has. Disparity values are only read at integer coordinates and downscaling them might not be worth the runtime penalty.

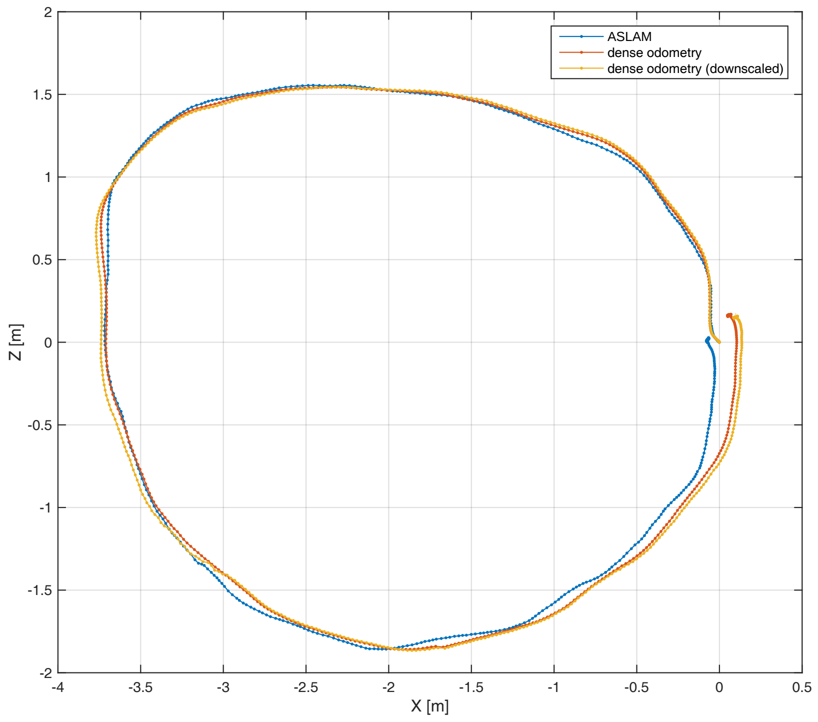


Figure 3.1: qualitative comparison of photometric odometry to ASLAM. The red trajectory was computed using full image resolution, while the orange one only uses $^1/_{16}{}^{th}$ of the available pixels by aborting early in the image pyramid, thus greatly speeding up runtime performance at negligible loss of accuracy.

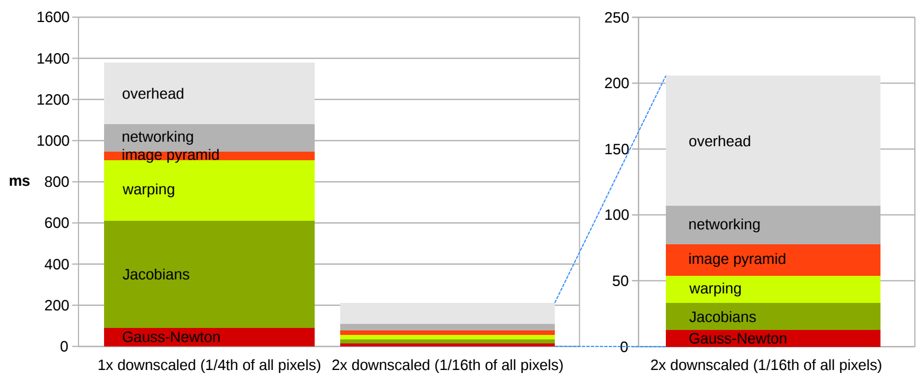


Figure 3.2: perfomance breakdown using early abort of the image pyramid

3.1.1 Keeping Pixel Numbers Stable

The optimizations described in section 2.4 can substantially reduce the pixel count, so much so that there are not enough for stable performance. Especially filtering pixels based on their image gradient as explained in section 2.4.2 requires a well-chosen threshold, as image gradients depend on the scene. [3] computes a histogram to select the N best pixels. An easier approach is to simply restart the current iteration with a lower threshold when the number of pixels gets too low and increasing it after a step with enough pixels, therefore implementing a proportional controller as described in [6]. The same problem also applies to other optimization parameters such as the size of the image pyramid.

3.2 Timing

The different parts of the algorithm have been timed by counting CPU cycles while running the full photometric odometry on a single 667 MHz Cortex A9 ARM core on the vi-sensor while moving around slowly. The time for the SGM is not shown in figure 3.2, as this part is running on the FPGA with the full 30 FPS provided by the cameras.

Using early abort in the image pyramid we can achieve an average run-time performance of about 5 Hz. This value not only depends on the algorithm's parameters such as max. pyramid levels, but also on the scene (more 'dense' environments which provide more structure converge faster) and on the movement speed. The bigger the steps, the more iterations are required for convergence, which is counterproductive as a longer running time implies an even bigger step while moving. This could potentially be addressed by incorporating integrated acceleration values of an IMU.

Another important point is that at least half the time is spent on building the image pyramid, warping pixels and computing the Jacobians. These parts are highly parallelizable because every pixel is completely independent of each other. This offers great potential for further optimizations: Moving the construction of image pyramids to the FPGA is simple. Warping and the calculation of Jacobians less so, because these steps involve a lot of floating-point arithmetic, but they would still greatly profit from another CPU core or usage of vector instruction (such as the ARM NEON instruction set).

Chapter 4

Conclusion

This work has shown that running photometric odometry on a computationally constrained platform is feasible and a perfomance of 5 Hz can be achieved by running semi-global matching on the FPGA and a few optimizations, mainly early abort in the image pyramid.

It also shows clearly, that there is still a big potential for moving even more parts onto the FPGA to one day achieve odometry out-of-the (small and lightweight) box.

Bibliography

[1] H. Hirschmüller, "Accurate and efficient stereo processing by semi-global matching and mutual information," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 2. IEEE, 2005, pp. 807–814.

[2] D. Honegger, H. Oleynikova, and M. Pollefeys, "Real-time and low latency embedded computer vision hardware based on a combination of fpga and mobile cpu," in *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on.* IEEE, 2014, pp. 4930–4935.

[3] A. I. Comport, E. Malis, and P. Rives, "Accurate quadrifocal tracking for robust 3d visual odometry," in *Robotics and Automation, 2007 IEEE International Conference on.* IEEE, 2007, pp. 40–45.

[4] A. Comport, M. Meilland, P. Rives *et al.*, "An asymmetric real-time dense visual localisation and mapping system," in *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on.* IEEE, 2011, pp. 700–703.

[5] J. Nikolic, J. Rehder, M. Burri, P. Gohl, S. Leutenegger, P. T. Furgale, and R. Siegwart, "A synchronized visual-inertial sensor system with fpga preprocessing for accurate real-time slam," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on.* IEEE, 2014, pp. 431–437.

[6] S. Omari, M. Bloesch, M. Burri, P. Gohl, M. W. Achtelik, and R. Y. Siegwart, "Real-time dense stereoscopic visual odometry," in *Workshop, Robotics: Science and Systems*, 2014.

[7] M. T. Dymczyk, "Visual-inertial motion estimation on computationally constrained platforms," Autonomous Systems Lab, Eidgenössische Technische Hochschule, Tech. Rep., 2014.

[8] S. B. Goldberg and L. Matthies, "Stereo and imu assisted visual odometry on an omap3530 for small robots," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on.* IEEE, 2011, pp. 169–176.

[9] C. Kerl, J. Sturm, and D. Cremers, "Robust odometry estimation for rgb-d cameras," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on.* IEEE, 2013, pp. 3748–3754.

[10] S. Leutenegger, P. T. Furgale, V. Rabaud, M. Chli, K. Konolige, and R. Siegwart, "Keyframe-based visual-inertial slam using nonlinear optimization." in *Robotics: Science and Systems*, 2013.

Appendix A

A Short Guide to the Codebase

A.1 Directory Overview

All figures, LaTeX and LibreOffice files for the presentation and this report are in thery respectively named directories. Other important folders are:

matlab Contains the full warping pipeline implemented in Matlab. Slow, but useful for debugging and automatic derivation of the Jacobians.

matlab/input Contains various example scenes for the simulation. The courtyard scene was downloaded from `http://www.luxrender.net/forum/viewtopic.php?f=14&t=6015` and can be used with LuxRenderer

cpp Implementation of all warping code in C++ and a few helpful GUIs and visualizations (including a client for the vi-sensor application).

arm Plumbing to run photometric odometry on vi-sensor. All my code is in the `src/odometry` subdirectory.

A.2 Blender Simulation

For debugging and testing purposes, Blender was used to create images with depth data and perfect ground truth. There are several scenes in `matlab/input`. Upon first opening a scene, the ground truth recording script must be registered once by running the `output\_frame\_pos.py` once (or by unsafely reloading the .blend-file by clicking on the warning in the Blender header menu).

This script will now run every time an animation is rendered and will output relative camera poses into a CSV file called `camera\_trajectory\_relative.csv` in the same directory. Upon first execution, it will also generate `camera\_intrinsics.csv` containing all important parameters of the virtual camera such as focal length and scaling of the raw disparity values.

The generated CSV files and the rendered images can be read by both the C++ and the Matlab implementation.

Note, that the export script expects the virtual camera to be at the object tree's root and should not be subordinated to another object. When using a path for animation, it has therefore to be baked into keyframes.

A.3 Matlab Implementation

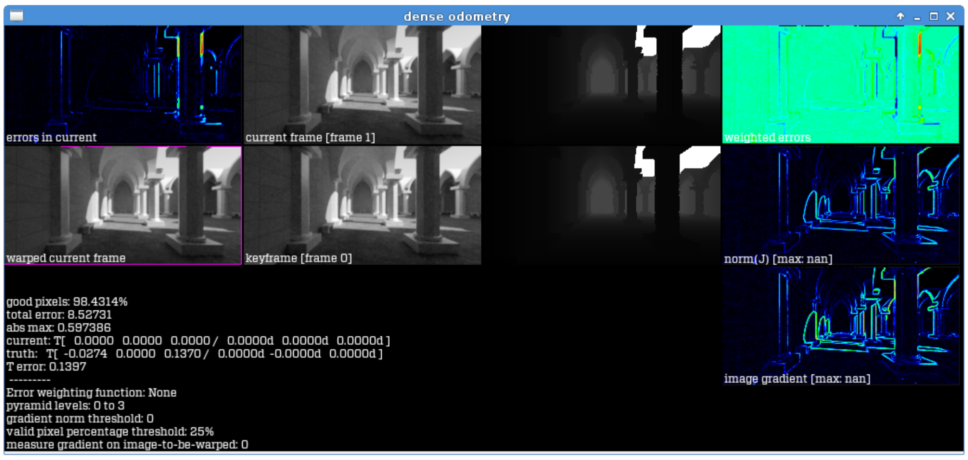
All Matlab functions expect their input data in column matrices (for example, a list of 100 3D points would have 3 rows and 100 columns). The Matlab codebase only works with simulated data and cannot use raw data from the vi-sensor.

Everything directly implementing warping as described in section 2.2 can be found in `core`. This warping pipeline uses Matlab's matrix operations for speed, but analytically derives the Jacobians for every iteration, which slows down things immensely.

A good starting point are the scripts in `test\_scripts`, which provide various examples of how to use the code: `run\_minimization.m` runs the full warping pipeline and Gauss-Newton minimization, while `test\_simple\_warping.m` simply plots the warped image for a given transformation.

Some unit-tests can be run by invocating `run\_unittests.m` in the root folder.

A.4 C++ Implementation



```
good pixels: 98.4314%
total error: 8.52731
abs max: 0.597386
current: T[  0.0000  0.0000  0.0000 /  0.0000d  0.0000d  0.0000d ]
truth:   T[ -0.0274  0.0000  0.1370 /  0.0000d -0.0000d  0.0000d ]
T error: 0.1397
 ---------
Error weighting function: None
pyramid levels: 0 to 3
gradient norm threshold: 0
valid pixel percentage threshold: 25%
measure gradient on image-to-be-warped: 0
```

Figure A.1: screenshot of simulation view, running with rendered data

The C++ implementation has a lot more features and is siginificantly faster than the version written in Matlab. It contains various useful tools and visualizations.

To compile and run the application, following libraries are required:

- Boost
- SFML 2
- Eigen 3
- OpenGL

To compile the code, cmake is used:

```
mkdir build
cd build
cmake -D CMAKE_BUILD_TYPE=Release ..
make dense_odometry
./dense_odometry
```

Specifying `CMAKE_BUILD_TYPE=Release` is optional, but greatly speeds up the code. There are afew tests which can be run by calling `make && make test`.

At the time of this writing, `main()` has to be manually edited to run the different applications.

There are two implementations of the photometric odometry: A normal one in `core/warp.cpp` and `core/minimization.cpp`, which is modular and can handle disparity and depth data, and an optimized version in `core/warp\_streamlined.cpp` and `core/minimization\_streamlined.cpp`, which re-uses as much computation as possible and only works with disparity data (and therefore cannot be used with data generated by the Blender simulation).

A.4.1 simulation

To view Gauss-Newton minimization in action, trying out different parameters, rendering cost surfaces etc. call `run_minimization()` in main. You can either load a rendered scene by specifing a scene directory (containing all the output generated by Blender) or load a rosbag file (containing image and disparity data).

The simulation view is operated using keyboard shourtcuts:

N advance to next step in scene

B or Shift-N go to previous step in scene

M enter a command-line menu to access other features, such as setting parameter values or rendering a cost-surface plot

space or P run/pause minimization

R reset current transformation to zero

T set current transformation to ground truth

D disturb current transformation by a small random amount (can be used to get minimization 'unstuck' from a local minima)

F5 run minimization from zero, without live visualization and using the image pyramid

F6 same as **F5**, but using the streamlined warping pipeline (only usable for real data with disparity images)

0 set error weighting function to none

1-6 set error weighting function to Huber weights, using various parameters

K overlay the warped image with the previous image, to visually judge the current transformation

S downsample images by a factor of two (moving to the next smaller pyramid level)

A reset images to full resolution

Numpad +/- scale view, to fit large images on to a small screen or to enlarge downscaled images (this does not affect the data in any way and just scales the GUI)

Numpad * reset view scaling

Numpad Numbers translate virtual camera (rotate with CTRL-Numpad)

F1 save warped image, gradients and Jacobian terms as PNG

F2 start or stop recording frames into `recording/` subdirectory

F12 reset GUI streching after the window has been resized

ESC quit

A.4.2 live view

`show_live_data()` shows data from the ARM code running on the vi-sensor. The visensor-node has to be active for receiving live tracking data. Various keybindings exist:

R clear recorded trajectory and reset orientation

Numpad +/- scale recorded trajectory

F1 store current trajectory to disk (`traj\_N.csv`)

PageUp/Down increase/decrease camera shutter time to brighten/darken the image (this is slow, as a system call has to be made to execute ROS' `dynamic_reconfigure`; Unfortunately no C++ API has been implemented for this)

S manually set shutter time trought terminal

space record current frame for later review in simulation view

return enter simulation view with recorded frames

F12 reset GUI streching after manually resizing window

ESC quit

A.4.3 batch processing of recordings

`write_trajectory_rosbag()` processes a recorded trajectory (either raw data directly from the vi-sensor, or data processed by OpenCV and optionally ASLAM). Outputs the tracking data as `measured\_trajectory.csv` and ASLAM ground-truth as `ground\_truth\_trajectory.csv` if available.

It automatically drops into live view afterwards with frames deemed as 'bad' (where a big transformation or a lot of iterations were noted, usually indicating failure of the Gauss-Newton minimization).

A.4.4 rendering error surface

To visualize the photometric error and the Jacobian, `draw_error_surface()` can be used to render a two-dimensional plot of cost surface, as for example shown in figure 2.4. This tool can also be called from live view by pressing **M** and choosing `12: render error surface`.

It expects two range parameters for the two dimensions of the plot. A range parameter consists of a dimension from 0 to 5, indicating which of the 6 transformation dimensions to iterate over (x, y, z, alpha/pitch, beta/yaw, gamma/roll), and a `linspace()` like range (from, to, number of steps in between).

Depending on the amount of steps, rendering an error plot can take a while. After the computations are complete, the plot is shown on screen and saved to disk. A few commands are available:

G show/hide gradients

Numpad +/- scale gradient arrows

F1 or S save current view (with gradients, if enabled) to disk

ESC or Q quit (and return to live view, if called from there)

A.5 ARM code

The C++ implementation can be cross-compiled to the vi-sensor's ARM Cortex A9 without any modifications by using the Xilinx SDK. The only ARM specific code is `src/odometry/odometry.cpp` which initializes the photometric odometry, passes along the data from camera and FPGA and transmits the tracked movement over the network.

A.5.1 communication between vi-sensor and PC application

To report the tracked movement, a fake camera is inserted (`src/odometry/utils/FakeCamSensor.cpp`) which is received by the visensor-node which passes it trough the ROS message system as a normal camera image. The PC app receives everything trough the ROS interface and extracts the `Telemetry` struct from the "image".

A.5.2 running the vi-sensor application

All parameters for the ARM code are set in `src/odometry/odometry.cpp`, a few important parameters can be overriden via command line tough: Smallest and biggest pyramid level can be set by using `--max-pyramid-level N` and `--min-pyramid-level N` (a minimum of 2 and a maximum 3 usually provide good and fast tracking). Transmitting the two camera images and disparity values over the network can be disabled with `--no-images`, which speeds up tracking by a few FPS at the cost of not being able to see what the sensor can see.

It is also important to disable all camera automatics. This can be done trough the ROS interface by executing:

```
rosrun dynamic_reconfigure dynparam set /visensor_node "{
        'individual_cam_config': 0, \
        'cam_agc_enable': 0, \
        'cam_aec_enable': 0,  \
        'cam_coarse_shutter_width': 300, \
        'penalty_1': 20, \
        'penalty_2': 200, \
        'threshold': 100, \
        'lr_check': 2 \
    }"
```

This is also executed by the live viewer application on startup and the live viewer has shortcuts available to change the shutter time at runtime.

A.5.3 timing

To measure performance, the code has been instrumented to count CPU cycles. To compile without the timing overhead, make sure `NO_TIMERS` is defined. Otherwise, measurement results are shown upon stopping the visensor-node on the host PC.