OpenSRD
Installation and user guide

Version 1.0

Lukas Engelen

March 27, 2017

Contents

Chapter 1

Introduction

OpenSRD, abbreviation of Open Source Shape from Refractive Distortion, is a software tool developed for the spatio-temporal reconstruction of a water surface. It started as a master thesis project by Lukas Engelen at the Hydraulics Laboratory of Ghent University and was further developed and validated in his subsequent PhD-research. Detailed information on the theoretical considerations, as well as the validation and accuracy assessment, can be found in the accompanying paper. The entire program is written in C++ of which the source code is freely available online[1] and licenced under GNU GPLv3 (see Appendix C). Any comments on how to improve to improve the program are more than welcome. In case you have suggestions or would like to contribute to the project, please contact Lukas Engelen[2]

The OpenSRD developers would like to acknowledge that the program makes use of three open source libraries, namely OpenCV (Bradski [3]), ALGLIB (ALGLIB Project [1]) and MINPACK (Burkardt [5]).

1.1 Need for accurate and image-based surface reconstruction techniques

Accurate measurement of the deformation of a liquid surface is important for numerous hydraulic and oceanographic research fields. Although numerical models offer the advantage of simulating with minimal effort much more variations in hydraulic and geometric boundary conditions, experimental research remains required to validate these numerical codes. Additionally, experimental research remains indispensable to discover new flow features and gain fundamental knowledge about the studied phenomena.

Conventional measurement techniques, such as capacitance, resistance or pressure gauges and ultrasonic distance sensors, only provide point measurements of the local water height. Additionally, the former are intrusive and therefore not recommended for a detailed study of the surface characteristics. In contrast, optical techniques allow to obtain information over an entire surface area without interference with the studied phenomenon. A single-view, image-based technique was pursued, making no use of laser-scanners or special optics. This allows to employ easily available, low-cost equipment without the safety issues related to lasers. As such, it is easily implementable in existing laboratory setups with only limited investment. Additionally, the presented program does not require any tracers or additives that need to be injected in the water. It therefore preserves transparency and can be combined with simultaneous velocity measurements such as PIV or PTV.

1.2 Short summary of the reconstruction methodology

With OpenSRD, an image-based reconstruction methodology is applied in which a known pattern, positioned at the bottom of the reconstructed volume, is captured by cameras located above the surface. The projection

[1] Available at: `https://github.com/lengelen/OpenSRD`

[2] Ir. Lukas Engelen, PhD student
E-mail: lukas.engelen@ugent.be
Hydraulics Laboratory, Civil Engineering Department Faculty of Engineering and Architecture, Ghent University
Sint-Pietersnieuwstraat 41 (Blok 5), B-9000 Gent, Belgium

of the pattern on the water surface, as seen by the cameras, becomes deformed due to the refraction of the viewing rays at the air-water interface. The distorted view of the fixed pattern depends on the shape of the water surface, whence the dynamically changing surface shape can be determined. The aforementioned methodology is combined with a low parameter surface model, based on the expected surface shape, which is fitted to the refractive disparities of the feature pattern located below the water. Finally, a mathematical description of the reconstructed surface area is obtained of which the spatial and temporal resolution is only limited by the available imaging equipment.

1.3 Installation of the experimental setup

1. Installation of a feature pattern
   The OpenSRD reconstruction methodology is based on the deformation due to refraction of a feature pattern, located below the free surface. To discern these feature points in an image of the water surface, a feature or corner detection algorithm is needed. Currently, a corner detection algorithm is implemented based on CHESS (Bennett and Lasenby [2]). It is specifically designed to locate checkerboard-patterns (i.e. 2×2 white and black squares) and is able to locate the internal corners robust and accurately. Therefore, it is required to create/place a feature pattern on the bottom of the reconstructed volume, for which the exact position of the feature points needs to be known. In case the user would like to use a different feature pattern, a user-defined feature detection algorithm needs to be implemented, as well as some necessary adjustments to the OpenSRD source files.

2. Installation of the illumination system
   To make the pattern distinguishable for the cameras, which see the pattern from above the water surface, good illumination of the feature plane is crucial. The most straightforward and easiest option is to light the pattern from the side/top. However, this has the disadvantage of possible reflections on the specular water surface which complicates feature detection. Our recommendation is therefore to light the pattern from below, using a uniform light panel as tank floor or place it below the bottom of a transparant tank. In case the white and black parts of the checkerboard pattern are respectively transparent and nontranslucent for the light source, the feature pattern should be clearly distinguishable without the risk of reflections on the surface. Another possibility is to use UV light and use a UV-sensitive pattern, which re-emits the energy of the UV light in the visible light spectrum. For more information, we refer to Engelen [6].

3. Calibration of the (multi-)camera system
   The calibration of the camera comprises the determination of the intrinsic camera parameters and distortion coefficients of each camera. This requires a sequence of images depicting a calibration grid, after which the the executable *Calibration* can be run (for more information, see section 3.2). Additionally, the extrinsic camera parameters need to be determined based on an image of a known reference grid. Make sure that the position of the cameras is not altered between this calibration step and the measurement campaign. The entire surface reconstruction methodology is highly dependent on the camera position in the chosen word coordinate system, which needs to be known as accurately as possible.

1.4 Data acquisition

1. Illumination of the feature pattern
   This can be done using a continuous or pulsed light source. Pulsed light has the advantage that in case high-lumen LED's or COB's are used, complications related to heat production are much smaller. Additionally, the LED's can be used in overdrive mode, resulting in more than 100% brightness of their theoretical brightness, which improves the contrast in the images. This also allows shorter shutter times and therefore reduces the risk of image blur.

2. Image acquisition
   The possible frame rate of the images depends on the image acquisition system, consisting of cameras, data link and storage device and determines the maximum temporal resolution of the surface reconstruction. Please keep in mind that also the shutter time needs to be short enough to avoid motion

blur, which requires strong lighting of the scene (previous step). Although decreasing the image size with binning sometimes allows a faster frame rate or larger covered area (for the same image size), this should be done with care as it decreases the reconstruction accuracy.

3. Local storage or data transmission
   The data (images) can then be stored in the camera memory or send through a data link to a local storage device (e.g. laptop).

As an example, the experimental set-up of the Hydraulics laboratory of Ghent University is described in the accompanying paper ().

1.5 Data processing

1. Feature detection
   Currently, a corner detection algorithm is implemented which is largely based on the CHESS-algorithm of Bennett and Lasenby [2] which employs a corner strength measure to locate checkerboard vetrices in the image. For more details, we refer to Appendix B.

2. Feature tracking/matching
   To obtain a temporal description of a dynamic surface, feature points need to be tracked over an image sequence. The location of a certain feature point is for every frame (image) predicted using a linear prediction model, based on its previous location and assuming a constant velocity within the image plane. In case a local maximum of the corner strength measure is sufficiently close to this estimated position, the corresponding maximum is accepted as the updated image point **q'**.

3. Multivariate optimization
   The optimization of the surface coefficients is done by running the Executable *SurfaceReconstruction*, using the error metric and optimization parameters that are chosen through the settings-file. The resulting list of time-dependent coefficients of the surface model can then be used to evaluate the local water height at every position in the reconstructed spatial domain.

1.6 Result

- Camera parameters (optional)
- Detected feature points per image (optional)
- Time-dependent surface coefficients
- Related average total error (collinearity or disparity difference metric) (optional)

1.7 Advantages of the technique

- Low-cost
- Non-intrusive
- Scalable
- Accurate and computational efficient
- Compatible with PTV or PIV velocity measurements
- Easily applicable to modal analysis of free surface oscillations

Chapter 2

Installation of the source code

2.1 Introduction

The reconstruction algorithm, briefly outlined in the previous chapter, is the implementation of a theoretical framework presented by Engelen [6]. It is written in C++ , and the different parts of the reconstruction are split over several modules that are coded in different source files and header files. OpenSRD employs functions and classes from other open-source libraries[1]. It is recommended to install and get familiar with these libraries before making changes to the OpenSRD program. To compile OpenSRD from source, it is required that OpenCV (version 3.0 or higher) is installed. In case you are on a Windows system, please read the instructions that are mentioned in the last section of this chapter carefully. Alternatively, you can download the compiled executables from Github[1] and go directly to Chapter 3.

2.2 Required third-party, open-source libraries

OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source BSD-licensed library, developed by Bradski [3], that includes several hundreds of computer vision and machine learning algorithms (Bradski and Kaehler [4]). The library is cross-platform and mainly written in C++ (although an older C interface exists) but contains bindings for other languages such as Python or JAVA. Some of the application areas include 2D and 3D feature toolkits, camera calibration, facial recognition systems, object identification and motion tracking, ... Different functions from the OpenCV library will be used for several aspects of the water reconstruction procedure:

- Camera calibration (estimation of the distortion coefficients)
- Camera pose estimation (estimation of the camera position and orientation w.r.t. a predefined reference system).
- Input and output (images, settings, ...)
- Object classes

ALGLIB

ALGLIB Project [1] is a cross-platform numerical analysis and data processing library and supports multiple programming languages, including C++. In general, the ALGLIB functions are used for several programming challenges such as:

- Data analysis
- Optimization and nonlinear solvers

[1] More information about the License agreements of these libraries can be found in Appendix C

- Interpolation and linear/nonlinear least-squares fitting
- Linear algebra (direct algorithms, EVD/SVD), direct and iterative linear solvers, Fast Fourier Transform and many other algorithms (numerical integration, ODE's, statistics, special functions)

OpenSRD employs an ALGLIB-implemented version of the Levenberg-Marquardt method. This method is used for the multivariate optimization of the surface coefficients of the chosen surface model. The algorithm uses a combination of the steepest descent method with a Newton-Raphson method in order to obtain maximum operational stability and rapid convergence towards the optimal solution. It also offers the possibility to set an upper and lower limit to the search interval of these coefficients, as well as the possibility to set a different optimization scale (differentation step, stopping condition). Finally, the algorithm uses a initial guess as starting position which has a minor influence on the final result but has a significant influence on the convergence rate. In case the frame rate of your camera is sufficient, a good initial guess, based on the previously processed time step, allows a more rapid convergence to the next solution due to the temporal smoothness of the water surface.

The ALGLIB's C++ source and header files are also included in the src directory found on Github[1]. These source files need to be compiled together with the other OpenSRD source files, which for most used compilers (GCC, MSVC, Sun Studio) does not require any additional settings. In other cases, pre-processor definition might be required for which we refer to the ALGLIB reference manual[1].

MINPACK

Additionally, the program employs a C++ version of MINPACK by Burkardt [5], originally a FORTRAN library by J. More, B. Garbow and K. Hillstrom, to solve a set of non-linear equations. The required source and header file (minpack.cpp and minpack.h), which contain an OpenSRD modified version of a non-linear solver, are also included in the src directory and need to be compiled together with the rest of the project.

2.3 Compiling OpenSRD on Linux (Ubuntu)

Installation of OpenCV

1. Start with the installation of the dependencies required for OpenCV and OpenSRD. This can be done through a terminal window using following commands or by using Synaptic Manager:

```
$ sudo apt-get -y install libopencv-dev build-essential cmake git
libgtk2.0-dev pkg-config python-dev python-numpy libdc1394-22
libdc1394-22-dev libjpeg-dev libpng12-dev libtiff4-dev libjasper-dev
libavcodec-dev libavformat-dev libswscale-dev libxine-dev
libgstreamer0.10-dev libgstreamer-plugins-base0.10-dev libv4l-dev
libtbb-dev libqt4-dev libfaac-dev libmp3lame-dev libopencore-amrnb-dev
libopencore-amrwb-dev libtheora-dev libvorbis-dev libxvidcore-dev
x264 v4l-utils unzip
```

2. Get the latest stable version of the OpenCV source code. This can be done manually by downloading the source archive from `http://opencv.org/downloads.html` and unpacking it. Alternatively, you can run the commands below to clone the OpenCV repository using git:

```
$ mkdir opencv
$ cd opencv
$ git clone https://github.com/opencv/opencv.git
```

3. After getting the source code, build OpenCV using CMake:

[1] `https://github.com/lengelen/OpenSRD`

```
$ cd ~/opencv
$ mkdir build
$ cd build
$ cmake -D CMAKE_BUILD_TYPE=Release -D CMAKE_INSTALL_PREFIX=/usr/local ..
$ make -j $(nproc)
$ sudo make install
```

4. Finalize the installation by running following command lines in the Terminal:

```
$ sudo /bin/bash -c 'echo "/usr/local/lib" > /etc/ld.so.conf.d/opencv.conf'
$ sudo ldconfig
```

5. After installing OpenCV, it is recommended that you reboot your system.

Installation of OpenSRD

1. Start with making the directory that will contain all the files that are related to OpenSRD:

```
$ mkdir ~/OpenSRD && cd ~/OpenSRD
```

2. Get the OpenSRD source code. This can directly be downloaded as a .zip file[1], which need to be unzipped in the OpenSRD directory. Alternatively, one can clone the repository from Github. This requires that git is installed (see step 1), after which the repository can be retrieved by following command:

```
$ git clone https://github.com/lengelen/OpenSRD.git
```

3. Enter the directory libOpenSRD and compile the library using CMake:

```
$ cd OpenSRD/libOpenSRD
$ cmake .
$ make all
```

This will result in two executables, *Calibration* and *SurfaceReconstruction*, which are located in the OpenSRD/libOpenSRD folder and are necessary for surface reconstruction.

2.4 Compiling OpenSRD on Mac OS X

Installation on Mac is very similar as for Linux. Follow the same instructions as the Linux guidelines, but use either MacPorts or Homebrew package manager where the apt-get package manager is used. Additionally, one can install CMake using the binary distribution, available as a *.dmg file from `http://cmake.org`. Once installed, simply run the following commands in the libOpenSRD/ directory:

```
$ cmake .
$ make all
```

2.5 Compiling OpenSRD on Windows

For Windows-users, compiling the OpenSRD program is more complicated. Whereas Linux and MacOS both implement POSIX standards and contain a default C build environment, this is not the case for Windows. Windows SDK Command Prompt window or Visual Studio implement compilers for Windows. However, in the experience of the OpenSRD development team these prove incompatible with internal functions of the OpenSRD library, which was originally developed in a Linux environment. For this reason, we advise to compile using MinGW and Code::Blocks. Although it requires more effort compared to a stand-alone CMake installation, it is believed that this is the most easy, fail-prove installation. The OpenCV library, on which OpenSRD relies on, can in this way be included without any difficulties. Additionally, Code::Blocks can be

used to make any necessary adjustments to the source files without any difficulties.

Code::Blocks is a free, open-source, cross-platform IDE and is oriented towards C/C++/Fortran. We will use this in combination with the compiler MinGW, an Open Source programming tool that is used for the development of native MS-Windows applications. In the rest of this section, the installation procedure that needs to be followed will be explained in detail. Please make sure that every step is followed in detail because some parts in the installation procedure rely on previous steps.

1. Download the latest version of OpenCV library, available at `http://opencv.org/`. Although a pre-built for Windows is available, it is recommended not to use those binaries. Extract the downloaded file (.zip) to C:*opencv* , in which two folders 'build' and 'sources' will be located.

2. Download Code::Blocks, available at `http://www.codeblocks.org/downloads/binaries`, without the mingw compiler: 'codeblocks-16.01-setup.exe'. This will allow to install the MinGW version that corresponds to the system used (32 or 64 bit), which is not the case for the default MinGW that comes with Code::Blocks (32 bit and outdated).

3. Download MinGW. As a suggestion, the OpenSRD team advises to use the TDM-GCC compiler suite for Windows, available at `http://tdm-gcc.tdragon.net/download` as a bundle installer in both a 32-bit or 64-bit binaries version. Install it locally as C:*mingw*. Select the default 'SourceForge mirror' and 'TDM-GCC Recommended C/C++ install'. Additionally, make sure that MinGW is installed with openmp by selecting openmp under gcc in the dropdown list (see Figure 2.1), because the default installation of TDM-GCC is without openmp support (used for multi-threading).

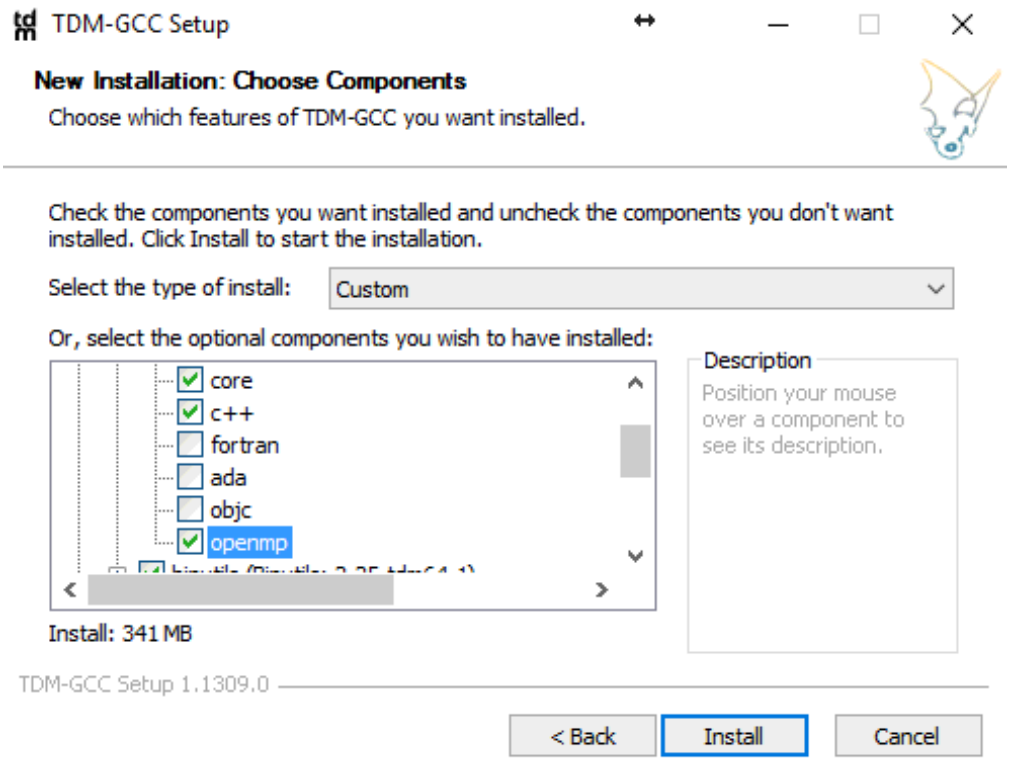

Figure 2.1: Add openmp to the components of TDM GCC that need to be installed.

4. Download CMake from `https://cmake.org/download/` for which the binary distribution can be used. Choose the applicable Windows Installer (32 or 64 bit) and when asked for, select 'Add CMake to the system PATH for all users'.

5. Set up the system path, for which we advise to use the software 'path editor' that is freely available on `http://patheditor2.codeplex.com/`. Add following paths to the local system path:
   - C:\$mingw$\$bin$
   - C:\$CMake$\$bin$
6. Create the Code::Blocks project to build OpenCV.
   a) Open CMake, using 'cmake-gui.exe' located in the bin folder of the CMake installation directory. Set 'source path' to C:\$opencv$\$sources$ and 'binary path' to C:\$opencv$\$Release$ (make a new folder within the opencv directory named 'Release').
   b) Hit the configure button, after which one should see a dialog box pop up. From the drop-down list, select 'codeblocks – MinGW Makefiles', make sure that 'Use default native compilers' is selected and click on 'Finish'.
   c) After a fist configuration run, click on the drop-down list next to WITH and select OPENMP. Click again on the configure button and finally click on 'Generate'.
   d) The created Code::Blocks-project file (opencv.cbp) will be located in the C:\$opencv$\$Release$ folder. Double click on it to open the project in Code::Blocks.
7. Build OpenCV.
   a) Adjust the Code::Blocks Settings by choosing 'Settings' in the Code::Blocks menu. Go to 'Compiler' and click 'Toolchain executables'. In the 'Compiler's installation directory' field choose the folder C:\$mingw$.
   b) Additionally, configure in the same tab the fields of 'Program Files' as shown in Figure 2.2.
   c) Selecting the target folder by choosing '$Build \rightarrow Select\ target \rightarrow install$' in the Code::Blocks menu. After clicking the button '$Build \rightarrow Build$', Code::Blocks will build all the binaries inside the 'install folder' C:\$opencv$\$Release$\$install$.
   d) When finished (this will take some time), add C:\$opencv$\$Release$\$install$\$x64$\$mingw$\$bin$ or C:\$opencv$\$Release$\$install$\$x84$\$mingw$\$bin$ to your system path using path editor.
8. Create the Code::Blocks project for surface reconstruction.
   a) Create a new empty project *OpenSRD* in Code::Blocks and specify the folder in which Code::Blocks will generate your project and the OpenSRD executables, e.g. create a folder C:\$Users$\$UserName$\$Documents$\$ProjectsCodeBlocks$\. Leave the other project settings to their default values.
   b) Click on the created project in the X window and select 'build options'. In the 'build options' window, go to 'search directories'. Then add following locations in the 'Compiler' tab:
      - C:\$opencv$\$Release$\$install$\$include$
      - C:\$opencv$\$Release$\$install$\$include$\$opencv$
      - C:\$opencv$\$Release$\$install$\$include$\$opencv2$

      Additionally, select the 'Linker' tab and add C:\$opencv$\$Release$\$install$\$x64$\$mingw$\$lib$ or C:\$opencv$\$Release$\$install$\$x84$\$mingw$\$lib$.
   c) Go in the 'build options' window to 'Linker Settings' and add the required libraries C:\$opencv$\$Release$\$install$\$x64$\$mingw$\$lib$\ $*$ $.dll.a$ as shown in Figure 2.3. In 'Other linker options', add -fopenmp.
   d) Go to 'Compiler settings' and select 'have C++11 ISO C++ language standard' ([-std=C++11]) in the tab 'Compiler Flags' under 'General'.
   e) In the tab 'Other compiler options', add -fopenmp.
9. Add the OpenSRD source and header files.

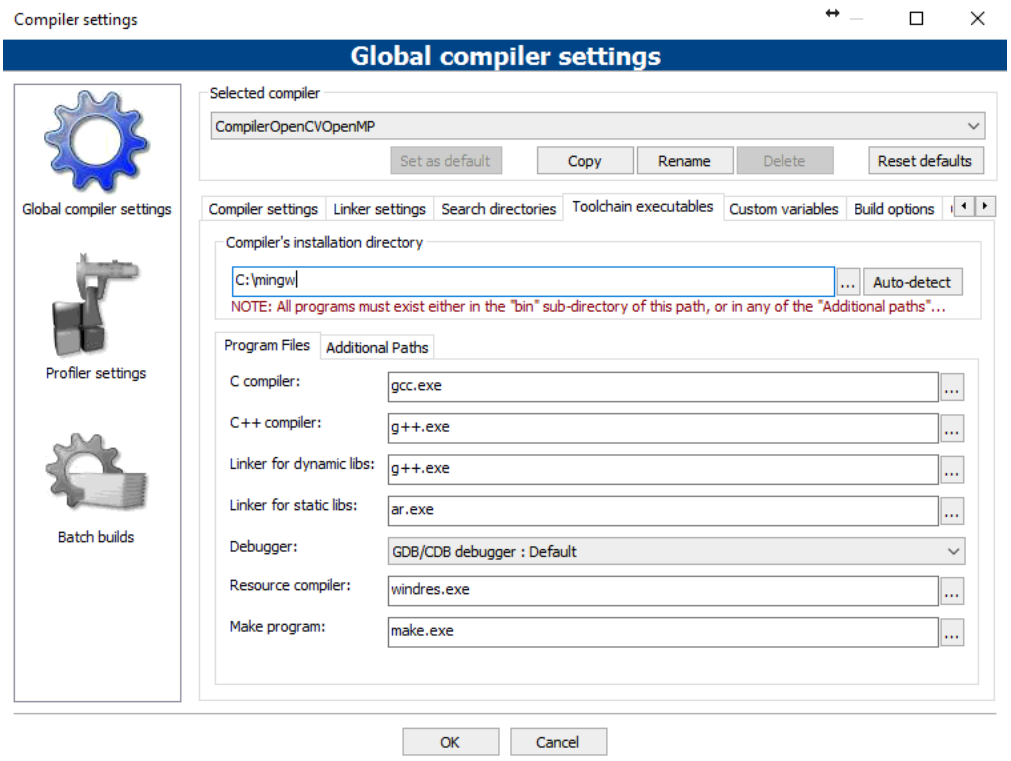

Figure 2.2: Fields that need to be adapted in step 7.

a) Download the OpenSRD library from Github: `https://github.com/lengelen/OpenSRD`.

b) Select the project and right-click on 'Add files...'. Select all OpenSRD header and sources files, located in the src folder of the downloaded OpenSRD library, except for *Calibration.cpp*.

c) Leave all settings on their default values.

10. Build the Windows-executable.

a) Choose the target executable by selecting '$Build \rightarrow Select\,target \rightarrow Release$'

b) Click on '$Build \rightarrow Build$'.

c) Once the process is terminated, you should find the Windows-executable (*OpenSRD.exe*) in the folder *OpenSRD\bin\Release* located in the directory in which you created the Code::Blocks project.

11. Follow the same steps 8-10 to create and build another Code::Blocks project for camera calibration. The only difference is that a new project needs to be created, named *Calibration*, in which only one source file *Calibration.cpp* needs to be added in step 9. The result is another executable, named *Calibration.exe* located in the *Calibration\bin\Release* folder.

12. Finally copy both executables to the directory which contains the OpenSRD library.

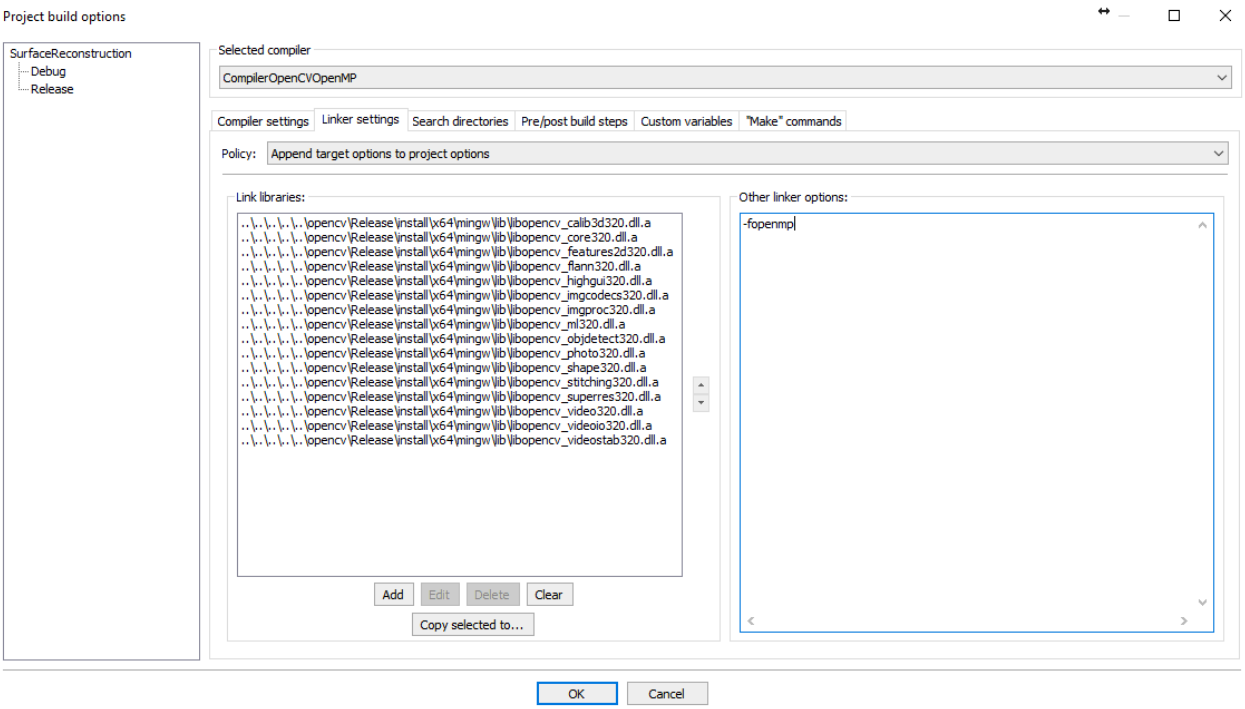

Figure 2.3: Tab 'Linker Settings' that needs to be configured in step 8 (although not really required, all available OpenCV libraries are installed).

Chapter 3

Using the compiled executables

This chapter explains how to use the OpenSRD executables, once build according to the instructions in Chapter 2. Alternatively, they can be downloaded directly from Github[1]. In Section 3.2, a detailed description of the camera calibration procedure, which needs to be done before processing of the measurement sequence, is provided. Section 3.3 describes how to use the program and adjust the settings to obtain the best results. A short note on how to post-process the output of the program is finally given in Section 3.4.

3.1 Required files and directories

Both the camera calibration and the surface reconstruction procedure require that some folders are present in the same directory as the executables. An example of each of these folders can be found in Github[1]

Folder INPUT

In this folder, some files must be present. A detailed description on how these files need to be adjusted will be given later in this chapter. An overview of the necessary files is given hereafter:

1. A xml-settings file to adjust the calibration settings.
2. A xml-settings file to adjust the surface reconstruction settings.
3. A text-file containing the physical coordinates of the reference pattern used for the extrinsic camera calibration, according to a chosen reference frame.
4. A text-file containing the physical coordinates of the feature pattern used for surface reconstruction, according to the chosen reference frame.
5. An image of the reference pattern without water present, used to estimate the extrinsic camera parameters.

   or

   A text file containing the extrinsic camera parameters from a previous camera calibration.

[1] `https://github.com/lengelen/OpenSRD`

Folder DATA

A folder which holds the images or text files containing the detected features (expressed in undistorted pixel coordinates) from a previous run (see 3. RESULTS/FEATURES_i). In case multiple cameras are used, the directory should contain sub-directories for each camera. To get a valid temporal description of the surface topography, choose the names of these images/files in a chronological order because the program reads them alphabetically. Due to the temporal smoothness of water, this also facilitates convergence because the result from a previous step provides a good initial estimate for the iterative optimization scheme.

Folder RESULTS

A folder in which the results of the surface reconstruction will be stored. Depending on the settings-file (see next section), this will contain (a subset of) the following files at the end of the reconstruction procedure:

- Folders *FEATURES_i*
  Output directories containing the detected features (in undistorted pixel coordinates) in the images of camera i for every time step. These intermediate results can be used for a second run with the same image sequence, using e.g. a different surface model. Reading in those files is much faster than detecting features in the images so highly recommended in case multiple runs are performed based on the same images.
- File *CameraPoseCam_i.txt*
  File containing the results of the 'camera pose estimation', i.e. estimation of the extrinsic camera parameters: the rotation matrix and translation vector of camera i.
- File *Coefficients.txt*
  File containing the result of the surface reconstruction. Each line represents the coefficients of the surface model corresponding to the instantaneous surface shape for a single time step (see Section 3.4).
- File *Errors.txt*
  File containing the residual error at the end of the optimization (normal collinearity metric or disparity difference metric), averaged over all feature points, for every time step. In case the error is too large, it might be better to exclude the corresponding optimization result during post-processing.

3.2 Calibration of the camera's

A first step in the reconstruction of the water surface consists of correcting the images for lens distortion. This form of optical aberration requires a correction of the images to obtain the undistorted pixel coordinates of the detected feature points. The distortion coefficients according to Brown's distortion model, which takes both radial as tangential distortion into account, are determined using an OpenCV function (based on Zhang [9]) and are used to undistort the pixel coordinates of the detected feature points for the rest of the calculations. For a more detailed elaboration about camera distortion, we refer to Engelen [6].

In order to compute the world coordinates of features seen in the images by the cameras, the camera parameters are also required. The camera is considered as a pinhole camera, schematically represented in Figure 3.1, which is characterized by its intrinsic parameters: principal point (c_x, c_y) and the focal lengths f_x and f_y. These camera characteristics, dependent on camera settings (image size, camera focus, ...), are stored in 'the intrinsic matrix', which is used to transform the 2D image points to 3D points in a camera reference frame. Using OpenCV's functions, the distortion coefficients are found together with the intrinsic (or internal) camera parameters. This requires for each camera a set of calibration images, in which a checkerboard pattern of a predefined size is depicted, using the same camera settings (camera focus, image size, ...) as during the measurement campaign. To obtain the best results, some recommendations are made with respect to this calibration procedure:

1. Calibration pattern
   First of all, it is recommended that the width and height of the checkerboard pattern are not identical. Otherwise, robust detection of the pattern (and its orientation) in the image becomes more difficult for

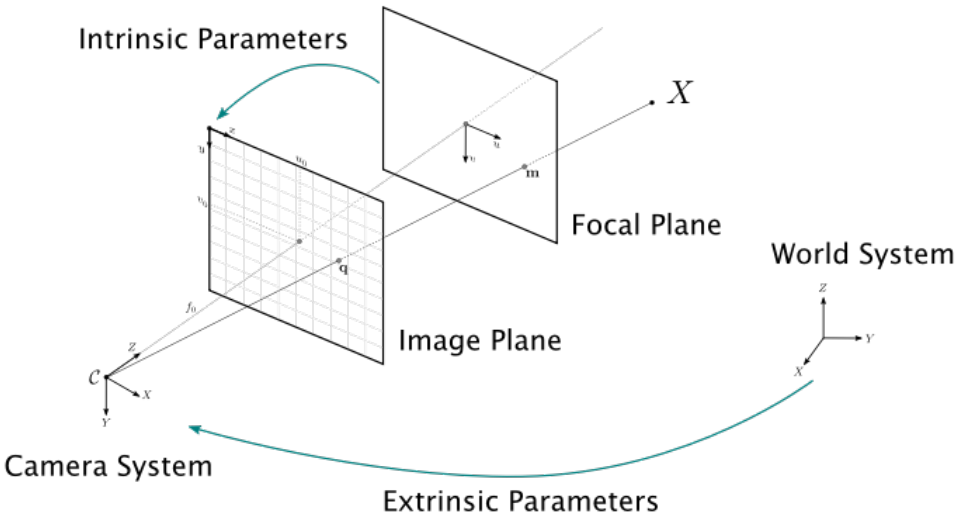

Figure 3.1: Schematic representation of pinhole camera and the transformation from pixel to world coordinates. From OpenMVG authors [8].

the OpenCV functions used during the calibration. Secondly, the number of rows and columns of the calibration pattern should not be too small to have sufficient amount of internal corners that serve as input. On the other hand, calibration patterns consisting of a huge amount of corners are difficult to detect using the pre-programmed OpenCV functions. As a suggestion, Engelen [6] uses a pattern of 11 X 17 internal checkerboard corners. Finally, the physical dimensions of the checkerboard-squares should be chosen as such that the pattern is largely enough depicted in the images to be easily distinguishable. This also depends on the distance between the camera and calibration pattern, for which we refer to recommendation 4 (Image quality).

2. Amount of images
   A sufficient amount of calibration images is required, in which the checkerboard pattern is viewed under different angles. In case too few images or images in which the camera position is not varied are used, a good estimate of the distortion coefficients and intrinsic parameters becomes impossible. However, too much images on the other hand just add 'noise' to the calibration and result in a lower calibration quality because corners are never detected perfectly. Based on scientific studies (e.g. Zhang [9]) and practical experience, a training set of approximately 20 images is advised.

3. Location of the checkerboard in the calibration images
   Enough corner points should be located close to the edges and corners of the image to get a better estimate of the distortion coefficients. Because lens distortion is most pronounced in the corners of an image, the correction of the pixel coordinates at the edges is more important than in the center of the image. A correct distortion model can however only be determined in case enough information about the distortion in those regions of the image is provided.

4. Image quality
   The quality of the calibration images should be as good as possible. The calibration images are taken with the camera parameters (focal length, aperture size, ...) that will be used during capturing of the reconstructed image sequence. In order to facilitate feature detection, the camera is focused to capture the feature pattern as sharp as possible. Scenes that are located at a significantly different distance to the camera will as such appear blurry for those specific camera settings. For this reason, the calibration images should be taken with a distance camera-calibration pattern approximately equal to the distance camera-water surface during the measurement campaign. This allows to obtain sharp calibration images in which the calibration pattern can unambiguously be located.

The executable that is used to to perform the calibration requires a XML 'settings file', of which an example is given by *settings_calib.xml*. The input variables that can be adjusted in this settings file are the following:

- AdaptImage
  Flag to indicate if calibration images need to be pre-processed to facilitate corner detection (perform brightness and contrast adjustments[2]): 1 → yes, 0 → no. In case 'yes', the α and β values are asked as input from the user.
- BoardSize_Width, BoardSize_Height
  Number of rows and columns of the calibration pattern, i.e. number of internal corners of the checkerboard pattern in each direction.
- InputDirectory
  Input directory containing the calibration images: give its full path or place the image folder in the same directory as the executable.
- ShowCalibration
  Flag to indicate if calibration images need to be shown to the user: 1 → yes, 0 → no.
- Show_UndistortedImage
  Flag to indicate if the undistorted calibration images need to be shown to the user: 1 → yes, 0 → no.
- Square_Size
  Dimensions of checkerboard squares (mm).
- Write_OutputFileName
  Name of the output file, e.g. *Calib_CAM1.yml*.
- WritePerView
  Flag to indicate if the average reprojection error for each calibration image[3] needs to be included in the output file: 1 → yes, 0 → no.

To perform the calibration of a single camera, run the executable *Calibration* in the command line by the command:

```
$ ./Calibration
```

A line will be printed to ask for the settings-file that will be used during calibration. Type the name of the correct xml-file, that needs to be present in the folder INPUT, in the command line and press enter. The result of the camera calibration is a YAML file, saved under the name that is given in the settings-file. This will contain the distortion coefficients, the intrinsic camera parameters (camera matrix) and, if requested, the reprojection error for each image.

3.3 Surface reconstruction

Extrinsic camera calibration

Expressing the position of points (X,Y,Z) in a world coordinate system, independent of the camera reference system, also requires the extrinsic camera parameters to be known. These allow a coordinate transformation between the camera's reference system and the world reference system. As such, the coordinate systems of the different camera's can be related to a single reference frame in which the position of the water surface can be described. Therefore, the current implementation requires a regular feature pattern (e.g. a checkerboard pattern on the bottom of the tank) for which the exact feature positions are known. The image of this pattern, as viewed by the camera, can then be compared to the 3D coordinates w.r.t. a chosen reference

[2]Each output pixel's value depends on the corresponding input pixel value and multiplication parameter α and addition parameter β: $g(x) = \alpha f(x) + \beta$ with f and g the original and modified intensity value respectively.

[3] The reprojection error is used as a qualitative measure of accuracy and is computed as the the distance between the image point detected in the calibration image and the projection of a world point in the image plane based on the calibration result. In case the overal mean reprojection error is large (as a finger-rule 0.5 pixels), it is better to recalibrate.

frame to estimate the camera position. These 3D coordinates need to be given by a text file, located in the folder INPUT. An example on the layout of such file can be found on Github[1] .

The relationship between point coordinates in the world coordinate system and the camera's coordinate system can be expressed with a matrix formulation, using a rotation matrix and translation vector. The estimation of these camera parameters, i.e. the 'camera pose estimation', is run automatically in the beginning of the surface reconstruction procedure for each camera used, based on the reference image present in the folder INPUT. In case multiple experiments are performed using the same camera system and world reference system, the extrinsic camera parameters can also be read from a text-file (saved output of a previous run). Moreover, it is possible to associate a different pattern to each of the camera's. This allows to use non-overlapping views of the different cameras, which is possible due to the single-view reconstruction methodology, in order to cover a larger surface area.

Temporal surface reconstruction

To start the global, temporal surface reconstruction procedure, run the executable *SurfaceReconstruction* through the command line by:

```
$ ./SurfaceReconstruction
```

This will initiate the executable, after which the name of the settings file will be asked. Specify the name of the correct XML file, which needs to be present in the folder INPUT, and press enter. An overview of the different variables that can be changed through the settings file is given hereafter. The main result of the temporal surface reconstruction program is a text file containing the optimized set of surface coefficients (see Section 3.4) that will be generated in the folder RESULTS.

General settings

- NumberOfCameras[4].
  Number of camera's used.
- TypeCameraPose
  Flag to indicate how the camera pose estimation will be performed: 1 → based on a reference image with reference pattern, 0 → using a file containing previously found matrix R and vector T.
- TypeFeatureInput
  Flag to indicate how feature position will be determined: 1 → detection of features in images using corner detection algorithm, 0 → using file containing previously detected feature locations.
- ThreadAmount
  Amount of threads used in multicore or multi-CPU systems (must be smaller the amount of threads possible): > 1 accelerates the reconstruction. Please keep in mind that, at the moment, the total amount of time frames must be a multiple of the amount of threads (otherwise some time frames will be skipped).

Settings concerning output of program

- SaveCameraPose
  Flag to indicate if the result of the camera pose estimation (matrix R and vector T) need to be saved: 1 → yes, 0 → no. Only possible in case camera pose estimation is performed using the specified reference image with reference grid.

[1] `https://github.com/lengelen/OpenSRD`

[4] In case < 3, the specified input/output files and directories of the additional cameras will be ignored (camera 3 or camera 2 & 3)

- SaveFeatureCoordinates
  Flag to indicate if the undistorted feature locations (in pixel coordinates) need to be saved to text files: $1 \rightarrow$ yes, $0 \rightarrow$ no.

- ShowCorners
  Flag to indicate if for every time step the detected features (corners) need to be shown in the corresponding image: $1 \rightarrow$ yes, $0 \rightarrow$ no. This can be useful to verify feature localization.

- SaveResiduals
  Flag to indicate if the residual error (averaged error metric over all features) corresponding to the optimized surface coefficients needs to be saved in a text-file: $1 \rightarrow$ yes, $0 \rightarrow$ no.

Input for surface reconstruction

- InputDirectory1
  Local path of the input directory (relative to directory of executable), containing images or text files of camera 1.

- InputDirectory2
  Local path of the input directory (relative to directory of executable), containing images or text files of camera 2.

- InputDirectory3
  Local path of the input directory (relative to directory of executable), containing images or text files of camera 3.

- InputFeatures1
  File name of text file containing fixed location of feature points $\mathbf{f}$ on the bottom plane F, viewed during reconstruction by camera 1 (needs to be present in Folder INPUT).

- InputFeatures2
  File name of text file containing fixed location of feature points $\mathbf{f}$ on the bottom plane F, viewed during reconstruction by camera 2 (needs to be present in Folder INPUT).

- InputFeatures3
  File name of text file containing fixed location of feature points $\mathbf{f}$ on the bottom plane F, viewed during reconstruction by camera 3 (needs to be present in Folder INPUT).

Input for extrinsic camera calibration

- InputInitial1
  File name of text file containing location of the checkerboard vertices in the image *InputReference1* w.r.t. to the world coordinate system during camera pose estimation of camera 1 (needs to be present in Folder INPUT).
- InputInitial2
  File name of text file containing location of the checkerboard vertices in the image *InputReference2* w.r.t. to the world coordinate system during camera pose estimation of camera 2 (needs to be present in Folder INPUT).
- InputInitial3
  File name of text file containing location of the checkerboard vertices in the image *InputReference3* w.r.t. to the world coordinate system during camera pose estimation of camera 3 (needs to be present in Folder INPUT).
- InputReference1
  File name of the reference image/file used for estimation of extrinsic camera parameters of camera 1 (needs to be present in Folder INPUT).
- InputReference2
  File name of the reference image/file used for estimation of extrinsic camera parameters of camera 2 (needs to be present in Folder INPUT).
- InputReference3
  File name of the reference image/file used for estimation of extrinsic camera parameters of camera 3 (needs to be present in Folder INPUT).

Input from intrinsic camera calibration

- CalibrationFile1
  File name of the YAML file containing the intrinsic camera parameters of camera 1 (see section 3.2).
- CalibrationFile2
  File name of the YAML file containing the intrinsic camera parameters of camera 2 (see section 3.2).
- CalibrationFile3
  File name of the YAML file containing the intrinsic camera parameters of camera 3 (see section 3.2).

Output of surface reconstruction

- OutputCameraPose1
  File name of the output file containing the saved camera pose estimation of camera 1 if flag *saveCameraPose* is set on true (will be generated in folder RESULTS).
- OutputCameraPose2
  File name of the output file containing the saved camera pose estimation of camera 2 if flag *saveCameraPose* is set on true (will be generated in folder RESULTS).
- OutputCameraPose3
  File name of the output file containing the saved camera pose estimation of camera 3 if flag *saveCameraPose* is set on true (will be generated in folder RESULTS).
- OutputDirectory1
  The name of output folder for saving feature coordinates of camera 1 in case flag *saveFeatureCoordinates* is set on true (will be generated in folder RESULTS).

- OutputDirectory2
  File name of output folder for saving feature coordinates of camera 2 in case flag *saveFeatureCoordinates* is set on true (will be generated in folder RESULTS).

- OutputDirectory3
  File name of output folder for saving feature coordinates of camera 3 in case flag *saveFeatureCoordinates* is set on true (will be generated in folder RESULTS).

- OutputFileName
  File name of output file containing the optimized set of surface coefficients for every time step, according to the chosen surface model (will be generated in folder RESULTS).

- OutputFileNameResiudals
  File name of output file containing the residual error for every time step, according to the chosen error metric and averaged over all feature points (will be generated in folder RESULTS).

Settings concerning dimensions of experimental set-up

- FeaturePatternSize_Width
  Number of rows of the feature pattern used for surface reconstruction.

- FeaturePatternSize_Height
  Number of columns of the feature pattern used for surface reconstruction.

- Lx
  Length scale in lateral x-direction (mm), see Appendix A.

- Ly
  Length scale in longitudinal y-direction (mm), see Appendix A.

- RefPatternSize_Width
  Number of rows of the reference pattern (internal checkerboard corners) used for camera pose estimation.

- RefPatternSize_Height
  Number of columns of the reference pattern (internal checkerboard corners) used for camera pose estimation.

Settings concerning feature detection & optimization

- DiffStep
  Numerical differentiation step[5] for gradient-calculation during optimization.

- Epsf
  Function-value based stopping condition[5] for optimization.

- Epsg
  Gradient based stopping condition[5] for optimization.

- Epsx
  Step-size based stopping condition[5] for optimization.

- ErrorMetric
  Error metric used: 1 $\rightarrow$ Normal collinearity metric, 2 $\rightarrow$ Disparity difference metric.

- Initialguess
  Initial guess to start optimization procedure of coefficients for every thread (amount specified with *ThreadAmount*). The amount of parameters passed in the initial guess does not has to match with the surface model that is chosen for the rest of the optimisation. This allows to pass a basic estimate, e.g. the average water level during the measurement campaign, which is then improved by the algorithm.

In case at least 4 surfaces are already reconstructed for a specific thread, a best estimate is chosen from the four previous reconstructions as well as this initial guess. This best estimate, i.e. the set of surface coefficients giving the smallest global error for the updated set of image points, can have a minor influence on the final result but more importantly influences the convergence rate of the optimization (with a very bad estimate convergence may never be reached).

- MatchesThreshold
  The maximum distance between the detected feature point (based on the corner strength measure) and predicted feature point (based on prediction from its location in the previous frame), expressed in pixels. (see Appendix B).

- Maxits
  Maximum number of iterations[5] during optimization.

- MinDistance
  The minimum distance in pixels between feature points, see Appendix B.

- Scaling
  Scale[1] of the coefficients during optimization (vector with length=SurfaceModelParameters). In case it is not expected that one term has no extremely large coefficients compared to the others, it is recommended not to use a variable scale and choose scale factor 1 for every term.

- SurfaceModelParameters
  Amount of parameters used in the surface model. Currently models with a 1, 3, 5, 8 or 12 terms are possible (see Section 3.4).

- ResponseRadius
  The radius to calculate the corner response: 5 or 10 pixels are currently possible (see Appendix B).

- ResponseThreshold
  The threshold for response strength of the detected, candidate image feature points (see Appendix B).

An example[6] of a settings file is given below:

```
<?xml version="1.0"?>
<opencv_storage>
<Settings>

  <!-- Number of camera's-->
  <NumberOfCameras>1</NumberOfCameras>
  <!-- Type of reference input for camera pose estimation: image->true:1, file->false:0
    -->
  <TypeCameraPose>1</TypeCameraPose>
  <!-- Type of input for feature positions: images->true:1, file->false:0-->
  <TypeFeatureInput>1</TypeFeatureInput>
  <!-- Amount of threads used-->
  <ThreadAmount>2</ThreadAmount>

  <!-- Save camera pose estimation in text file-->
  <SaveCameraPose>1</SaveCameraPose>
  <!-- Save feature coordinates in text file-->
  <SaveFeatureCoordinates>1</SaveFeatureCoordinates>
  <!-- Show image with found corners-->
  <ShowCorners>0</ShowCorners>
  <!-- Save average residual error in text file-->
  <SaveResiduals>1</SaveResiduals>

  <!-- Name of the input directory of camera 1-->
```

[5]For more information, we refer to the manual of the open source library ALGLIB of which the Levenberg-Marquardt algorithm is implemented.

[6]This example is also available on Github: `https://github.com/lengelen/OpenSRD`

```
<InputDirectory1>"DATA/CAM_1"</InputDirectory1>
<!-- Name of the input directory of camera 2-->
<InputDirectory2>"DATA/CAM_2"</InputDirectory2>
<!-- Name of the input directory of camera 3-->
<InputDirectory3>"DATA/CAM_3"</InputDirectory3>
<!-- Fixed location of points f for camera 1-->
<InputFeatures1>"FeaturesF_1.txt"</InputFeatures1>
<!-- Fixed location of points f for camera 2-->
<InputFeatures2>"FeaturesF_2.txt"</InputFeatures2>
<!-- Fixed location of points f for camera 3-->
<InputFeatures3>"FeaturesF_3.txt"</InputFeatures3>
<!-- Location of checkerboard vertices w.r.t. to world coordinate system during camera
     pose estimation of camera 1-->
<InputInitial1>"VerticesPoseEstimation_1.txt"</InputInitial1>
<!-- Location of checkerboard vertices w.r.t. to world coordinate system during camera
     pose estimation of camera 2-->
<InputInitial2>"VerticesPoseEstimation_2.txt"</InputInitial2>
<!-- Location of checkerboard vertices w.r.t. to world coordinate system during camera
     pose estimation of camera 3-->
<InputInitial3>"VerticesPoseEstimation_3.txt"</InputInitial3>
<!-- Name of the reference image/file - camera 1-->
<InputReference1>"REF1.png"</InputReference1>
<!-- Name of the reference image - camera 2-->
<InputReference2>"REF2.png"</InputReference2>
<!-- Name of the reference image - camera 3-->
<InputReference3>"REF3.png"</InputReference3>

<!-- Name of the calibration file of the camera 1-->
<CalibrationFile1>"Calib_C1.yml"</CalibrationFile1>
<!-- Name of the calibration file of the camera 2-->
<CalibrationFile2>"Calib_C2.yml"</CalibrationFile2>
<!-- Name of the calibration file of the camera 3-->
<CalibrationFile3>"Calib_C3.yml"</CalibrationFile3>

<!-- Name of ouput file for saving camera pose estimation camera 1-->
<OutputCameraPose1>"Camera_Pose_1.txt"</OutputCameraPose1>
<!-- Name of ouput file for saving camera pose estimation camera 2-->
<OutputCameraPose2>"Camera_Pose_2.txt"</OutputCameraPose2>
<!-- Name of ouput file for saving camera pose estimation camera 3-->
<OutputCameraPose3>"Camera_Pose_3.txt"</OutputCameraPose3>
<!-- Name of ouput directory for saving feature coordinates camera 1-->
<OutputDirectory1>"FEATURES_1"</OutputDirectory1>
<!-- Name of ouput directory for saving feature coordinates camera 2-->
<OutputDirectory2>"FEATURES_2"</OutputDirectory2>
<!-- Name of ouput directory for saving feature coordinates camera 3-->
<OutputDirectory3>"FEATURES_3"</OutputDirectory3>
<!-- Name of ouput file for optimized coefficients according to chosen surface model
    -->
<OutputFileName>"Coefficients.txt"</OutputFileName>
<!-- Name of ouput file for average residual error per frame-->
<OutputFileNameResiudals>"Errors.txt"</OutputFileNameResiudals>


<!-- Number of rows and columns of feature pattern -->
<FeaturePatternSize_Width>5</FeaturePatternSize_Width>
<FeaturePatternSize_Height>17</FeaturePatternSize_Height>
<!-- Length scale in x-direction (lateral direction)(mm)-->
<Lx>40</Lx>
<!-- Length scale in y-direction (streamwise direction)(mm)-->
<Ly>160</Ly>
<!-- Number of rows and columns of reference pattern -->
<RefPatternSize_Width>5</RefPatternSize_Width>
<RefPatternSize_Height>17</RefPatternSize_Height>


<!-- Numerical differentation step for calculation of gradient-->
<DiffStep>0.01</DiffStep>
<!-- Function-value based stopping condition-->
```

```
    <Epsf>0</Epsf>
    <!-- Gradient based stopping condition-->
    <Epsg>0.000001</Epsg>
    <!-- Step size-based stopping condition-->
    <Epsx>0</Epsx>
    <!-- Error metric used: 1->normal collinearity metric, 2->disparity difference metric
        -->
    <ErrorMetric>2</ErrorMetric>
    <!-- Initial guess to start optimization in each thread-->
    <InitialGuess>"[70.0,0.000,0.000,0.000,0.000]"</InitialGuess>
    <!-- Maximum distance between located and predicted feature point -->
    <MatchesThreshold>10</MatchesThreshold>
    <!-- Maximum number of iterations during optimization-->
    <MaxIts>0</MaxIts>
    <!-- Minimum distance in pixels between corner points-->
    <MinDistance>15</MinDistance>
    <!-- Scaling of coefficients for optimization (length=SurfaceModelParameters)-->
    <Scaling>"[1.0,10.000,10.000,10.000,10.000]"</Scaling>
    <!-- Number of parameters used in the surface model-->
    <SurfaceModelParameters>5</SurfaceModelParameters>
    <!-- Radius to calculate corner response: 5 or 10 currently possible -->
    <ResponseRadius>5</ResponseRadius>
    <!-- Threshold parameter for response strength corners-->
    <ResponseThreshold>350</ResponseThreshold>


  </Settings>
  </opencv_storage>
```

3.4 Post-processing

The text file containing the optimized surface coefficients can finally be used to obtain a temporal description of the surface topography. Based on the Fourier theory regarding the decomposition of a periodic signal in a Fourier series, a surface model containing different cosine terms is currently implemented in the program. Especially for the study of standing gravity waves, Lamb [7] proved that the water surface can be described by following model:

$$\eta(x,y) = \sum_m \sum_n A_{mn}\, cos(\frac{m\pi x}{L_x})\, cos(\frac{n\pi y}{L_y}), \quad m,n = 0,1,2,\ldots$$

Because most often higher order terms in this infinite sum have a small amplitude and are negligible, different low-parameter surface models with only a limited amount of terms are employed to study the domain of interest (see Appendix A). A linear term in both the x- and y-direction, $B\frac{x}{L_x}$ and $C\frac{y}{L_y}$, is usually also included to cope with a feature plane F that is not perfectly parallel with a still and flat water surface in the tank. Table 3.1 gives an overview of the surface models currently implemented, in which the subscript of P_i indicates the amount of adopted parameters.

Table 3.1: Overview of possible surface models

Model	$\boldsymbol{\eta(x,y)}$
P_1	A_{00}
P_3	$A_{00} + B\frac{x}{L_x} + C\frac{y}{L_y}$
P_5	$A_{00} + B\frac{x}{L_x} + C\frac{y}{L_y} + A_{10}\,cos(\frac{\pi x}{L_x}) + A_{01}\,cos(\frac{\pi y}{L_y})$
P_8	$A_{00} + B\frac{x}{L_x} + C\frac{y}{L_y} + A_{10}\,cos(\frac{\pi x}{L_x}) + A_{01}\,cos(\frac{\pi y}{L_y})$ $+A_{11}\,cos(\frac{\pi x}{L_x})\,cos(\frac{\pi y}{L_y}) + A_{20}\,cos(\frac{2\pi x}{L_x}) + A_{02}\,cos(\frac{2\pi y}{L_y})$
P_{12}	$A_{00} + B\frac{x}{L_x} + C\frac{y}{L_y} + A_{10}\,cos(\frac{\pi x}{L_x}) + A_{01}\,cos(\frac{\pi y}{L_y})$ $+A_{11}\,cos(\frac{\pi x}{L_x})\,cos(\frac{\pi y}{L_y}) + A_{20}\,cos(\frac{2\pi x}{L_x}) + A_{02}\,cos(\frac{2\pi y}{L_y})$ $+A_{21}\,cos(\frac{2\pi x}{L_x})\,cos(\frac{\pi y}{L_y}) + A_{12}\,cos(\frac{\pi x}{L_x})\,cos(\frac{2\pi y}{L_y}) + A_{30}\,cos(\frac{3\pi x}{L_x}) + A_{03}\,cos(\frac{3\pi y}{L_y})$

To visualize the surface evolution corresponding to the optimized coefficients, insert the set of optimized coefficients in the chosen surface function. Finally, plot the function $\eta(x, y)$ in the domain (x,y) in which feature points were located, as the optimization result is only valid for those x- and y-values with software of choice.

Bibliography

[1] ALGLIB Project. ALGLIB: a cross-platform open source numerical analysis and data processing library, 1999. Currently available at: `http://www.alglib.net/`, Last revised: 2017.

[2] S. Bennett and J. Lasenby. Chess – quick and robust detection of chess-board features. *Computer Vision and Image Understanding*, 118:197 – 210, 2014. ISSN 1077-3142.

[3] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. Currently available at: `http://opencv.org/` , Last revised: 2016.

[4] G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc., 2008.

[5] J. Burkardt. MINPACK Least Squares Minimization , 2010. Currently available at: `http://people.sc.fsu.edu/~jburkardt/cpp_src/minpack/minpack.html`, Last revised: 2010.

[6] L. Engelen. Spatio-temporal Image-based Water Surface Reconstruction: Application to Sloshing In Navigation Lock Filling. Master's thesis, University of Ghent, Belgium, 2016.

[7] H. Lamb. *Hydrodynamics*. Cambridge University Press, 1932.

[8] OpenMVG authors. cameras-Pinhole camera model. Currently available at: `http://openmvg.readthedocs.io/en/latest/openMVG/cameras/cameras/`, Last revised: 2016.

[9] Z. Zhang. A flexible new technique for camera calibration. *IEEE Trans. Pattern Analysis and Machine Intelligence (TPAMI)*, 22(11):1330–1334, 2000.

Appendix A

Definition of chosen coordinate reference frame

In paper X, the surface reconstruction was done in a reference frame in which the x- and y-direction correspond with the lateral and longitudinal direction respectively. The z-axis is then directed upwards, resulting in positive water heights. Figure A.1 gives a graphical representation of the chosen reference frame, in combination with the adopted feature pattern.

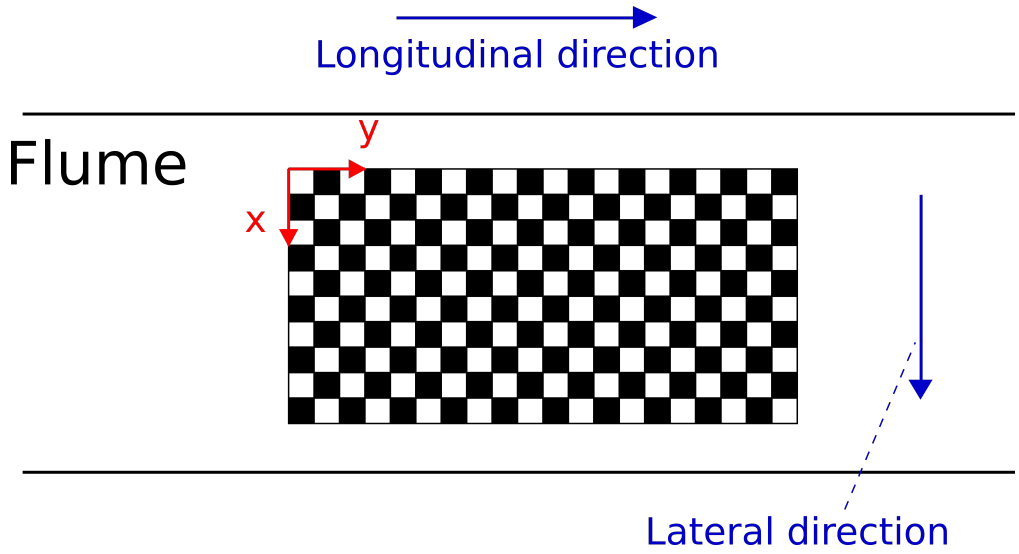

Figure A.1: Chosen reference frame in X.

Appendix B

Corner detection algorithm

Corner detection is based on the algorithm developed by Bennett and Lasenby [2]. The algorithm employs a corner strength measure which is calculated for every pixel in the image plane. Checkerboard corners are characterized by their 2×2 reciprocal black and white squares. By comparison of the pixel values on a discretized sampling circle around the pixel of interest, a corner strength measure can be defined which gives a maximum value for the pixel at the center of such a 2×2 pattern. Bennett and Lasenby [2] prove that a robust but still computational efficient corner strength measure can be obtained by comparison of the pixel intensities (for a gray-scale image between 0 and 255) on a discretized sampling circle of 16 points and a given radius. The corner strength measure is composed of two parts, namely the 'sum response' and the 'diff response'.

1. Sum response

On a vertex of a checkerboard pattern, pairs of points located on opposite sides of the sampling circle around the vertex should be of similar intensities. Points that are 90° out of phase on the circle should be of very different intensity. We denote I_n the image intensity of the n^{th} sampling point by proceeding on the discretized sampling circle (16 points), starting at an arbitrary point with intensity I_0. Around a vertex of the checkerboard pattern, the magnitude $(I_n + I_{n+8}) - (I_{n+4} + I_{n+12})$ should be very large (positive or negative) for every n. The 'sum response' (SR) is therefore computed as the summation of these image pairs over all points on the sampling circle (notice that we only have to loop from point 0 to 3 to compare all pixel pairs):

$$SR = \sum_{n=0}^{3} |(I_n + I_{n+8}) - (I_{n+4} + I_{n+12})|$$

SR is large at a checkerboard vertex point, but additional calculations are necessary to avoid false positives which might occur at edges or due to image noise. Edges, especially in combination with image blur and image distortion, can have a significant sum response as well and are therefore difficult to distinguish from vertices's of the checkerboard pattern.

2. Diff response

Points located on a simple edge are characterized by very different intensities on opposite sides of the sampling circle. Therefore, the 'diff response' (DR) should be large for edge points:

$$DR = \sum_{n=0}^{7} |I_n - I_{n+8}|$$

3. Total response

By subtracting the diff response from the sum response, the signal-to-noise ratio of the corner strength measure is significantly improved. This results in the 'total response' (TR):

$$TR = SR - DR$$

The algorithm retains the list of image points of which the corner strength measure (TR) is larger than the 'ThresholdResponse'. Points located at a distance smaller than the 'MinDistance' are combined to a single corner point by taking their averaged position, each weighted with their corresponding corner strength. The implemented methodology is rotationally invariant and proved to locate checkerboard corners accurately and robust, even in case of optical blur and image noise. The radius of the sampling circle should however be fitted to the size of the pattern within the image plane, for which a compromise between two conflicting criteria needs to be made. The radius of the sampling should as small as possible in order to avoid aliasing on the other grid squares and allow the use of more dense patterns. On the other hand, the radius should be large enough to escape the central region in case of blurriness in the image. At the moment, two different 'response radius's' are possible, namely with 5 or 10 pixels. Choose the one best-fitted for your experimental set-up. In case both are insufficient, one should adjust the current response functions (located in *ParticleTracking.cpp*).

Appendix C

License

OpenSRD is freely available and distributed under the GNU General Public License version 3 (GNU GPLv3 or GPLv3) terms. Detailed information about the terms and conditions for use and redistribution, with or without changes to the program source code, can be found on `http://www.gnu.org/licenses/`. Acknowledgement of the accompanying paper () is appreciated.

As mentioned in Chapter 2, OpenSRD makes use of three Open-Source libraries. We therefore like to add following notes to comply with the licenses of these libraries.

OpenCV

License Agreement
For Open Source Computer Vision Library
(3-clause BSD License)

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the names of the copyright holders nor the names of the contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided by the copyright holders and contributors "as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall copyright holders or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

ALGLIB

ALGLIB, the library used for multivariate optimization, is licensed under the GNU General Public License version 2. It can therefore be modified and redistributed under the terms and conditions that are described on `http://www.gnu.org/licenses/`.

MINPACK

The C++ version of MINPACK by Burkardt [5] is distributed under the GNU LGPL license, which is also compatible with GNU GPLv3. We remark that the some functions used from this library are modified to our needs, as mentioned in the corresponding source code.