Politecnico di Torino

Politecnico di Torino

Computer Engineering

A.Y. 2024/2025

Graduation Session October 2025

Particle Tracking Acceleration

Supervisors:
Riccardo Cantoro

Company Tutor:
Kamel Abdelouahab

Candidate:
Francesco Risso

Acknowledgements

TODO

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Particle tracking

Thanks to the existence of the Earth's atmosphere, we live in an enormous fish bowl, filled to the brim with air. In our everyday life, we constantly perceive this mass of air: for instance we breathe it, we are subjected to the atmospheric pressure, we see insects flying through it, as if they were swimming.

Sometimes we can perceive that this giant mass is moving, for example when there is a slight breeze blowing on us. This may lead curious people to desire to understand deeply how and why this movement happens.

Since air is transparent, we cannot base our research on observing it. The tactile feelings are not enough, either, since we are only able to perceive movements of a certain strength. We can however leverage our good vision to observe small particles floating in the air: they move with the surrounding, but unlike the air, we can see them.

Directly observing the particles can be a starting point to understand the air movement. However, this method lacks the possibility to "pause" the time to reflect, and the possibility to rollback the observation to check what has just happened. The most common solution to this problem is to record the observation with cameras, to be able to pause and re-watch the footage. However, a simple 2D recording cannot encode the essential information about depth: for these applications, it is therefore crucial to capture the experiment in a way that allows to reconstruct the trajectories in full 3D.

While being easy for the human brain, the task of recreating a 3D scene from some 2D views is not trivial for an algorithm. In my thesis, I had to try to accomplish it with good quality, with very stringent requirements about computation time.

1.2 Structure of the thesis

This thesis is structured as follows:

Chapter 2 explains the overall goals of the project and of my thesis;

Chapter 3 provides theoretical knowledge about camera calibration, the technique of stereoscopy and image moments, and illustrates the main programming languages, libraries and tools used in the thesis work;

Chapter 4 describes the hardware where the solutions were compared and tested, as well as the cameras providing the images;

Chapter 5 explains the steps in which the particle tracking pipeline is commonly split;

Chapter 6 deeply analyses the requirements, the alternatives and the final implementation for the *Locate* step of the algorithm;

Chapter 7 performs a similar analysis for the *2D Link* step;

Chapter 8 examines in a similar way the approaches for *3D Matching*;

Chapter 9 describes the 3D alternatives for the *Link* step, in contrast to the 2D approaches explained in chapter 7;

Chapter 10 illustrates the two visualizers that were developed to interactively see the reconstructed trajectories;

Chapter 11 explains how the steps described in the previous chapters are joined together in a single pipeline;

Chapter 12 evaluates the quality and speed of the final solution;

Chapter 13 draws the final conclusions from the thesis, and hints at possible future improvements.

Chapter 2

Goals

2.1 Project objectives

Not long after the COVID-19 pandemic started, people realized that the disease could spread by means of water droplets suspended in the air. This lead many research groups in the fluid dynamics domain to investigate how air would move inside a room, carrying such droplets (some examples: [1][2]).

The overall project that includes my thesis is a research topic on this momentum. The final goal is to understand the common patterns that air follows when moving — while apparently being still — in a room. To understand this, an experimental setup was created in a small room. In a corner of the chamber there was a machine [3] able to create bubbles, similar in concept to the soap bubbles that children use to play (figure 2.1). By observing the movement of these bubbles, the movement of the air would then be inferred.

Since the bubbles need to move in the same way of the surrounding air, a way to cancel out all the other forces is required. The surface and filling material for the bubbles therefore need to be carefully chosen, in order to have an overall weight density of the bubble similar to the air density. This allows the buoyancy force to compensate almost exactly the gravity force, leaving only the force of the surrounding air to move the bubble.

The experiments are conducted in two steps. First, some bubbles are created: when there are enough, the machine is stopped, to avoid air currents caused by the machine itself. Then, a small amount of time is waited, to allow the bubbles to lose their initial speed, and to settle in the room air movement. After this this short period, the observation starts. Due to this composed procedure, the "bubble material" would be required to create long-lasting bubbles, whose average lifetime is at least 5 minutes.

On top of that, soap bubbles are too big for the purpose: there is a high chance that a bubble in front occludes another bubble in the back, reducing the quality of the observation. For this reason, the "soap" must be a material that creates bubbles with a maximum diameter of some millimeters.

From all these considerations, the bubbles were created with a coating made of a proprietary substance created by Sage Action [4], filled with helium.

Figure 2.1: The experimental setup: the machine in the corner of the room is creating some small bubbles that fill the space

2.2 Company objectives

2.2.1 The cameras

SMA-RTY France [5], the company where I did my internship, has as core business the selling of special-purpose, FPGA-driven cameras. As such, one of the two tasks contracted to them by the research group was the construction of a specific camera for this purpose: this task was tackled by their internal team of embedded developers. In the final setup, 3 or 4 cameras were used in a stereoscopic arrangement, as shown in figure 2.2.

Figure 2.2: A tripod with 3 SMA-RTY cameras in a stereoscopic arrangement

2.2.2 Reconstruction algorithm acceleration

The research group internally developed a MATLAB tool for analyzing the video footage from an arrangement of 3 cameras, but the processing speed was extremely slow. As such, they contracted SMA-RTY to create an accelerated version, either by improving the original one, or by creating a totally new script, in whichever programming language was best. The objective of the acceleration was to have a real-time software, that would be able to process the videos live, with an allowance of some seconds of jitter. That is, a delay between capturing the frame and outputting the reconstruction was acceptable, as long as it did not increase over time.

SMA-RTY internally started working on this, with a different solution that accelerated the processing to 19 FPS. This script was already orders of magnitude faster than the original one, but the obtained speed was still less than the target. On top of this, this proposed solution was working with ToF (Time of Flight) cameras to perform the depth estimation, thus avoiding the expensive 3D reconstruction methods. However, the project required visible cameras, therefore this idea had to be discarded.

2.3 Thesis objectives

I was assigned by the SMA-RTY team the task of accelerating the MATLAB script for processing the videos. Originally, the vision was to exploit my CUDA skills to leverage the parallelization of GPUs. However, as the body of this thesis will make clear, there was not so much parallelization that could be done. Instead of focusing on "better"

hardware architectures, the best course of action was indeed to optimize the various software steps.

Therefore, the objective of my thesis became the recreation of the full pipeline, from image capturing to 3d markers rendering. The main constraint of the result would be the speed, 24 FPS were required at steady-state, while the output quality should be as good as possible.

Chapter 3

Background knowledge

3.1 Camera calibration

Camera calibration is the process of estimating the parameters of a vision system. Three types of calibration parameters exist: distortion, intrinsics and extrinsics.

3.1.1 Distortion [6]

What is distortion

A camera lens can introduce two types of distortion: radial and tangential. Radial distortion makes straight lines appear curved in the image, while tangential distortion can make some image parts look closer than they are. Figure 3.1 provides a graphical example of both distortion problems.

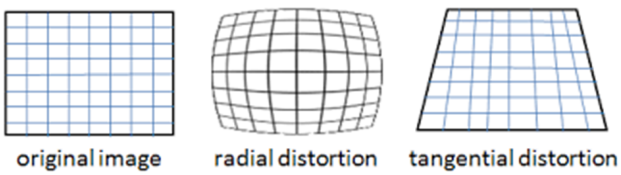


Figure 3.1: How radial and tangential distortion affect an image

Mathematical definition

Given an undistorted point $P = \begin{bmatrix} x \\ y \end{bmatrix}$ (at distance r from the center of the image), a radial distortion will transform it into

$$P_{dist} = \left(1 + k_1 \cdot r^2 + k_2 \cdot r^4 + k_3 \cdot r^6\right) \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

while a tangential distortion would transform it into

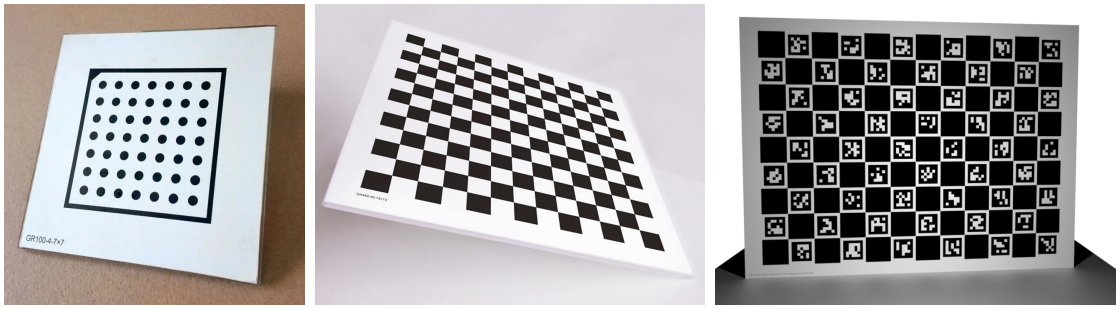
$$P_{dist} = \begin{bmatrix} x + \left[2p_1xy + p_2 \cdot \left(r^2 + 2x^2\right)\right] \\ y + \left[2p_2xy + p_1 \cdot \left(r^2 + 2y^2\right)\right] \end{bmatrix}$$

As such, the full distortion can be described with the vector $\begin{bmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{bmatrix}$.

How to calibrate the parameters

The estimation of the distortion parameters (among with other parameters) can be computed using `OpenCV`'s `calibrateCamera` function. It requires data extracted from multiple calibration frames, each one with a set of coplanar points. Each frame must provide the list of pixel coordinates of the points detected in the image. On top of that, each point must be labeled with a coordinate system local to the plane where the points lay: an example can be a row/column index for a grid-like pattern.

Common calibration patterns are dots (figure 3.2.a) or the corners of a chessboard (figure 3.2.b). Often, to help the detection of the chessboard corners, ArUco [7] markers are added in the white cells (figure 3.2.c). This combination of **ch**essboard and **ArUco** markers is called **ChArUco**.



(a) (b) (c)

Figure 3.2: Calibration planes: (a) dots, (b) chessboard, (c) ChArUco

How to remove distortion

Using `OpenCV`'s `undistort` function, it is possible to remove the lens distortion. The result is the image as it would be taken by an ideal camera setup.

3.1.2 Intrinsic parameters [8]

What are intrinsic parameters

The intrinsic parameters describe how the lens and sensor alter the image captured by a single camera. Using these parameters, it is possible to remove all this distortion, transforming the image into a common frame of reference. These transforms allow to obtain the same image when two different cameras, with different optics, photograph the same scene.

Mathematical definition

Consider a simple scene, with a camera observing a point. Define a frame of reference centered in the camera. The point can be described as $R = \begin{bmatrix} P_x & P_y & P_z \end{bmatrix}^T$. The camera will project the point onto the image plane, at the homogeneous coordinate $R' = \begin{bmatrix} P'_x & P'_y & 1 \end{bmatrix}^T$. It is possible to show that P' can be written as P transformed by a matrix K, called **intrinsic matrix**: $P' = K \cdot P$.

In particular, K is in the form:

$$K = \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

where:

- f_x and f_y are the focal lengths in pixels of the optic system. They may differ along the horizontal and vertical direction;
- s is the skew, that can be caused by the digitization process;
- x_0 and y_0 are the coordinates of the pixel where the center of projection of the camera stands.

How to calibrate the parameters

The full intrinsic matrix is estimated by the same `OpenCV` function that evaluates the distortion coefficients.

How to remove intrinsic parameters

Using `OpenCV`'s `undistort` function, it is possible to remove also the intrinsic parameters. The result is the image as it would be taken by a camera with $K{=}I_3$. This enables to compare pixel-wise images captured with different optic and sensor arrangements.

3.1.3 Extrinsic parameters

What are extrinsic parameters

The extrinsic parameters describe position and orientation of a camera with respect to a specific frame of reference. Usually they are not computed when a single camera is present, since it is convenient to assume as frame of reference the position and orientation of the camera. With multiple cameras, usually one of them is chosen as "main", and it acts as the frame of reference for the other cameras. In this frame of reference it is therefore crucial to understand position and orientation of all the cameras.

Considering camera A as main, the extrinsic parameters of another camera B can be expressed in three different ways:

- **rotation matrix** R and **translation vector** t: R describes how to rotate A to be in the same orientation as B (equivalently, how B is rotated in A's frame of reference); t is the versor in B's frame of reference towards the origin (equivalently, B is located in $-Rt$ in the main frame of reference);
- **essential matrix** E. Assuming $P_A = \begin{bmatrix} x_A & y_A & 1 \end{bmatrix}$ and $P_B = \begin{bmatrix} x_B & y_B & 1 \end{bmatrix}$ are the *undistorted* projections of a point P on the two cameras, E is a matrix such that $P_B \cdot E \cdot P_A^T = 0$. E can also be computed as $E = [t]_\times R$, where $[t]_\times$ is the matrix representation of the cross product of t;
- **fundamental matrix** F: the definition is the same as E's, but using the points with K still applied (only the distortion has to be removed). If the two cameras have intrinsic matrices K_A and K_B, F can be computed as $F = \left(K_A^{-1}\right)^T \cdot E \cdot K_B^T$.

How to calibrate the parameters

The extrinsic calibration can be obtained from the same data as the intrinsic calibration, provided that the cameras took a picture of the exact same scene (which likely implies that the pictures must be taken at the exact same time instant). The calibration points detected on both images can be processed by:

- `OpenCV`'s `recoverPose` function to obtain R and t;
- `OpenCV`'s `findEssentialMat` function to obtain E;

- `OpenCV`'s `findFundamentalMat` function to obtain F.

How to remove extrinsic parameters

Non-null extrinsic parameters mean that the two images are fundamentally different, therefore it is not possible to remove these parameters. Instead, these parameters are essential for reconstructing the 3D scene using stereoscopy.

Definition up to a scale factor

If everything (objects in the picture, distances between objects and pixel size and focal length of the cameras) is scaled by a factor k, then the resulting images do not change. For this reason, the extrinsic parameters can only be defined up to a scale factor. Most importantly, this affects t: it cannot be the vector of the displacement, since the distance is unknown, but it is only the versor of the direction of the displacement.

This makes the reconstruction (stereoscopy) use an arbitrary unit of measurement, which corresponds to the distance between the cameras. This must be taken into account particularly if there are more than 2 cameras: each camera will have a different distance from the main one, thus having different units of measurement. The problem can be solved by computing the scaling factor between the units such that the reconstructed calibration points have coherent distance between the different camera couples. To have a realistic measurement unit, it is also possible to impose this scaling factor in a way that the distance between the reconstructed points is coherent with the real one.

3.2 Stereoscopy

Our brain is able to understand depth by leveraging the fact that we have two eyes in slightly different positions. This mechanism is called stereoscopy, and it can be used by a computer vision system, provided that it has two or more cameras.

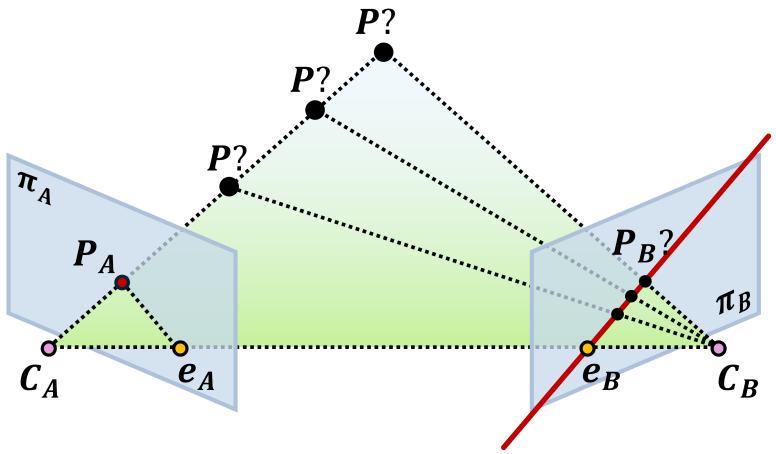


Figure 3.3: Reprojecting a point from 2D to 3D: it could be anywhere on a specific line

3.2.1 Depth estimation

Consider a camera A, with center of projection C_A and image plane π_A (left in figure 3.3). If a 3D point P is seen by the camera, in the image plane it will be $P_A = \pi_A \cap \overline{PC_A}$. From a single camera, it is impossible to reconstruct P from P_A: there would be infinitely many possible Ps, all the points that lie on the extension of $\overline{P_A C_A}$.

If another camera B is available (right in figure 3.3), and the same point P is projected as P_B, then a new information is added: that P will lie on the extension of $\overline{P_B C_B}$. By intersecting these lines, it is ideally possible to reconstruct the original 3D position of P.

In order to do so, the relative position of the cameras needs to be known: all calibration parameters are required. In particular, the function `triangulatePoints` from `OpenCV` is able to reconstruct the 3D positions given the 2D matched observations, the intrinsic and distortion parameters of the two cameras, and the extrinsic matrix of the couple.

3.2.2 Epilines

As stated before, the point P_A could correspond to a full line of 3D points. When seen by camera B (with a different point of view), this line translates to a 2D line in π_B (red in figure 3.3). This line is called **epiline**.

Different points P_A will correspond to different epilines, which however all pass by the **epipoint** e_B. The epipoint is defined as $e_B = \pi_B \cap \overline{C_A C_B}$.

Computing the epiline equation

As explained in the previous section, the essential matrix E is such that $P_B \cdot E \cdot P_A^T = 0$, which is called the **epipolar constraint**. If P_B is a generic point on the image, it can be described as $P_B = \begin{bmatrix} x & y & 1 \end{bmatrix}$. The result of $E \cdot P_A^T$ is a 3×1 vector, that can be written without loss of generality as $\begin{bmatrix} a & b & c \end{bmatrix}^T$. When all this knowledge is substituted into the epipolar constraint, we obtain the following:

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \cdot E \cdot P_A^T = 0$$

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = 0$$

$$ax + by + c = 0$$

which is the equation of the epiline in B's frame.

3.2.3 3D matching

For estimating the depth, `triangulatePoints` needs matched point. That is, the $i{-}th$ point provided by camera A and the $i{-}th$ point provided by camera B must be the two projections of the same 3D point. To perform this matching, traditional stereoscopy follows this procedure:

1. Choose a point in the main image;
2. Compute the equation of the corresponding epiline;
3. Consider a patch around the original point;
4. For each point on the epiline (in the second image), compute the similarity of a patch centered in that point with the original patch;
5. Select the most similar point as the match;
6. Compute the 3D coordinate of the point from the obtained match.

3.3 Image moments

In mathematics, a **moment** is a quantitative measurement related to a function's graph. In particular, the moment of index n of the function $f(x)$ is defined as $\int x^n f(x)dx$. The same underlying concept can be adapted to images, with some modifications to adapt to the two, discrete dimensions.

Given a grayscale image of size $N \times M$, name $I(x, y)$ the intensity of its pixels. Then, the **raw moment of index** $(p+q)$ can be computed as:

$$M_{pq} = \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} x^p \cdot y^q \cdot I(x, y)$$

These moments are able to describe some features of the image. In particular, for our purpose we consider the moments of index 0 and 1:

- M_{00} is the sum of the gray level of the image. For binary images, it coincides with the area;
- M_{10} is a measure of how far to the right the illuminated pixels are positioned. If the total area is taken into account, the x coordinate of the centroid of the image is available: $\overline{x} = {}^{M_{10}}/_{M_{00}}$;
- Similarly, $\overline{y} = {}^{M_{01}}/_{M_{00}}$ is the y coordinate of the centroid of the image.

3.4 Programming languages and libraries

The particle tracking software developed in the scope of this thesis is fully written in Python. This choice was made considering many reasons, including:

- the extensive quantity of optimized libraries for accomplishing many sub tasks (e.g. `NumPy`);
- the simplicity of the syntax, leading to fast development and testing;
- the presence of many existing approaches to the problem in this language;
- the possibility to run GPU kernels.

Initially, there were discussions about testing the different ideas in Python for development speed, to then translate the code into C++, to leverage its faster execution speed. At the end, the speed of the Python implementation resulted to be good enough, so it was kept as the final version, without rewriting. On top of that, the program relied heavily on advanced `NumPy` features: a C++ porting would require equivalent manual implementations, thus losing the intrinsic optimizations.

The following chapters briefly describe the libraries used in the project.

3.4.1 NumPy, SciPy, CuPy

`NumPy` [9] and `SciPy` [10] are the classical optimized libraries used for mathematical computations. `CuPy` [11] is an alternative to `SciPy`, that makes use of a GPU to parallelize and accelerate even more the functions it implements.

3.4.2 OpenCV

`OpenCV` [12] is the most common library used for image handling and computer vision tasks.

3.4.3 Numba

`Numba` [13] is a Just-In-Time compiler for Python: it enables to compile the code instead of interpreting it, improving on Python's infamous slow speed. On top of this, it enables to write Python kernels that can be compiled into CUDA code, enabling to fully exploit the GPU at the programmer's discretion.

3.4.4 Open3D

`Open3D` [14] is an open-source library to support the visualization of 3D data.

3.4.5 Other libraries

The libraries `trackpy` [15], `MyPTV` [16], `TracTrac` [17], as well as the MATLAB tool `4d-ptv` [18], are different existing solutions for attempting the task. A better analysis follows in the chapters where all the steps are examined.

The library `PyTorch` [19], one of the main pillars of machine learning in Python, is also used in some of the attempts at finding the best solution.

3.5 Unity

Unity [20] is a Game Engine, a software designed to help developers creating interactive applications — including, but not limited to, games. Unity natively supports the rendering of 3D geometries in real-time, while reacting to the user's inputs. These characteristics made Unity the perfect tool to create a visualizer for the reconstructed 3D data.

Chapter 4

Experimental setup

During development, everything was tested on a Jetson Orin Nano, while a final benchmarking was also done on a Jetson AGX Xavier. Both platform were connected to a set of 3 or 4 custom-built cameras. A schematic of the full hardware setup can be seen in figure 4.1.

4.1 Jetson Orin Nano

The Jetson Orin Nano [21] is a compact but powerful system developed by NVIDIA. It is powered by a 6-core Arm® Cortex®-A78AE v8.2 64-bit CPU. It provides an 8GB, LPDDR5 RAM, and accepts an SD card and an external NVMe as mass storage. It also features a NVIDIA GPU with Ampere architecture, that offers 1024 CUDA cores and 32 tensor cores. The power consumption can oscillate between 7 and 25W: during our tests, it was always set to maximize performance.

4.2 Jetson AGX Xavier

The Jetson AGX Xavier [22] is another embedded chip developed by NVIDIA. The Xavier generation means that this chip is older than the Orin, while the AGX family means that it is on the most powerful side of its generation.

The Jetson AGX Xavier is a system powered by a 8-core Carmel ARM CPU. It provides a 16GB, LPDDR4 RAM, and accepts an SD card and an external NVMe as mass storage. Its GPU is a NVIDIA chip with Volta architecture, with 512 CUDA cores and 64 tensor cores. As for the Orin, the power consumption was set to maximize performance during our tests.

4.3 Cameras

The cameras in use are custom-made, global shutter cameras, with a 960×960 sensor that provides 16 bits per pixel. Their maximum frame rate is 30 FPS, but the full resolution limits it at 24 FPS for data transfer speed. Each camera is equipped with an FPGA module, that performs a background subtraction and a binarization, providing as output binary images, with white bubbles over a black background. The connection between each camera and the processing element is realized following the MIPI protocol, which also enables to have synchronization among the frames captured by the different cameras.

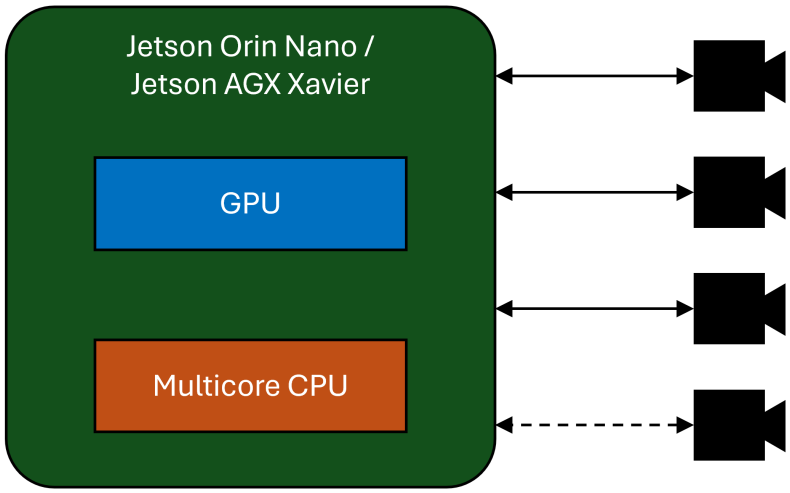


Figure 4.1: Schematic of the hardware setup

Chapter 5

Basic particle tracking pipeline

Commonly, the particle tracking pipeline is split into the following sub tasks:

1. ***Locate*:** considering each frame of each camera separately, find the pixel coordinates of all the bubbles in the image;
2. ***Link*:** consider two consecutive time instants. For each bubble in the first frame, link it to where it moved in the next frame, to form trajectories (called **tracklets**). This step must take into account the fact that bubbles may appear or disappear;
3. ***3D Matching*:** consider simultaneous frames of the different cameras. For each bubble in one camera, find which is, if any, the corresponding bubble in the other cameras. This information is then used to reconstruct the 3D position of the bubbles;
4. ***Visualization*:** display in a suitable way the reconstructed 3D scene on a 2D screen.

In literature, some approaches follow the order 1-2-3-4 (blue in figure 5.1), performing a camera-wise *Link*. Some other libraries chose to invert 2 and 3, obtaining a 1-3-2-4 order (orange in figure 5.1). This anticipates the 3D reconstruction before the *Link*, thus performing a single *Link* operation on 3D coordinates, obtaining 3D tracklets.

In the next paragraphs, each step will be analyzed separately. For each step, all solutions found in literature will be compared both among themselves, and with the ones developed within the scope of this thesis, to find the overall best one.

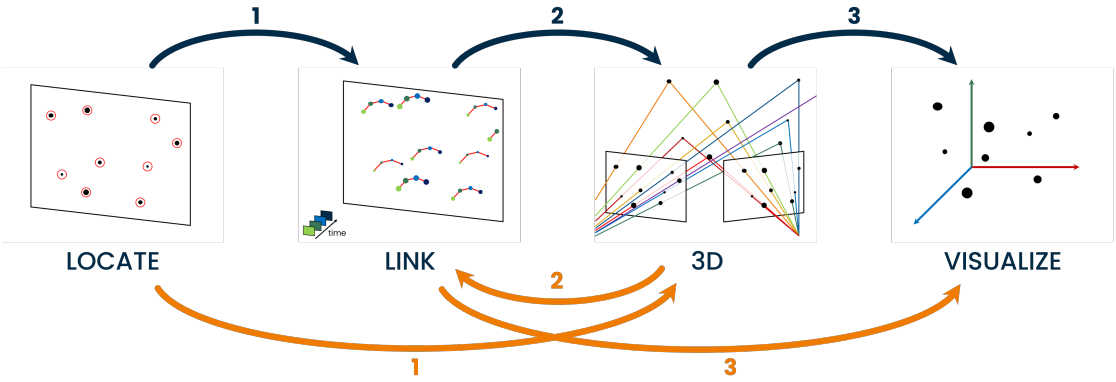


Figure 5.1: The two different orders in which the pipeline can be executed

Chapter 6

The *Locate* step

The task of the *Locate* step consists in extracting the positions of the bubbles in pixel coordinates, independently for each frame of each camera.

6.1 Requirements

6.1.1 Input

The *Locate* step receives as input the videos captured by the cameras, frame by frame. The images are pre-processed by an FPGA included in the cameras: the background is removed, and the resulting image is binarized with a threshold, to have distinct white bubbles over a black background. Figure 6.1 depicts an example input frame.

6.1.2 Output

The output of the *Locate* step is a couple of `numpy` arrays. An array called `positions` describes the coordinates of each bubble present in a frame. It is a four-dimensional, floating-point array, where `positions[C][F][B]` describes the B-th bubble of the F-th frame of camera C, in the form of an `(x, y[, area])` tuple. Bubbles are ordered in a random, arbitrary way for each frame: there is no correlation between two bubbles with the same B but different C and/or F.

Due to `numpy` limitations, the array is pre-allocated of a fixed size: while the number of cameras is fixed, an upper limit on the number of frames and bubbles must be decided before execution. Knowing which frame indices contain meaningful data is trivial, since the experiment time is choosable by the user. The number of bubbles found within a frame, however, is not known, mostly because it is not stable over time.

For this reason, a second array was introduced: `numTracers` is a two-dimensional, integer array. `numTracers[C][F]` carries the information of how many tracers are valid inside the F-th frame of camera C. The coordinates of the valid tracers will therefore be `positions[C][F][:numTracers[C][F]]`.

Figure 6.1: An example of frame returned by the cameras

6.1.3 Speed

When used on a setup of N cameras with frame rate f each, the *Locate* step would receive $N \cdot f$ independent frames each second. To respect the real-time constraint, the *Locate* step would therefore need to operate at a minimum of $N \cdot f$ FPS.

When the analysis was performed on the *Locate* step, the plan was to have 3 cameras working at 30 FPS, requiring a 90 FPS *Locate* step. Later, the cameras turned out

to be slower, at 24 FPS, but there were 4: the final *Locate* implementation was able to manage also these 96 FPS.

6.1.4 Quality

Ideally, all tracers should be detected, since errors in the locating process would propagate to future steps:

- **False positives:** the *3D Matching* phase will have more candidates, leading to *possible* wrong reconstructions: the *3D Matching* can both choose the correct bubble, or the one added by the error (or another real one);
- **False negatives:** the same bubble in another frame will not have the correct match, leading to *certain* wrong reconstructions.

As such, it is better to overpredict (false positives) than to miss bubbles.

It is however to be noted that the most important requirement is the speed: a worse implementation which is speedwise above target should be preferred to a better implementation that does not meet speed requirements.

6.2 State of the art

When searching on the Internet for existing solutions to perform the *Locate* task, the following approaches were found:

- The Trackpy [15] Python library (evaluated in section 6.3.1);
- The MyPTV [16] Python library (evaluated in section 6.3.5);
- The TracTrac [17] Python program (evaluated in section 6.3.8);
- The 4d-ptv [18] MATLAB script (evaluated in section 6.3.11).

In the aim of finding the best approach, the listed algorithms were evaluated in the same way as novel algorithmic ideas. In some cases, potential weaknesses in the algorithm were found: some of the new approaches developed within this thesis are therefore evolutions of these algorithms. For this reason, their description and evaluation is in the following chapters.

6.3 Approaches

The following sections describe the many different approaches evaluated for the *Locate* step. Their speed and quality is compared on a common 1-camera, 100-frame

sequence. Each approach reports an example frame: it is the result of performing the *Locate* on figure 6.2, which itself is a portion of the frame in figure 6.1.

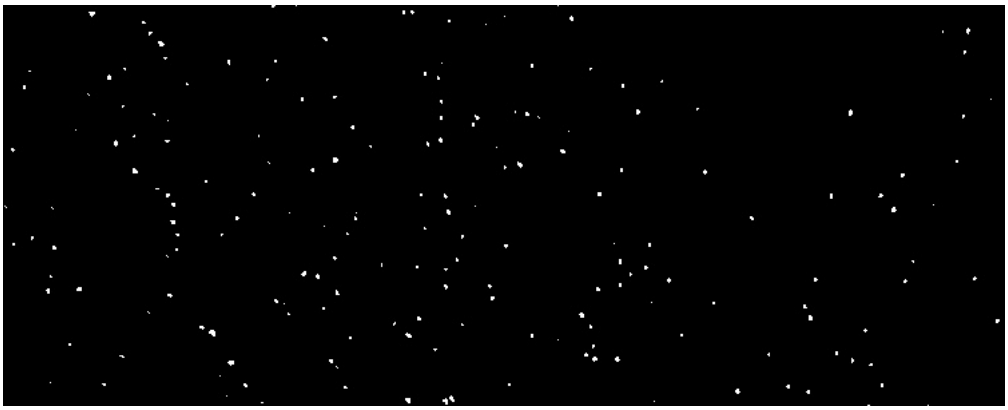


Figure 6.2: The portion of figure 6.1 used to display the locate result

6.3.1 Trackpy

Trackpy [15] is a particle tracking library developed by Soft Matter. Its `Locate` function performs the task required, if an extra output format transformation step is applied.

Algorithm

As described in the documentation, the `Locate` function implements the following algorithm:

1. Pre-process the image by performing a band pass and a threshold.
2. Locate all peaks of brightness, characterize the neighborhoods of the peaks and take only those with given total brightness ("mass").
3. Refine the positions of each peak.

Evaluation

As displayed in figure 6.3, the algorithm performed well on quality, finding about 85% of the tracers, but with some strange offset in the positions. The speed was however extremely low, at just 3 FPS.

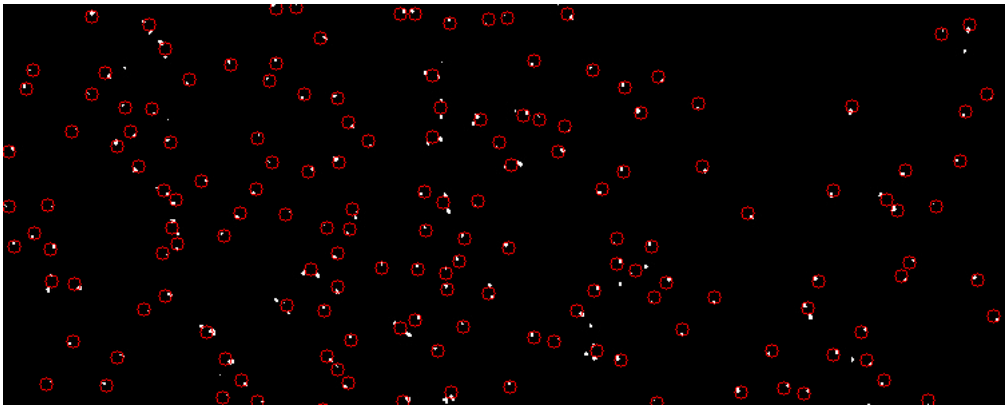


Figure 6.3: Trackpy's result

6.3.2 Trackpy (CuPy)

When profiling the Trackpy code, the functions `fourier_gaussian`, `uniform_filter1d` and `correlate1d` from `SciPy` took a considerable amount of time. For this reason, Trackpy's code was altered to use the `CuPy` library instead of `SciPy` for these operations. This aimed to exploit the GPU for faster computation.

Algorithm

The algorithm is the same as Trackpy's, with some extra steps required to transfer the various arrays to/from GPU memory. These transfers were reduced to the minimum, to reduce the overhead as much as possible.

Evaluation

Figure 6.4 shows the result, which for unknown reasons is different than Trackpy's: it lost the offset problem, but the percentage of identified bubbles reduced to 79%. Speed-wise, the algorithm is slightly faster, running at 7 FPS: still extremely far from the target.

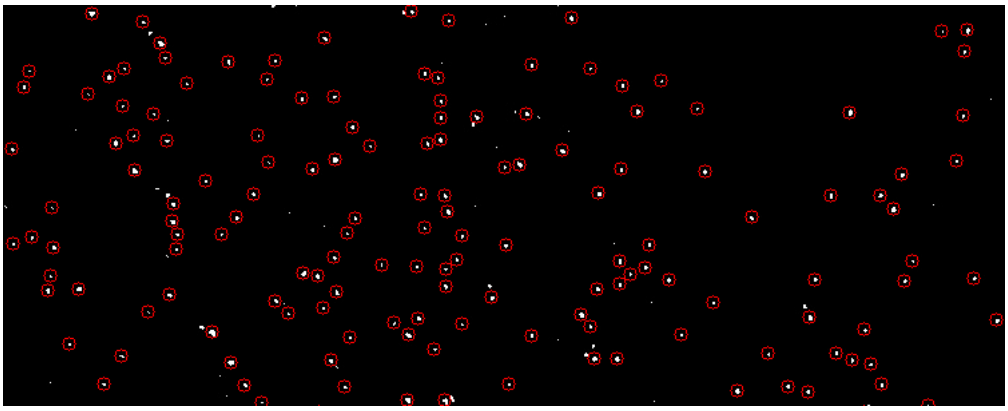


Figure 6.4: Trackpy with CuPy's result

6.3.3 CNN

The task of *finding the coordinates of the bubbles* can be read as *for each pixel, check if it is the center of a bubble*. The task of looking for the same thing across all pixels of an image is the foundation of image convolution and convolutional layers in neural network. As such, a SAME CONV neural layer was proposed, with a kernel size big enough for containing a full bubble. The CNN would transform the input image into a binary image, where "on" pixels would represent bubble centers.

Algorithm

- Initially, the single-layer CNN evaluates the image, to find centroids of the bubbles;
- At a second stage, a loop would collect all the "on" pixels of the image into a list of coordinates.

Evaluation

Initially, only a feasibility study was performed: the CNN was trained with just a single epoch of 8 images, to evaluate if the inference speed was good enough to justify a longer training. The result shown in figure 6.5 is promising for the little training performed, but it clearly needs more refinement.

Most importantly, the speed was much greater than the previous ones, at 55 FPS, but still far from the target. As such, further approaches were evaluated before performing a deeper training, to investigate if a greater speed was achievable.

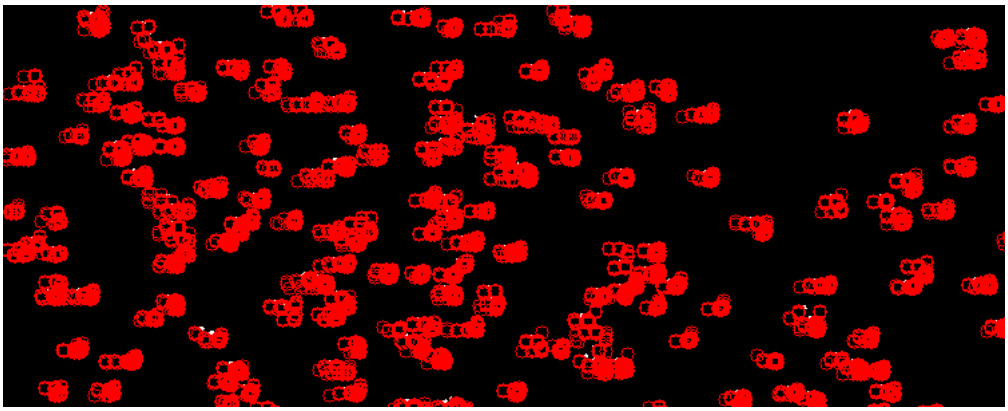


Figure 6.5: The CNN's result

6.3.4 Torch.unfold concept

The `unfold` function from `PyTorch` takes an image, and breaks it into either disjoint or partially overlapping tiles. The idea is to use a divide and conquer approach, where the full image is split into smaller images, hopefully making it faster.

Algorithm

The overall algorithm divides the image into patches, to then process each one separately. Different tile sizes were compared, to find the best, if any.

Evaluation

As visible in table 6.1, having smaller patches increases the time required to perform the overall *Locate*. This is likely due to the overlap between patches. The overlap is however necessary, to avoid bubbles split across patches to be considered as separate.

Patch size [px]	501×501	101×101	51×51	25×25	15×15	11×11
Time [s]	1.57	3.90	7.02	18.22	49.65	96.59

Table 6.1: Time required to process 1 frame with different patch sizes

6.3.5 MyPTV

MyPTV [16] is a Python library developed by Ron Shnapp for 3D particle tracking. Its `segmentation` command performs the task of the *Locate* step.

Algorithm

MyPTV has two different algorithms to perform the *Locate* step, "Labeling" and "Dilation". The best speed was obtained with "Labeling", that is composed by this sequence of steps:

1. Choose all pixels whose gray value is higher than a given threshold;
2. Pixels that are touching each other are considered to be "blobs" and grouped together;
3. For each blob, the center of mass, the bounding box size and the mass are computed.

Evaluation

The output quality is very good, at about 98% of bubbles correctly identified, as visible in figure 6.6. When concerning the speed, the library achieves 30 FPS, which is better than other approaches, but still only a third of the target speed.

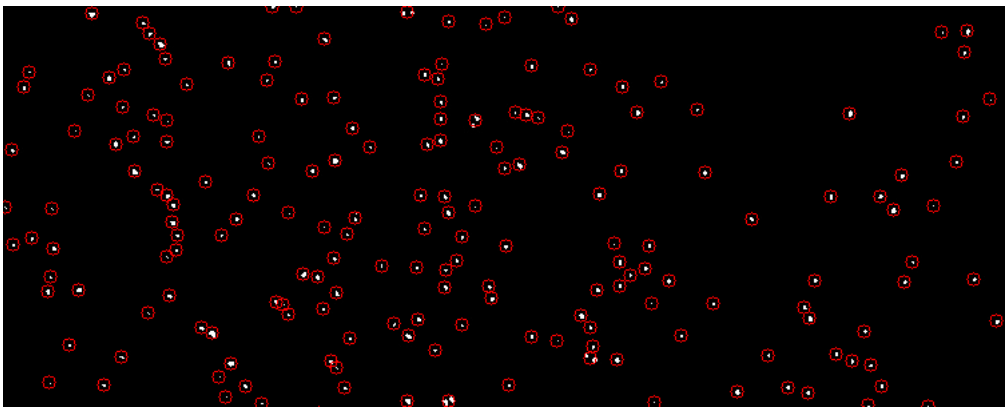


Figure 6.6: MyPTV's result

6.3.6 GPU algorithm

This is a custom approach, aimed at maximizing parallelization on GPU.

Algorithm

The algorithm analyzes each pixel in GPU, to then collect the results with a CPU portion:

1. For each possible radius r between 1 and a parameter `MAX_RADIUS`, execute a GPU kernel for each image pixel. The kernel operates the following:

   (a) Using Bresenham's rasterization algorithm, find the list of points that compose the circle of radius r around the pixel of the thread;

   (b) Find the fraction of "on" pixels among these, and use it as a pixel "score";

   (c) Decide if the pixel can be a bubble or not, based on the following conditions:

   - If within `MAX_RADIUS` there is already a bubble, this cannot be another one (to avoid finding the same bubble twice);
   - If the pixel has 100% score, and the surrounding pixels have a lower score, then the pixel is considered to be a bubble;
   - If this pixel was already marked as bubble in a previous iteration, leave it as it is;
   - If none of the previous conditions hold, and r is not `MAX_RADIUS`, continue to the next iteration;
   - If r is `MAX_RADIUS` and the pixel is a local maximum, consider it as a bubble;
   - Otherwise, the pixel is not a bubble.

2. In CPU, collect all the pixels marked as "bubble" into a common list.

Evaluation

While useful as a learning tool, this approach yielded poor results: speed was just reaching 15 FPS, and only 43% of the bubbles were identified, as visible in figure 6.7.

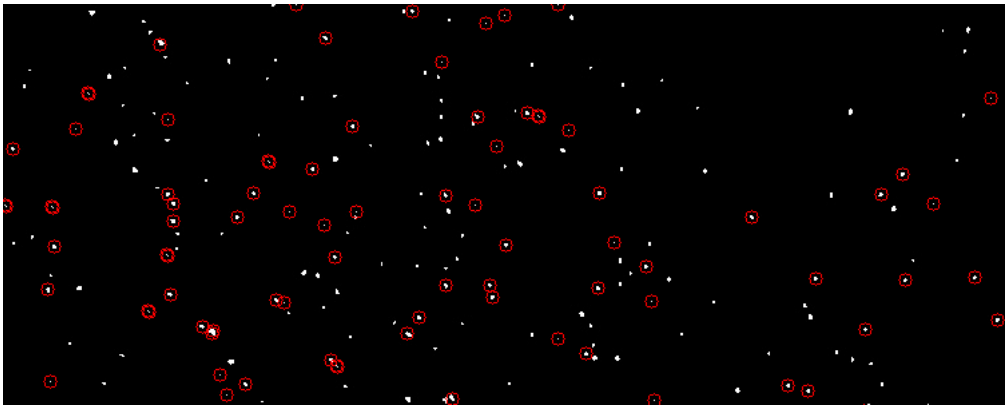


Figure 6.7: GPU algorithm's result

6.3.7 Iterative concept

All the approaches generally consider each frame to be independent, not related to the other ones. This is sometimes required, since for the first processed frame no information is available. However, since bubbles do not move much between frames, for subsequent frames an algorithm can reduce the searching window around where the bubbles could potentially be. This knowledge can reduce the searching space for these following frames.

While the idea of searching in smaller patches may seem silly due to the results discussed in section 6.3.4, here the situation is different. Instead of searching in smaller, but meaningless and overlapping regions, this concept uses meaningful and more sparse patches.

Algorithm

The algorithm stores position, velocity and acceleration of the previously found bubbles into a list. Velocity and acceleration are computed from the last and last two positions, respectively. If such information is not available, the values are considered to be 0.

The different frames are processed differently, based on their index:

1. First frame:
   (a) Perform a full frame *Locate*;
   (b) Add all the bubbles into the (previously empty) list.
2. All other frames:
   (a) Consider the bubbles currently in the list;
   (b) Estimate their future position based on the current velocity and acceleration;
   (c) In a patch around the predicted position, perform a *Locate*;
   (d) If the bubble is found, update its trajectory;
   (e) Otherwise, if a bubble is lost for some frames, remove it from the list.
3. Every N frames:
   (a) Update the existing bubbles according to step 2;
   (b) Perform a full frame *Locate*, to find potential bubbles that appeared in the last N frames;
   (c) Add to the list the bubbles that were found by this full frame *Locate*, and are not yet present in the list.

This concept is a "meta-algorithm", in the sense that it relies on another *Locate* algorithm as a backend. For the evaluation, multiple underlying algorithm were chosen as backend.

Evaluation

Both the quality and the speed of the meta-algorithm were worse than the original algorithms. For example, when using the Hough algorithm (see section 6.3.9), the quality was reduced from 84% to 43% (figure 6.8), and the speed from 55 to 15 FPS.

This meant that the divide and conquer approach was not advantageous, even if the tiles were meaningful.

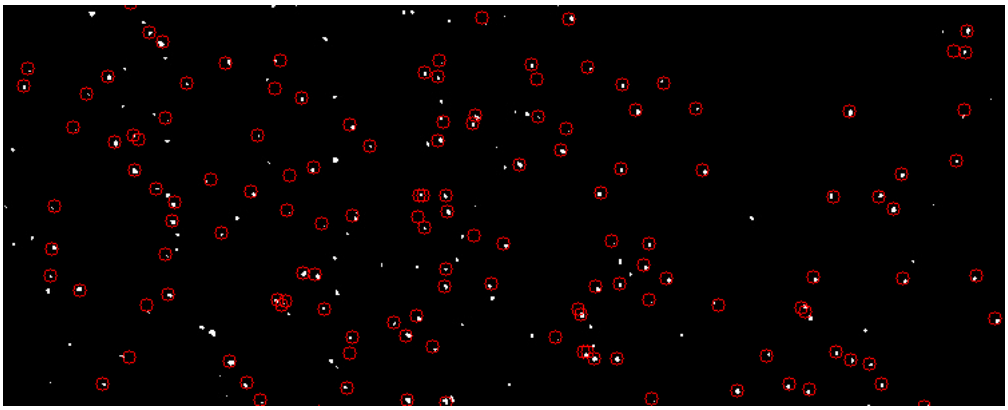


Figure 6.8: Iterative concept with Hough backend's result

6.3.8 TracTrac

TracTrac [17] is a fast Python software written by Joris Heyman, to track generic moving objects.

Algorithm

The detection is performed by means of DoG and LoG filters, together with Shi-Tomasi corner detection. Then, a sub-pixel refinement is applied.

Evaluation

The only output obtained from the script was an image like figure 6.9 for each frame. No numeric coordinate could be extracted from the execution, nor it was possible to understand the bubbles' positions from the output. On top of that, the speed was also extremely low (1 FPS), despite selling as "very fast software, that can track 10 000 particles per second".

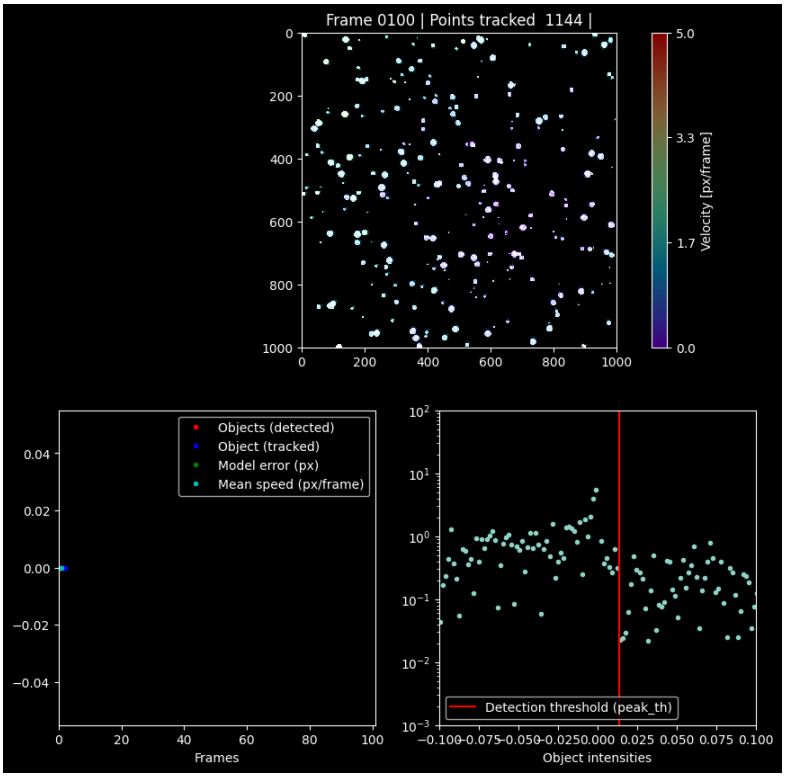


Figure 6.9: TracTrac's result

6.3.9 Hough transform

The Hough transform [23] is a technique commonly used in computer vision to find specific patterns, such as lines of circles. Here, a circular Hough transform is used to identify bubbles, which have a mostly circular shape. The implementation used is the function `HoughCircles` provided by `OpenCV`.

Algorithm

The Hough transform operates the following algorithm:

1. Use the Canny filter to identify the edges within the image;
2. For each pixel of this resulting image:
   (a) For growing radii r, compute the percentage p_r of the pixels that are "on" within the ones at distance r from the considered pixel;
   (b) Take note of $P = max(p_r)$ and $R = argmax(p_r)$ for the pixel;
   (c) Filter out pixels whose P is lower than a chosen threshold (no meaningful circle is found around them);
3. Perform non-max suppression on the values of P of all pixels;
4. Consider all the remaining pixels as bubbles, of corresponding radius R.

Evaluation

Figure 6.10 shows the results of the algorithm: about 84% of the bubbles were found, at a speed of 55 FPS.

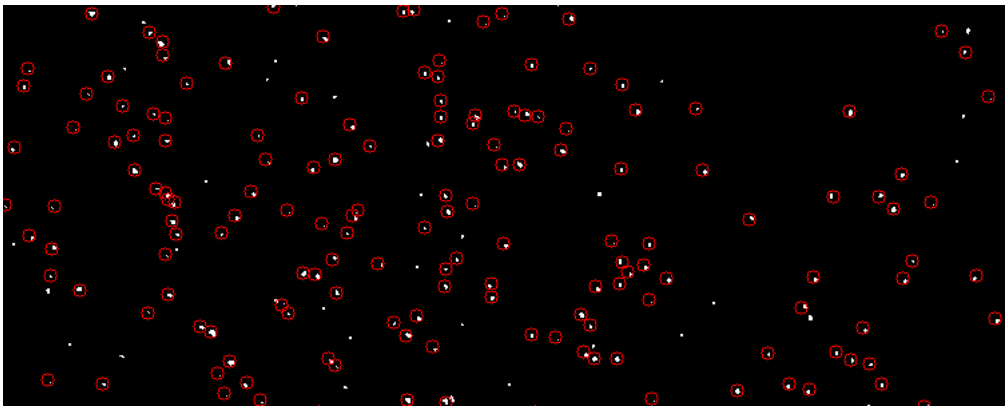


Figure 6.10: Hough's result

6.3.10 GPU Hough

The Hough algorithm operates iterating over all pixels of an image. As such, a new approach was developed, where this loop is replaced by a parallel GPU kernel call.

Algorithm

The general algorithm is the same as presented in 6.3.9, with all steps executed in GPU. In particular:

- Step 1 is performed by a kernel, with synchronization after it;
- Step 2 is executed by a separate kernel, with synchronization after;
- Steps 3 and 4 are run by a unique kernel, that executes non-max suppression on its own pixel, and if it not suppressed, uses atomic methods to add the pixel as coordinate center.

On top of the pixels of each frame, also the frames themselves were processed in parallel.

Evaluation

The performance of this approach is worse than the CPU Hough transform, both in quality and speed: the algorithm only identifies about 65% of the bubbles (as visible in figure 6.11), running at 17 FPS.

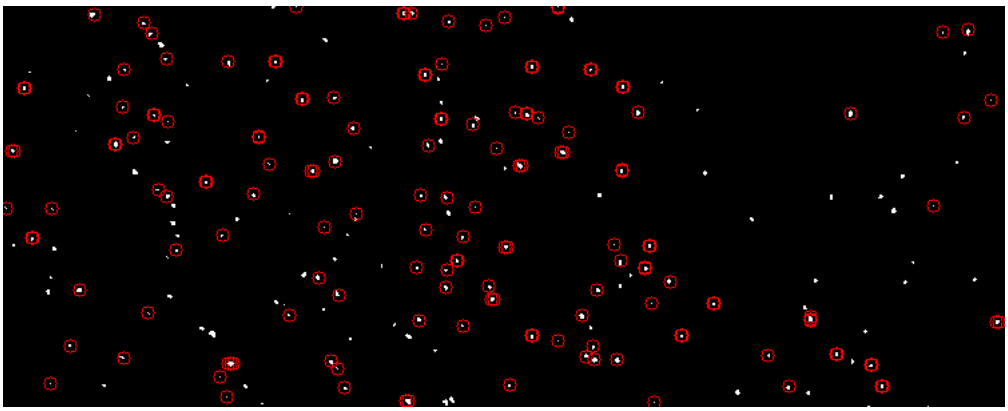


Figure 6.11: GPU Hough's result

6.3.11 4d-ptv

4d-ptv [18] is a MATLAB software for doing 4D Particle Tracking Velocimetry. In particular, the `CenterFinding2D` computes the same information as required by the *Locate* step.

Algorithm

1. Find candidate bubbles by finding local maxima, comparing each pixel with its 8 neighbors;
2. Refine the positions to sub-pixel accuracies;
3. Filter out non-gaussian bubbles.

Evaluation

Figure 6.12 shows that this approach has terrible (about 7%) accuracy. On top of that, speed is sub-optimal (40 FPS), and the script requires a non-standard calibration process.

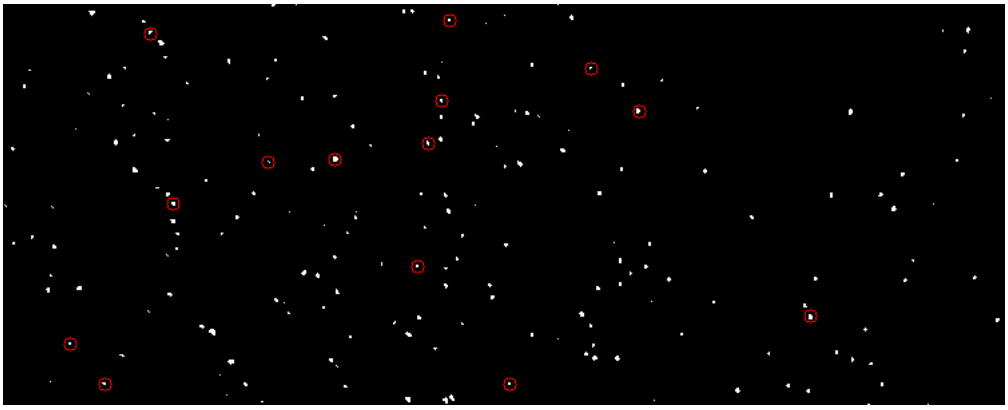


Figure 6.12: 4d-ptv's result

6.3.12 Image moments

Thanks to the preprocessing done by FPGA, the *Locate* task is equivalent to finding homogeneous regions of white pixels on top of a black background. This task is implemented by the function `FindContours` [24] from `OpenCV`, that finds a rectangular bounding box around pixels with the same intensity. The precise coordinates of the bubble can be found by computing the centroid of each bounding box, using image moments (described in section 3.3).

Algorithm

1. Through `OpenCV`'s `FindContours`, find all the bubbles for a given frame;
2. For each region in the output:
   (a) Compute the image moments with `OpenCV`'s `moments` function;
   (b) Use the moments of order 0 and 1 to compute the centroids;
   (c) Add the coordinates to the output list.

Evaluation

As visible in figure 6.13, the result quality is almost perfect, finding about 99% of bubbles. The speed of 67 FPS is extremely good as well, while still being lower than the target.

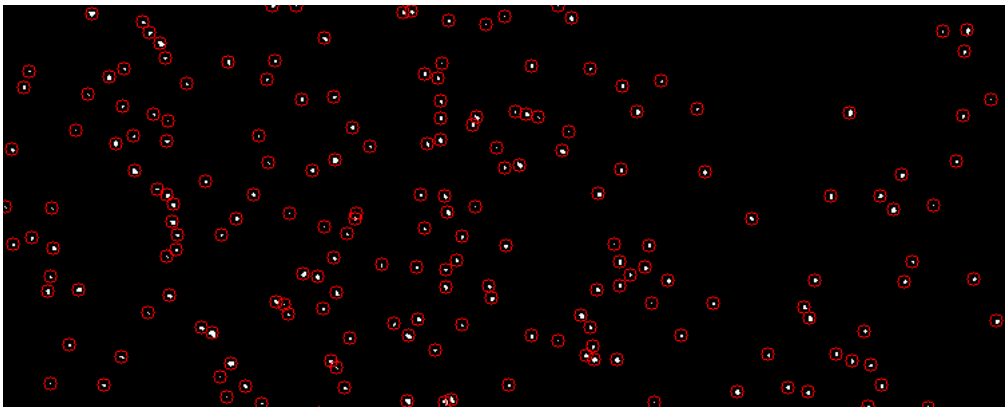


Figure 6.13: Image moments' result

6.3.13 GPU moments

From the previous approach, it was noticed that computing the moments on the different bubbles was a highly parallelizable task. In fact, the same `moments` function was called for each bubble found by `FindContours`.

On top of that, the `moments` function used to compute moments up to order 3 (for a total of 24 moments), while only moments of order 0 and 1 were used. As such, a tailored, GPU implementation was created to compute the centroids of the bubbles. The resulting implementation was also extended to process contours of multiple cameras at the same time, as well as considering more frames of the same camera in a batch.

Algorithm

1. Through `OpenCV`'s `FindContours`, find all the bubbles for a given frame;
2. Find the largest contour among them (this allows all GPU threads to work on the same data size, to avoid divergence: smaller contours are 0-padded to this common size);
3. For each contour found, a GPU thread runs the following kernel:
   (a) Iterate over all pixels, accumulating M_{00}, M_{01} and M_{10};
   (b) Use the moments to compute the centroids;
   (c) Add the coordinates to the output list.

Evaluation

Since the algorithm is the exact porting of the corresponding CPU algorithm to GPU, the output is the same (figure 6.13, 99% of bubbles found).

For the speed evaluation, some tests were conducted to find the best, if any, batch size. Figure 6.14 compares the speed of the CPU algorithm to the speed of the GPU algorithm, with respect to the total number of frames the GPU algorithm processes together. In particular, this number corresponds to the product between the number of cameras and the batch size per camera. It is visible that the GPU algorithm is faster, provided that at least 5 frames are processed concurrently, while reaching plateau performance when 10 frames are considered at each iteration. For the final evaluation, we chose a batch size of 20 per camera, leading to 80 frames processed at the same time by the GPU.

While increasing this batch size adds a delay on the output, this delay only produces a one-time latency, not accumulating over time. This is acceptable according to the project requirements, which allow for some processing latency, while requiring real-time regime speed.

The maximum speed achieved for this approach was 73 FPS, making this the fastest approach among all, but still not fully reaching the real-time target.

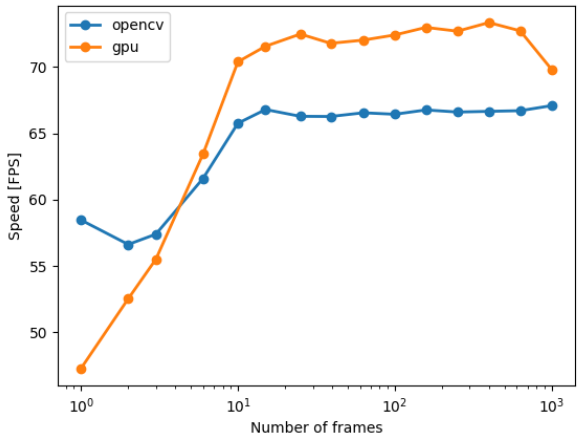


Figure 6.14: Comparing speed between CPU and GPU Image Moments algorithms with respect to number of frames processed

6.4 Final choice

All the previously described approaches were compared among each other and with the target speed. As visible in figure 6.15, none of the approaches was able to reach the target 90 FPS. Since no other approach or idea was available, the fastest algorithm (GPU moments, described in section 6.3.13) was chosen. Incidentally, the selected algorithm was also one of the best in terms of output quality.

Figure 6.15: Performance evaluation of different approaches for the *Locate* step

To compare the different approaches with each other, some modifications were required to ensure consistency, for simpler comparison. After choosing the final algorithm, it was implemented again from scratch, in order to make it as fast as possible, with no overhead. This resulted in a valuable speedup, which enabled the algorithm to execute at 102 FPS, faster than the target speed, as visible in figure 6.16.

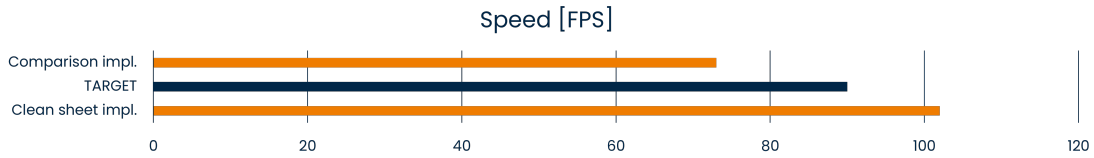


Figure 6.16: Computation speed of the chosen *Locate* algorithm before and after the re-implementation, compared with the target speed

Chapter 7

The *2D Link* step

The *Link* step aims to link together consecutive time instants, by joining the coordinates of each individual bubble across the various time instants it is seen. The result is a series of trajectories, or **tracklets**. Specifically, the 2D version of the *Link* step operates on 2D coordinates, producing 2D tracklets.

7.1 Requirements

7.1.1 Input

The input to the *Link* step coincides with the output of the previous step. If the pipeline is using *2D Link*, the full pipeline under exam is *Locate - Link - 3D Matching - Visualization*. This means that the input to this step is the output of the *Locate* step, as described in section 6.1.2.

7.1.2 Output

The coordinates of the particles in the tracklets follow the same format as the *Locate* output. A four-dimensional `positions` array describes the `(x, y)` coordinates of the bubbles inside `positions[C][F][B]`, C, F and B being the camera, frame and bubble indices, respectively. The difference with the *Locate* format is that values of B are scoped across the whole acquisition, not limited to the single frame. This means that all values with the same C and B will represent the same real bubble across the different frames.

With this representation, valid bubbles are not clustered at the smallest values of B: for example, bubble $B{=}0$ may disappear after some frames, leaving the rest of its tracklet to contain invalid positions. As such, the `numTracers` array is not anymore

enough to describe the valid positions. Instead, a different array is used: `validTracers` is a three-dimensional, boolean array. `validTracers[C][F][B]` contains the information of whether the bubble B of camera C was detected at frame F. False values indicate that, at a specific frame, the bubble was either not found yet, or already lost, or the overall number of bubbles traced by camera C was lower than B.

7.1.3 Speed

As for the *Locate* step, each second the *2D Link* has to process the bubbles from $N \cdot f$ frames, where N is the number of cameras and f is their frame rate. As such, the required speed for this step is the same 90 FPS that is required by the *Locate* step.

7.1.4 Quality

The overall quality of an algorithm can be estimated by combining manual observation with the number of resulting tracklets found. For the manual observation, the input video was overlay-ed with a tail composed of points and segments, describing the last few frames of trajectory. Figures 7.1 and 7.4 are examples of frames used for manual observation: the single links are quite small, it is hard to see them individually, it's much easier to consider the general view.

Possible situations of reconstructed links are:

- Link correctly detected: the number of total tracklets does not change from the previous frame, and the link is coherent with the rest of the trajectory;
- Link not detected: visually, it's hard to notice the missing link; however, this splits the tracklet into two pieces, increasing the number of tracklets by 1;
- Wrong link detected: the number of tracklets remains the same, while an inconsistent movement is visible by eye.

As such, a good reconstruction is one with few tracklets and a coherent visual representation.

7.2 State of the art

For the *Link* step, online research was less successful: no new approach was found, and only some of the libraries found for the *Locate* step were also performing the task:

- Section 7.3.1 explores the Trackpy [15] Python library;
- Section 7.3.2 explores the MyPTV [16] Python library.

As for the *Link* step, they are described and evaluated in the following chapters.

7.3 Approaches

The following sections describe the different approaches evaluated for the *2D Link* step. They are evaluated on a 201-frames video [25], whose frames look like figure 6.1. For the different approaches, a crop of a sample frame is reported as per the *Locate* approaches (section 6.3), with the tail of the tracklet. Full videos are available on YouTube, following the links in the corresponding citations.

7.3.1 Trackpy

The `link` function from the Trackpy [15] library is able to perform the *Link* task both in 2D and in 3D.

Originally, it required the located positions to be inside a `Pandas Dataframe`, and it used to convert it into a `NumPy` array. However, since our data was already inside a `NumPy` array with the same format, the library was altered to avoid this useless conversion, thus saving time.

Algorithm

The Trackpy library implements the Crocker-Grier linking algorithm [26].

Evaluation

For a single camera, the linking speed was 40 FPS. If different cameras were analyzed in parallel processes, 3 cameras could be processed at an overall speed of 120 FPS.

The quality was good at a visual inspection (see figure 7.1 or the full video [27]), and the total number of tracklets was around 6500.

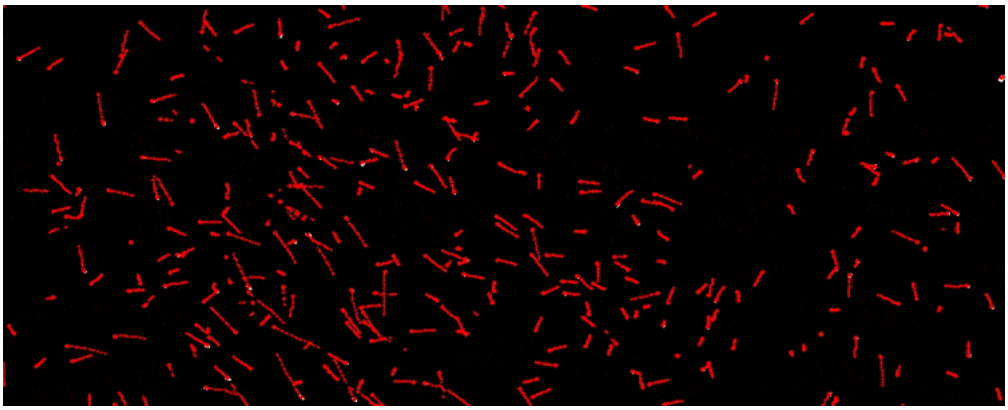


Figure 7.1: A frame from the Trackpy *2D Link* result, full video available at [27]

7.3.2 MyPTV

MyPTV [16] implements *2D Link* with the `2D_tracking` command.

Algorithm

The library implements the "best estimate method" described by Ouellette-Xu [28]: a nearest neighbor initial guess, followed by 4-frame tracking for the following frames.

Evaluation

The speed of this approach, lower than 1 FPS, was unacceptable: as such, the quality was not even evaluated.

7.3.3 Kalman CPU

The iterative concept for the *Locate* approach (described in section 6.3.7) was using the trajectories to predict the future position of each bubble. As such, it was already performing the *2D Link* task. For this reason, a modified version was considered as a novel *2D Link* approach.

Algorithm

The algorithm starts with an empty list of previously found bubbles. It then loops over these steps, for each frame in the video:

1. Store the positions of the located bubbles in a suitable data structure. Based on the settings, a basic coordinate list or a 2D binary tree could be used;
2. For each bubble in the "previous bubbles" list:
   (a) Compute its velocity and acceleration from the last trajectory points, if available;
   (b) Compute a predicted position;
   (c) Find the candidate bubbles in the next frame:
      - If the binary tree was used as representation, consider all bubbles within a *delta* from the predicted position;
      - If the bubbles list was used as representation, consider all bubbles;
   (d) Among the candidates, chose the closest one (in terms of Manhattan distance) to the predicted position;
   (e) Check that the distance of the match is reasonable:
      - If it is, link the two bubbles, and mark the chosen one as linked;
      - If it is not, consider the bubble lost for this frame. If the bubble is lost for multiple consecutive frames, it is removed from the list;
3. Add all unlinked bubbles to the list, as new tracklets.

The 2D binary tree was added to reduce the amount of possible matches, by splitting the coordinates into 4 bins for every node layer, storing the bins as Python lists. To obtain the bubbles within a *delta* from the predicted position, the tree would consider all bins that (at least partially) satisfied the condition.

Evaluation

Considering the tree choice, it was possible to choose between:

- no tree at all: as stated in the algorithm overview, the original `NumPy` array of bubbles was used as set of candidates;
- tree with 0 layers: no split was performed, therefore the original `NumPy` array was simply translated into a Python list. The distance computation is the same as with no tree, with added overhead of creating and accessing the Python list instead of the `NumPy` array.
- tree with maximum number of layers: each leaf bin represents a single pixel of the original image, therefore will contain either 1 or 0 bubbles. The number of layers is $\lceil log_2(P) \rceil$, where P is the side, in pixels, of the captured image. In our case, $P{=}960$, prompting to choose 9 layers. This option has the most fine-grained way to choose the bubbles at a specific distance. It however adds the most overhead related both to constructing a bigger tree, and traversing multiple paths to construct the set of bubbles to be evaluated.
- tree with intermediate number of layers: this is a compromise between efficiency in building and using the tree, and reducing the number of distances to compute.

These different approaches are compared in figure 7.2: when considering the tree, the best choice is a compromise between granularity and tree complexity. However, the overhead of building the tree does still not match the performance without it, thanks to the extreme optimization of the `NumPy` library. As such, the version with no tree was the chosen one.

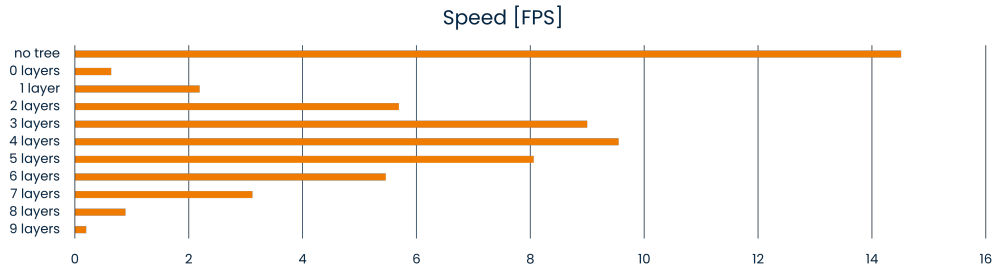


Figure 7.2: Comparing speeds for different tree sizes in the Kalman CPU *2D Link* approach

As visible in figure 7.2, the overall speed is limited to about 14 FPS, much slower than Trackpy. On top of that, the result is also slightly worse: while it looks good at visual inspection (see figure 7.3), the total number of traces is higher than Trackpy, at about 8400 tracklets found.

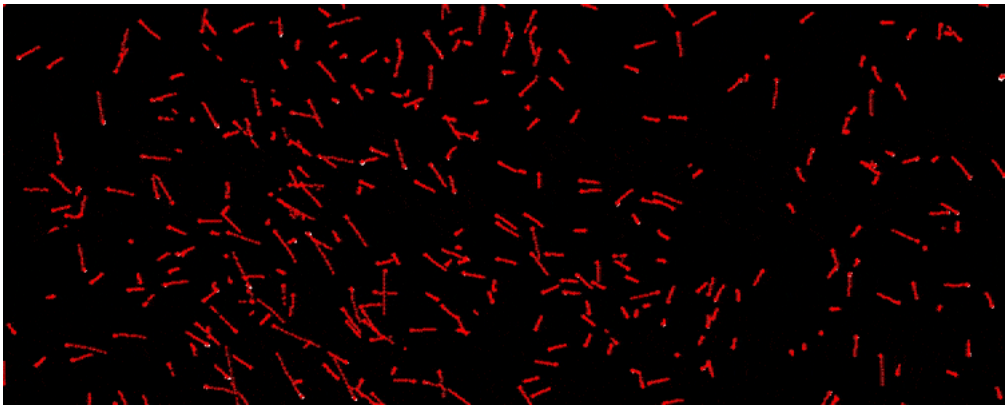


Figure 7.3: A frame from the Kalman CPU *2D Link* result, full video available at [29]

7.3.4 Kalman GPU

The Kalman CPU approach performs the same operation over all the bubbles of all the images. A GPU implementation was therefore evaluated, to test its potential parallelization.

Algorithm

The algorithm is the same as the previous approach, with step 2 transformed into a GPU kernel. This kernel processes all bubbles of all images captured at the same time instant.

Evaluation

The speed is quite faster than the Kalman CPU approach, but still considerably slower than Trackpy, standing at 68 FPS. The quality is however worse: the number of tracklets increased to about 8900, and a visual inspection found some inconsistencies (in figure 7.4, two tracklets in the middle include an unreasonable jump).

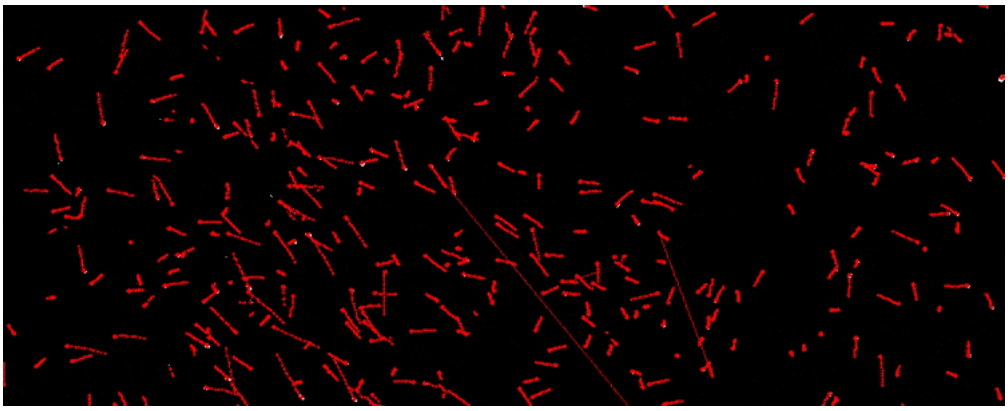


Figure 7.4: A frame from the Kalman GPU *2D Link* result, full video available at [30]

7.4 Final choice

Figure 7.5 compares the speeds of the various *2D Link* approaches: Trackpy is the only one with an adequate speed. On top of that, it is also the approach with the best overall quality. As such, if the pipeline is traversed in the *Locate - Link - 3D Matching - Visualization* order, the *Link* step will use the Trackpy implementation.

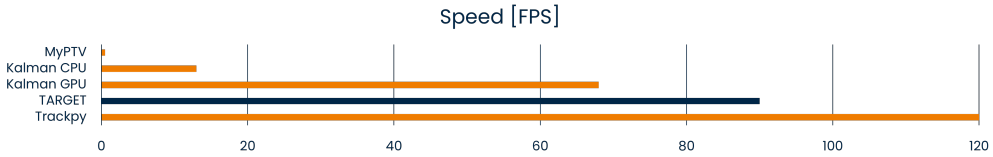


Figure 7.5: Comparing the speeds of the different *2D Link* approaches

Chapter 8

The *3D Matching* step

The *3D Matching* step has the objective of leveraging the different camera views to estimate the depth (and, consequently, the 3D coordinates) of the bubbles. The depth estimation is done with the technique of stereoscopy (explained in section 3.2). In particular, the *3D Matching* step needs to match the same bubble across different cameras, to then call `OpenCV`'s `triangulatePoints` function for reconstructing the 3D positions.

8.1 Requirements

8.1.1 Input

Depending on the pipeline order, the input to the *3D Matching* step can be either the output of the *Locate* step or the *2D Link* step. In both cases, the input is composed of two arrays:

- the array of the coordinates `positions[C][F][B]` (described in sections 6.1.2 and 7.1.2), which has the same format in both cases;
- the representation of valid coordinates, which varies based on the previous step: *Locate* has a `numTracers[C][F]`, as explained in section 6.1.2, while *2D Link* uses a larger `validTracers[C][F][B]`, described in section 7.1.2.

In both cases, the *3D Matching* operates on the coordinates of the valid 2D coordinates: an initial step extracts such coordinates from the `positions` array, leveraging the other array in the correct way. After that, the *3D Matching* can be executed without caring about the source of the data.

8.1.2 Output

The *3D Matching* step's output consists in a couple of arrays, similar to the output of the *2D Link* (described in section 7.1.2).

The 3D coordinates of the bubbles are stored in a three-dimensional, floating-point array called `positions`. `positions[F][B]` contains the coordinates of the B-th bubble in the F-th time instant, as a tuple `(x, y, z)`. The camera index disappeared, since this step combines the information from all cameras into a single, 3D description of the bubbles.

To represent the valid bubbles, a three-dimensional, boolean `validTracers` array is used. `validTracers[F][B]` marks whether `positions[F][B]` contains a valid position or not. Similarly to the `positions` array, the passage from 2D to 3D removes the dimension of the camera index.

Depending on the source of data, this step can either produce plain 3D coordinates, or 3D tracklets. If the input is the *Locate* step, then the bubble coordinates will be all grouped towards smaller Bs, and there will be no correlation between bubbles with the same index in consecutive frames. Instead, if the input is the *2D Link* step, the distribution of real coordinates within the B dimension of the arrays will be less regular, due to the added constraint that "same value for B implies same real bubble".

8.1.3 Speed

To respect the real time constraint, the *3D Matching* step should operate at 30 FPS.

8.1.4 Quality

The matching algorithms start from a bubble in one "main" camera, and try to find the matching bubble on the other cameras. Such attempt can have three outcomes:

- correct match: the bubble is matched to the correct one in the other camera. This is the ideal case, since it would lead to a correct 3D reconstruction;
- missing match: the original bubble is not matched to another one. Since the setup is composed by more than 2 cameras, a missed match is not too terrible, since it is still possible that the matches in the other cameras are correct, to have a correct 3D reconstruction;
- wrong match: the bubble is matched to a wrong one. This leads to certain reconstruction errors, since the `triangulatePoints` function will have either wrong or incoherent information.

The ideal matching algorithm is therefore one that produces correct matches for all bubbles, but if it's not possible, it's better to have missing matches rather than wrong matches.

8.2 State of the art

For the *3D Matching*, no existing solution was found when looking on the Internet: the matching is usually done on full images, not on single sets of coordinates. The only starting point available for this step was the original tool developed by the research group, the objective of the acceleration. In particular, it allows to choose between an epiline-only approach (examined in section 8.3.2) and a brute force algorithm (evaluated in section 8.3.6).

When exploring the Trackpy [15] documentation, the phrase "Trackpy is a Python package for particle tracking in 2D, 3D" may lead to think that Trackpy also performs the *3D Matching* task. Contrarily, this means that Trackpy is able to perform tracking with 3D data in input, collected for instance using the confocal microscopy technique. As such, it is not a useful approach for this step.

8.3 Approaches

The following chapters describe the different approaches explored for the *3D Matching* step The evaluation was performed separately for speed and quality, with the speed measured on the same dataset as the *2D Link* step. For the quality, it was not possible to have a ground truth containing the correct match between two cameras, neither with a real dataset nor with a synthetic one. As such, a manual classification of correct/missed/wrong matches was done. As visible in figure 8.1, the displacement between the view of two cameras is mostly constant: checking this consistency allows a human eye to evaluate the correctness of a match.

The qualitative evaluation was performed using two Blender-generated datasets, composed by a single frame, with respectively 30 and 100 bubbles. An attempt was done with more bubbles, but the image was too crowded to identify correct or wrong matches.

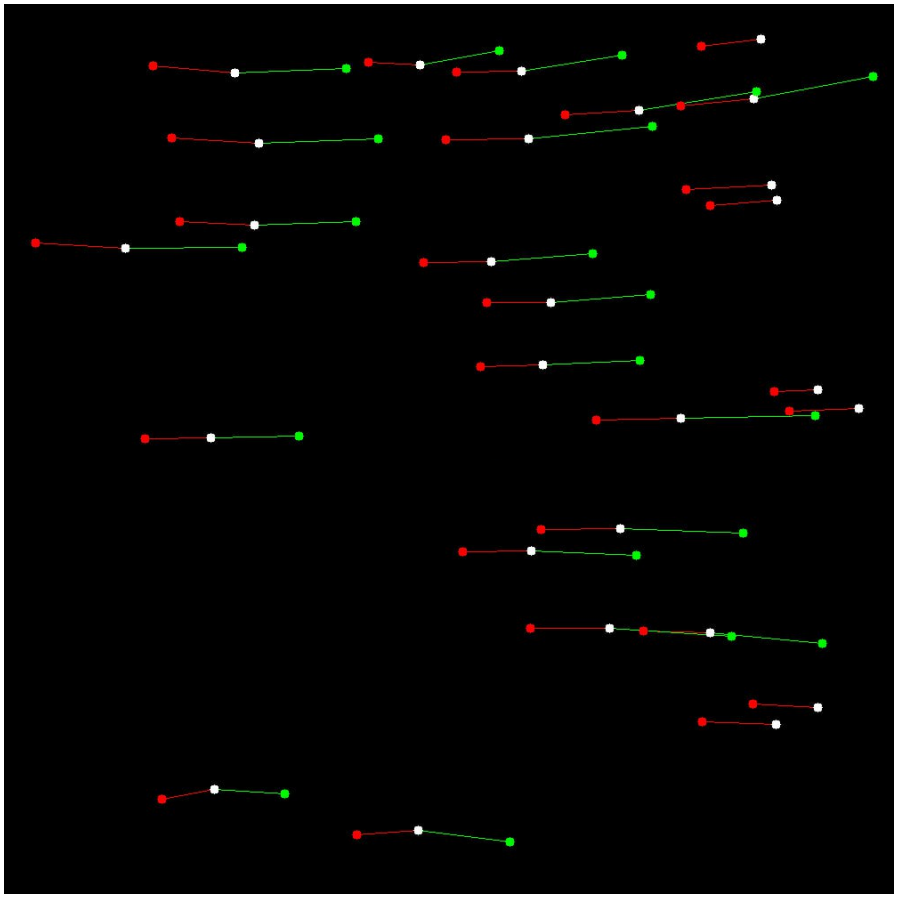


Figure 8.1: An example of *3D Matching*: the bubbles seen by the main (white) camera, matched to the corresponding bubbles seen from the other two (red, green) cameras

8.3.1 Long trajectory

The research group that commissioned this thesis was working in parallel on a way to estimate calibration data without the need of a calibration process. Instead, the model developed tried to perform a *3D Matching* (without knowledge of the epilines), to infer the calibration data from it. Since this task was in common among the two projects, their solution was also considered and evaluated for the scope our purpose.

Algorithm

The solution uses a Deep Learning model, Lightglue [31], to match long trajectories seen from different cameras. In particular, the model performed the match on a specific frame by looking at the 200-frames-long trajectories that started in the frame itself.

Evaluation

While this approach was useful in the complex situation of auto-calibration, for our task it was too computationally intensive. Indeed, the maximum speed obtainable with this approach was 3 FPS, too far from the target 30 FPS.

8.3.2 Closest to epiline

As described in section 3.2.2, a specific point in a camera will be seen on a specific epiline on another camera.

In traditional stereoscopy, the epiline is a set of discrete pixels, while in our case it's a continuous line of floating-point values. On top of that, the many pixels of each bubble in the image get compressed into a single point, without size. Due to these two factors, it is impossible to have a bubble perfectly on top of the epiline: instead, the match is selected as the closest bubble to the epiline, provided that the distance is under a reasonable threshold.

This is one of the approaches originally used in the MATLAB script provided by the research group. It was considered as a candidate algorithm, and as such it was re-implemented in Python for easier comparison.

Algorithm

Chosen a "main" camera, the algorithm processes each other camera with the following steps:

1. For each bubble in the main camera:
   (a) Compute the coefficients a, b and c of the epiline $ax + by + c{=}0$;
   (b) Compute the distance (with a scale factor) of all bubbles (x_i, y_i) in the side camera from the epiline: $d_i = ax_i + by_i + c$;
   (c) Select as candidate match the bubble with $d = min_i(d_i)$;
   (d) Compare d with the reasonability threshold T:
      - If $d < T$, consider the bubble as a valid match;
      - Otherwise, consider the original bubble as unmatched.

Evaluation

The algorithm had excellent speed, running at 300 FPS, which is 10x faster than the requirements. While the quality looked quite good with the smaller dataset (with 28/0/2 correct/missed/wrong matches), the performance was much worse with more bubbles (51/4/45). As such, there was space for improvement, trading some of the useless speed with better quality. In fact, a possible improvement could be to have some more data to describe the bubbles: the match could be chosen as the bubble with the most similar description, among those close to the epiline.

8.3.3 Epilines + median correction

As previously stated, the offset between two camera views of the same bubble is consistent across all the bubbles in the scene. This fact is used in this approach to find and potentially correct mistakes. Originally, the average displacement was hypothesized to be a good comparison term for detecting errors, but it would be very sensitive to errors. Instead, the median displacement is used, since the number of errors in one direction is estimated to cancel out the number of errors in the opposite direction: this leaves the correct displacements towards the center range. The median used is a vector whose components are the medians of the displacement components. As an example, displacements of $(10, 1)$, $(11, 0)$ and $(-3, -1)$ would generate a median displacement of $(10, 0) = (med(10, 11, -3), med(1, 0, -1))$.

Algorithm

For non-main camera is processed with the following steps:

1. For each bubble in the main frame:
   (a) Compute the coefficients a, b and c of the epiline $ax + by + c$=0;
   (b) Compute the distance (with a scale factor) of all bubbles (x_i, y_i) in the side camera from the epiline: $d_i = ax_i + by_i + c$;
   (c) Select as candidate match the bubble with $d = min_i(d_i)$;
   (d) Compare d with the reasonability threshold T:
      - If $d < T$, consider the bubble as a valid (temporary) match;
      - Otherwise, consider the original bubble as unmatched.
2. Compute the median of all the matches of the frame;
3. For each bubble in the main frame:
   (a) Compute the displacement and compare it with the median:
      - If both the difference in length and the angle between the vectors are under specific thresholds, confirm the match, and continue with the next bubble;
      - Otherwise, correct the match by performing the following steps;
   (b) With a procedure similar to steps 1.a to 1.d, find the N (parameter) bubbles closest to the epiline;
   (c) Among them, find the one whose displacement is the most similar to the median displacement;
   (d) Check the correctness of this new match:

- If the bubble is further than the threshold T (step 1.d) from the epiline, consider the match wrong and remove it;
- If the new displacement still does not satisfy the distance and angle from the median, consider the match wrong and remove it;
- Otherwise, correct the previous match with this one.

Evaluation

Clearly, the introduction of the epilines check slowed down the algorithm, that can now process 55 frames per second. However, the speed was traded with an improvement on the quality: for the 30-bubbles dataset, the distribution of correct/missed/wrong matches was 27/2/1, and 74/20/6 for the 100-bubbles dataset.

8.3.4 Epilines + short trajectory

As discussed in section 8.3.2, it would be beneficial to have some more information about the bubbles, more than just the position. While section 8.3.1 shows that evaluating many frames of the trajectory can lead to high computational cost, considering just some frames of trajectory could be a good way to describe a bubble to facilitate the matching.

Algorithm

Each camera is processed according to the following algorithm:

1. Compute the N-frames trajectory of all bubbles of both the main and the side cameras, storing them as lists of relative offsets between consecutive positions;
2. For each bubble in the main camera:
   (a) Compute the coefficients a, b and c of the epiline $ax + by + c{=}0$;
   (b) Compute the distance (with a scale factor) of all bubbles (x_i, y_i) in the side camera from the epiline: $d_i = ax_i + by_i + c$;
   (c) Consider only the bubbles with $d_i < T$, for a specific threshold T (if there are none, leave the bubble unmatched);
   (d) Compute the similarity between the bubble in the main camera and the others, considering the trajectory values;
   (e) Select as match the bubble with highest similarity, provided that it has at least a minimum value for that (if not, leave the bubble unmatched).

Evaluation

While working at acceptable speed (125 FPS), the quality of the result was not so great: in the 30-bubbles dataset the number of correct/missed/wrong matches was 12/2/16, and it was 62/8/30 for the 100-bubbles dataset.

8.3.5 Epilines + KNN

Traditional stereoscopy finds the correct pixel on the epiline by comparing the patch around the original pixels to patches around the epiline. This idea of "looking at the neighborhood" was evolved to our case into a k-Nearest Neighbor search.

Algorithm

The algorithm processes each camera with the following steps:

1. Compute the kNN of all bubbles of both the main and the side cameras, storing them as relative offsets from the bubble's position;
2. For each bubble in the main camera:
   (a) Compute the coefficients a, b and c of the epiline $ax + by + c{=}0$;
   (b) Compute the distance (with a scale factor) of all bubbles (x_i, y_i) in the side camera from the epiline: $d_i = ax_i + by_i + c$;
   (c) Consider only the bubbles with $d_i < T$, for a specific threshold T (if there are none, leave the bubble unmatched);
   (d) Compute the cosine similarity between the bubble in the main camera and the others, considering the positions of the kNNs;
   (e) Select as match the bubble with highest cosine similarity, provided that it has at least a minimum value for that (if not, leave the bubble unmatched).

Evaluation

Similarly to the previous approach, this algorithm has an acceptable speed (122 FPS), but an extremely bad quality, with 2/28/0 correct/missed/wrong matches in the 30-bubbles dataset, and 1/98/1 in the 100-bubbles dataset.

8.3.6 Brute force

This approach is one of the two implemented in the original MATLAB script: it leverages the fact that more than two cameras are present, not only as a confirmation, but as a knowledge-extracting method.

Given a bubble on the main camera, it should be matched to a specific bubble on the other two cameras. If the 3D position is reconstructed from the main and one side camera, the result should be similar to the reconstruction done with the main and the other side camera. On the other side, wrong matches would reconstruct totally different 3D coordinates.

Algorithm

1. For each bubble in the main camera:
   (a) Reconstruct the 3D position, matching it with all the bubbles on one non-main camera. The result will be a set of 3D points (P_i);
   (b) Do the same, with the other non-main camera (to obtain a set P'_j);
   (c) Find the points $p \in P_i$ and $p' \in P'_j$ whose distance is the smallest;
   (d) Check their distance:
      - If it is below a specific threshold, the two reconstructions are considered to be the same bubble, hence the two matches are approved;
      - Otherwise, the reconstructed bubbles are the closest plausible, but still different bubbles, hence the match is considered missing.

Evaluation

This approach was not evaluated directly, since just limiting to the bubbles close to the epiline could highly reduce its computational cost. This algorithm was therefore only considered as starting idea for the approach described in the next paragraph.

8.3.7 Epilines + brute force

Given the knowledge that the correct match is near the epiline, there is no need to perform a brute force check on all the bubbles, as proposed in the previous approach. Instead, it is enough to check the bubbles close to the epiline.

Algorithm

1. For each bubble in the main camera:
   (a) Compute the distance d (with a scale factor) of all bubbles in each side camera from the corresponding epiline;
   (b) Consider only the bubbles with $d < T$, for a specific threshold T (if there are none, leave the bubble unmatched);
   (c) Reconstruct the 3D position, matching it with all the bubbles on one non-main camera. The result will be a set of 3D points (P_i);
   (d) Do the same, with the other non-main camera (to obtain a set P'_j);
   (e) Find the points $p \in P_i$ and $p' \in P'_j$ whose distance is the smallest;
   (f) Check their distance:
      - If it is below a specific threshold, the two reconstructions are considered to be the same bubble, hence the two matches are approved;
      - Otherwise, the reconstructed bubbles are the closest plausible, but still different bubbles, hence the match is considered missing.

Evaluation

This approach yields excellent result, with no mistakes recorded. This is achieved thanks to its requirement to have a "3-way confirmation" on the bubbles: when in doubt, it prefers to leave the bubble unmatched. The qualitative results were 18/12/0 correct/missing/wrong matches on the 30-bubbles frame and 55/45/0 on the 100-bubbles dataset. This however comes at a slight cost of speed, since this algorithm is only able to reach 19 FPS.

8.4 Final choice

Figure 8.2 compares speed and quality of the different approaches. Due to the real-time constraint, the long trajectories and the brute force algorithms are discarded. Among the remaining approaches, the median approach is the best one, hence it is the selected one.

Approach	Speed	Correct / Not found / Wrong	
Epilines only	300 FPS	28 / 0 / 2	51 / 4 / 45
Epilines + median	55 FPS	27 / 2 / 1	74 / 20 / 6
Epilines + KNN	122 FPS	2 / 28 / 0	1 / 98 / 1
Epilines + short trajectory	125 FPS	12 / 2 / 16	62 / 8 / 30
Epilines + "bruteforce" 3D	19 FPS	18 / 12 / 0	55 / 45 / 0
Long trajectory	3 FPS	-	-

Figure 8.2: Comparing speed and quality of the various *3D Matching* approaches

Chapter 9

The *3D Link* step

The *Link* step aims to link together consecutive time instants, by joining the coordinates of each individual bubble across the various time instants it is seen. The result is a series of trajectories, or **tracklets**. Specifically, the 3D version of the *Link* step operates on 3D coordinates, producing 3D tracklets.

9.1 Requirements

9.1.1 Input

The data for this pipeline step comes from the output of the *3D Matching* step, described in section 8.1.2. The input is therefore composed of two arrays, `positions` and `validTracers`.

9.1.2 Output

Similarly to the *2D Link* step (whose output is described in section 7.1.2), the output format is the same as the input, with the added constraint that equal bubble indices across different frames imply same real-life bubble.

The output will be therefore composed of the three-dimensional, floating-point `positions[F][B]` array, with the `(x, y, z)` coordinates of bubble B at time F, and the corresponding two-dimensional, boolean validity array `validTracers`.

9.1.3 Speed

Working directly on 3D data and not on the single cameras, the *3D Link* only needs a speed of 30 FPS, similar to the *3D Matching* step.

9.1.4 Quality

Similarly to the *2D Link* step, the quality can be estimated by number of tracklets and visual inspection.

9.2 State of the art

Similarly to the previous steps, further research did not find more tools for performing the *3D Link* task. Among the ones already found, only Trackpy [15] was able to perform *3D Link*. Its algorithm and performance are evaluated in section 9.3.1.

9.3 Approaches

In the following sections, the two approaches to *3D Link* are explained. They are evaluated on the same 201-frames dataset used for the *2D Link* step, preprocessed by the *3D Matching* step.

9.3.1 Trackpy

With minor modifications, it was possible to update the Trackpy *2D Link* (described in section 7.3.1) to perform *3D Link*.

Algorithm

The Trackpy library implements the Crocker-Grier linking algorithm [26].

Evaluation

The speed is similar to the Trackpy *2D Link* for one camera, at 38 FPS. However, most of the tracklets were extremely short, resulting in about 9300 individual tracklet.

9.3.2 Nearest neighbor

Moving from 2D space to 3D space, the bubbles are naturally much sparser. As such, the 3D bubbles are far enough away from each other, that the nearest neighbor is undoubtedly the correct link.

Algorithm

For each bubble in a time frame, the link towards the next frame is chosen as follow:

1. Among the 3D bubbles in the next frame, find the one closest to the current position.
2. Evaluate the distance d between the current and selected position with respect to a threshold T:
    - If $d > T$, it's not plausible that the bubble has moved such distance in such short time: the original bubble is likely lost, and the selected one is probably a different bubble. As such, consider the tracklet to end at the current frame;
    - If instead $d \leq T$, the movement is plausible, therefore the two bubbles are linked.

Evaluation

The simplicity of this algorithm makes it extremely fast, with the potential to reach 5000 FPS. The main advantage is however the reconstruction quality: in the evaluation, only about 4800 tracklets were created.

9.4 Final choice

Differently from the other steps, all approaches of *3D Link* had sufficient speed. As such, the choice fell on the simple Nearest Neighbor, which had better quality.

Chapter 10

The Visualization step

The goal of the *Visualization* step is to display the reconstructed 3D particles on a 2D screen, in the most understandable way. Since no adequate tool was found on the Internet, the Unity renderer (explained in section 10.1) was initially developed as a versatile, offline tool. After it was finished, an update to the requirements demanded for a renderer able to display the reconstruction as it was being done: the Open3D renderer (described in section 10.2) was then added as an online alternative.

10.1 Unity renderer

Displayed in figure 10.1, the Unity renderer is the first visualizer developed in the scope of this thesis. It is an offline tool, meaning that it is able to show the data after the acquisition is fully processed. The pipeline produces as output two `.npz` files, containing the arrays `positions` and `validTracers`. These files can be directly loaded by the Unity scene setup to display their content.

A custom loader was necessary to transform the arrays from the `NumPy` format into C# arrays. Further modules are able to display such arrays in the 3d environment, using simple primitives as backbone. In particular, the bubbles are represented by spheres in the 3D environment, connected by small rods to display the linked trajectories.

The time evolution of the bubbles is visible with real time: the user is able to "play" the scene for inspecting the general behavior. The user also has control over the time speed, allowing to play back the experiment result in slow-motion, for better observing the fast-moving bubbles. Alternatively, it is possible for the user to advance or go back by a single frame at a time, allowing to better inspect what happened at the smallest scale of time.

For better understanding the 3D position of the bubbles, the observer is able to

move around the simulation. In particular, the user controls a "floating camera" that observes the scene from inside. It is possible both to rotate the camera around its axes, and to move it. The movement is possible in the direction where the camera is looking, and in the orthogonal, horizontal direction. This movement ability allows the user to inspect the reconstruction from whichever angle they prefer. Figure 10.2 displays the way the user can move the visualization camera.

The simulation proposes some more controls to tailor the visualization to the needs of the user. In particular, it is possible to customize at any time:

- How many frames of trailing trajectory to display: 0, N or all. This allows to concentrate on whatever is required at the moment: a specific time instant, a short-term evolution, or the full history.

- The size of the bubbles, that can be reduced up to make them disappear completely. This allows to either focus on the specific time instants where the frames were captured, or on the overall time evolution without focusing on the specific instants.

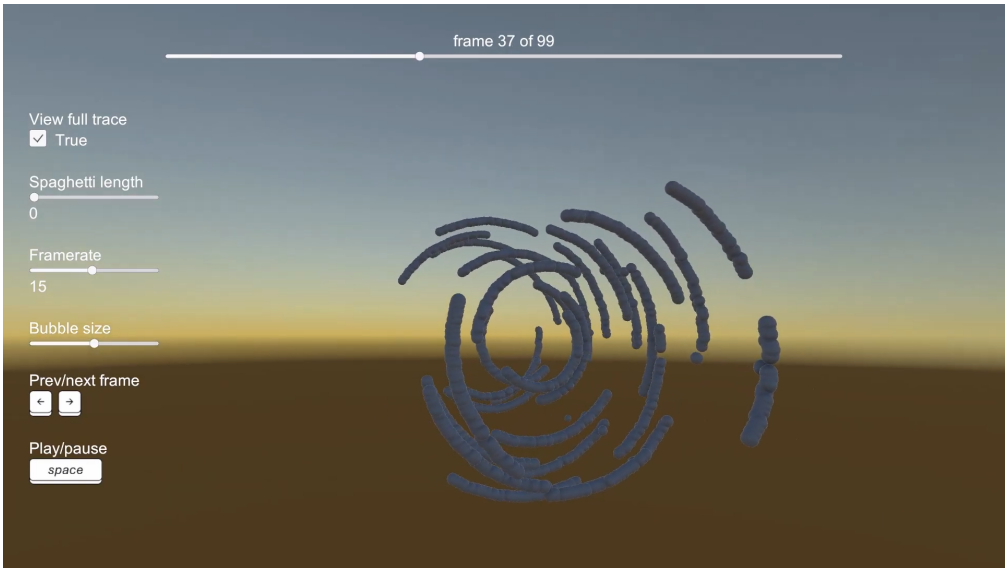


Figure 10.1: An example of bubble visualization using the Unity visualizer. Full video available at [32]

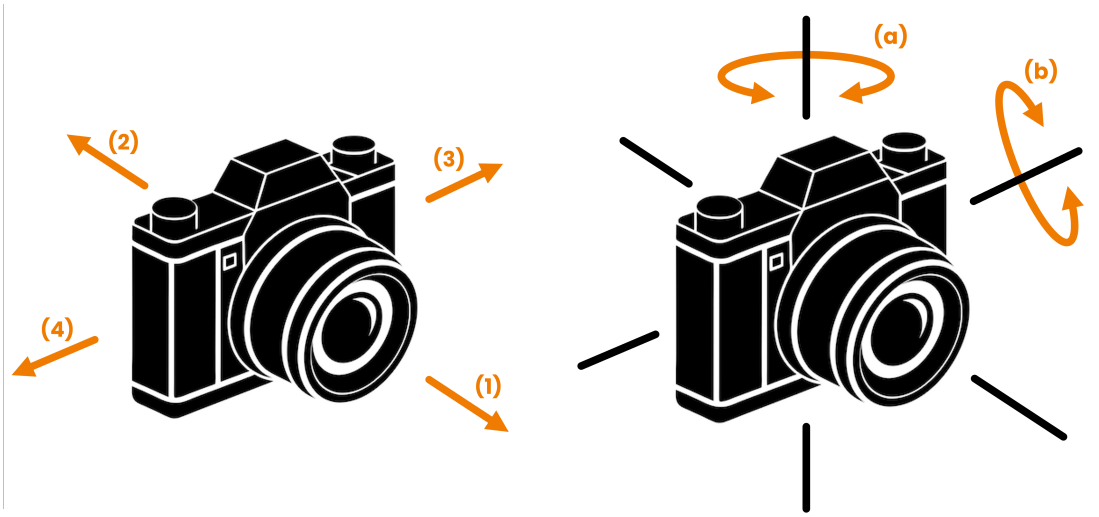


Figure 10.2: To the left, the four directions in which the user can move the visualization camera: (1) forward, (2) backwards, (3) left and (4) right, with respect of the current "looking-at" direction. To the right, the ways it can turn: (a) horizontally or (b) up and down

10.2 Open3D renderer

The Open3D renderer (a screenshot can be seen in figure 10.2) was later added, to fulfill the requirements of an online renderer, meaning a renderer that would update in real-time, adding the new bubbles as soon as they are processed.

This was added as a separate step in the pipeline, taking the data directly from the last step.

The user movement is the same as in the Unity renderer (schema available in figure 10.2). It may however be slightly jumpy, since it's running on an embedded device, which at the same time is executing the rest of the pipeline.

Differently from the Unity renderer, this one cannot use real time as representation of reconstruction time: instead, a color gradient is used, with blue representing the first and red the last time instants.

Another drawback of this visualizer is its volatile nature: displaying the data as soon as it is computed, it is not possible to view back the result once the visualizer is closed. In such cases, the Unity renderer can be used instead.

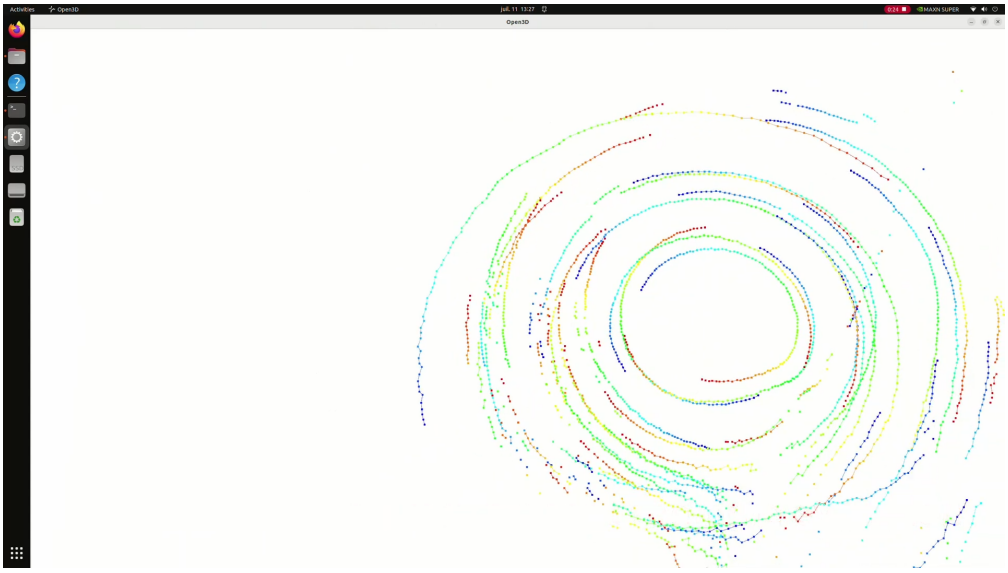


Figure 10.3: An example of bubble visualization using the Open3D visualizer. Full video available at [33]

Chapter 11

The full pipeline

11.1 Pipeline order

As described in chapter 5 and in figure 5.1 (reported here as figure 11.1 for convenience), the pipeline can be implemented either in the blue or in the orange order.

From the results of chapter 9, the *3D Link* (in the orange order) was quite faster than the *2D Link*: a faster step implies less computation resources used, which leaves more available for the most intensive tasks. The other benefit of the orange pipeline is that the linking is performed in 3D, where more information are available. In particular, it was noticed relatively often that the *3D Matching* step would reconstruct bubbles belonging to the same 2D tracklet at different depths, creating a sudden jump in the 3D trajectory, indicating an error. Instead, when the *3D Link* was used, the results were more coherent. As such, the orange order was chosen for the pipeline.

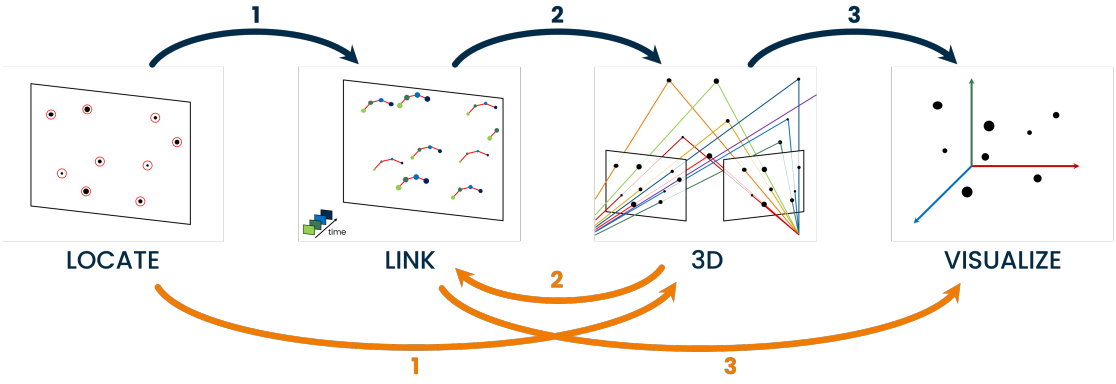


Figure 11.1: The two different orders in which the pipeline can be executed

11.2 Implementation

The pipeline is implemented on the operating system as a set of processes, to fully use all the cores of the CPU. Simpler threads would not be equivalent, since all Python threads are executed on the same CPU core. The communication among the processes is realized by means of shared memory locations, where the input/output arrays are stored in a shared way.

The different processes are organized as follows:

- the *Locate* step is performed by its own process, that:
  - loads the images either from the cameras or from file;
  - finds the bubbles using the `findContours` function;
  - launches and waits for the GPU computation of the moments.
- a process runs the *3D Matching* step, that:
  - waits until new data is available from the *Locate* step;
  - computes the first guess of matching;
  - computes the median displacement;
  - refines the first guess with the computed medians;
  - uses the matches to reconstruct 3D coordinates;
- the *Link* (*3D Link*) step is executed in another process, that:
  - waits until new data is available from the *3D Matching* step;
  - performs the linking;
- if enabled, the *Visualization* step is run by a separate process, that:
  - activates a new virtual environment, since `Open3D` requires a `NumPy` version not compatible with the rest;
  - runs the visualization script, that constantly checks for new values, displays them and responds to the user input.
- if enabled, a final debug process constantly updates the output in the terminal, writing the total number of frames processed by each step.

The different waits are realized as a loop that performs a 0.5s sleep until more data is available.

Chapter 12

Results

12.1 Quality evaluation

The main source of errors in the full pipeline is the *3D Matching* step, whose mistakes are then passed on to the *3D Link* step. The root cause to the *3D Matching* errors is the bubble density. In fact, if few bubbles are present in the scene, there will be only a handful of candidates near the epiline. This makes it trivial to choose the correct one, even with basic approaches. As the bubble density increases, however, the number of candidates close to the epiline augments, too. With that, the *3D Matching* task becomes harder and harder, reaching overwhelming levels also for a human at just 100 bubbles. With every matching error, the algorithm reconstructs some "random, isolated bubbles" that not only are wrong by themselves, but also act as missing points in the trajectory chain, splitting the full trajectory into three separate parts.

For measuring how good the algorithm behaves with different bubble densities, some synthetic datasets were created in Blender, and used as input for the pipeline. Such datasets were composed of 30 frames with the same format as the real ones: 960×960 images, representing white bubbles over a black background. The datasets were constructed to have a specific number of bubbles N, always visible by all the three cameras observing the scene. All the bubbles rotate in a clockwise direction, with the same tangential speed: this enables to avoid bubbles shadowing each other, and creates a regular pattern that can be recognized by eye. Given the specifications of the dataset, ideally the algorithm should be able to reconstruct exactly N 30-frames-long trajectories: more trajectories are index of reconstruction errors.

As previously stated, the main source of errors is the *3D Matching*. Errors at this stage position a bubble away from its correct location, splitting its trajectory in three parts. In particular, for the 30-frames dataset, a single error at frame f would split

the full trajectory into two shorter segments, with lengths f and $29 - f$, plus a single-frame tracklet, with only the erroneous bubble. As such, by analyzing the distribution of trajectory lengths, it is possible to evaluate the quality of the reconstruction.

Experiments were conducted with four different datasets, respectively with 50, 100, 250 and 1000 bubbles each. Initially, the graph in figure 12.1 was created, with the most intuitive content: the percentage of trajectories with length x, for every possible length x. This visualization was however misleading: for example, the 50-bubbles dataset had 29 full-length (30 frames) trajectories, which is more than half. However, the corresponding point on the graph had a value of about 30%. The cause of this inconsistency in the graph is the fact that splitting a theoretical trajectory into smaller ones would increase the total number of trajectories, thus reducing all the percentages. In other words, the correct reconstruction counts as 1, but a trajectory with an error in the middle counts as 3.

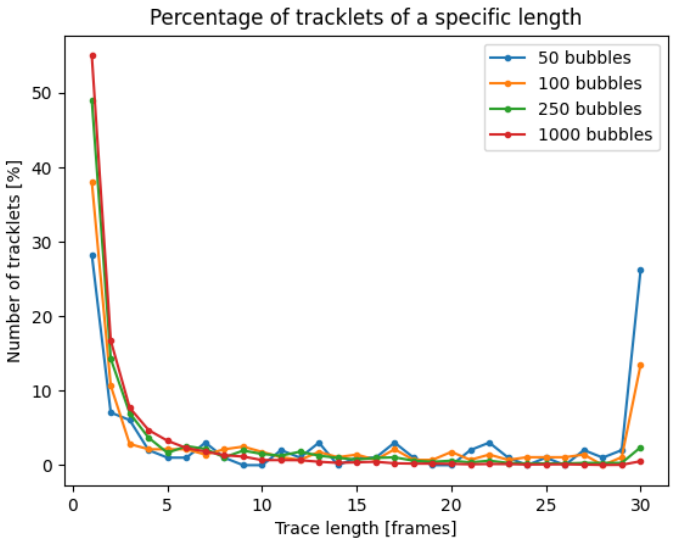


Figure 12.1: The distribution of trajectory lengths for the different datasets with varying number of bubbles: considering each trajectory as "one"

To make up for this error, a new graph was created, depicted in figure 12.2. It does not compute the average over the number of trajectories, but over the number of bubbles, which is not affected by the reconstruction quality. As such, the data points in this new graph are weighted over the trajectory length. For example, a 1-long trajectory would count as a "single unit", while a 30-long trajectory has a value of "30 units" for this computation. This enables to correctly showcase the overall quality of the reconstruction: in the 50-bubbles example considered before, now the value indicated by the graph for complete trajectories is about 60%, coherent with the measure of 29

fully reconstructed tracklets over the total 50 bubbles.

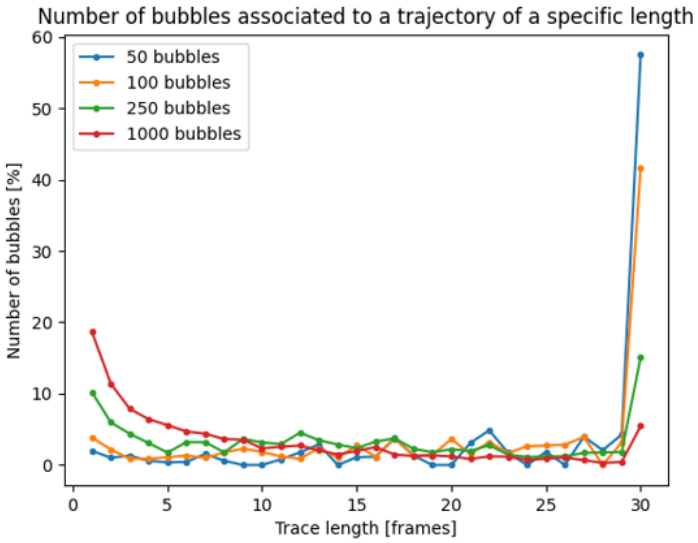


Figure 12.2: The distribution of trajectory lengths for the different datasets with varying number of bubbles: considering trajectories weighted on their length

In the ideal scenario, both graphs should have a flat value of 0% for all trajectory lengths, with a 100% spike at the value 30: whatever departs from this is index of errors. While it may seem counterintuitive, higher values (not 30) do not necessarily imply better results. Assume there is a reconstruction error in a tracklet: instead of having a full, 30-frames trajectory, there will be 3, with lengths f, 1 and $29 - f$, with f being the wrongly reconstructed frame. By changing where the mistake was (f), the two peaks in the graph will move, spacing from 1 and 29 for $f{=}0$, to 14 and 15 for $f{=}14$. There would however not be a real effect on the quality of the data: there will always be two coherent tracklets, separated by a missing point. As such, values smaller than 30 indicate the presence of errors, without correlation between quality and individual lengths.

As visible in figure 12.2, in the first two datasets most of the tracklets cover the full length, while the quality diminishes visibly with the other datasets. As such, the quality is considered good with observations of up to 100 bubbles.

12.2 Speed evaluation

12.2.1 Overall speed

The full pipeline took 42s to process a video composed of 1000 frames, thus resulting in a speed of 38 FPS, higher than what was required. To avoid waiting times due to the camera frame rate, these measurements were computed by using a video previously captured and saved as set of frames on the disk.

12.2.2 Speed of the pipeline steps

As visible in figure 12.3, the process is 2.3x as fast as the initial SMA-RTY implementation, which was already faster than the provided MATLAB script. Naturally, the last step to complete is the last one of the pipeline, the *Link*: however, as the next sections will show, the bottleneck is in the *Locate* step.

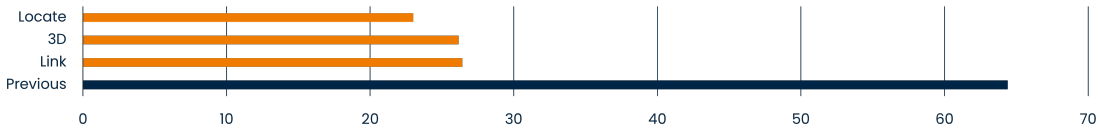


Figure 12.3: The time required (in seconds) by the different pipeline steps to process a 1000-frames video

Locate step

As shown in figure 12.4, the *Locate* step was always fully operational, without ever having to wait for data. This is obvious, since its input was already fully available at the start of the execution. It must however be noted that, as visible in figure 12.3, the *Locate* step did not finish much earlier than the others, indicating that it does not have a speed advantage over the other steps.

As visible in figure 12.4, the main bottleneck within this step is the time required for loading the images from the HDD to the RAM, to start processing them.

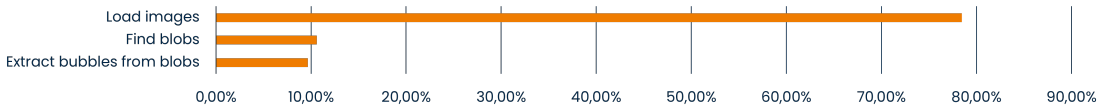


Figure 12.4: Distribution of how the *Locate* step spent its execution time

3D Matching step

The *3D Matching* step, as visible in figure 12.5, requires to spend a small amount of time waiting for the *Locate* data. This means that the *3D Matching* step is not the bottleneck. The waiting time is however not extensive, indicating that even small slowdowns may make this step the bottleneck, affecting the whole pipeline performance.

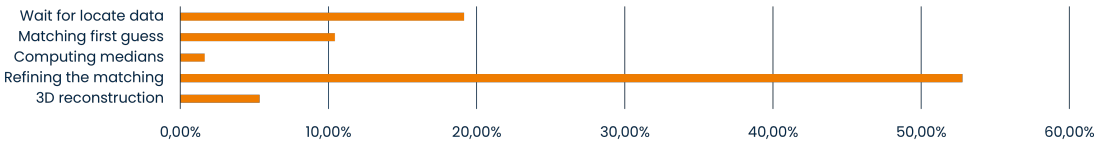


Figure 12.5: Distribution of how the *3D Matching* step spent its execution time

Link step

Figure 12.6 shows that most of the time the *Link* step is idling, waiting for new inputs to be processed. This indicates that it is far from being the bottleneck, and potentially more complex algorithm could be used instead, if they provided better results.

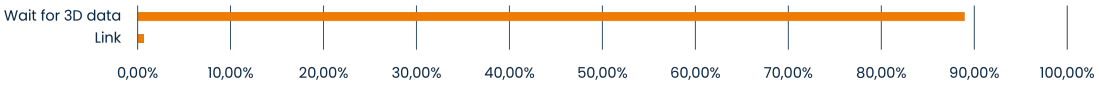


Figure 12.6: Distribution of how the *Link* step spent its execution time

12.2.3 Bottleneck evaluation

As highlighted by the previous sections, the *Locate* step is the current bottleneck of the system. Figure 12.7 confirms this, by showing in a graphical way the timing of the different steps: the various *3D Matching* frames require to wait some time for the previous *Locate* batches, and the *Link* frames are constantly waiting for the *Locate* output.

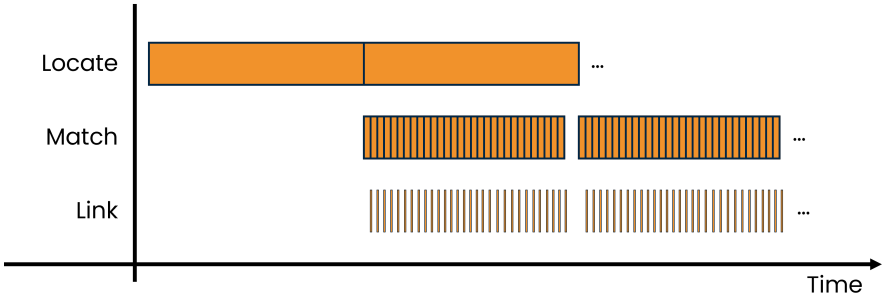
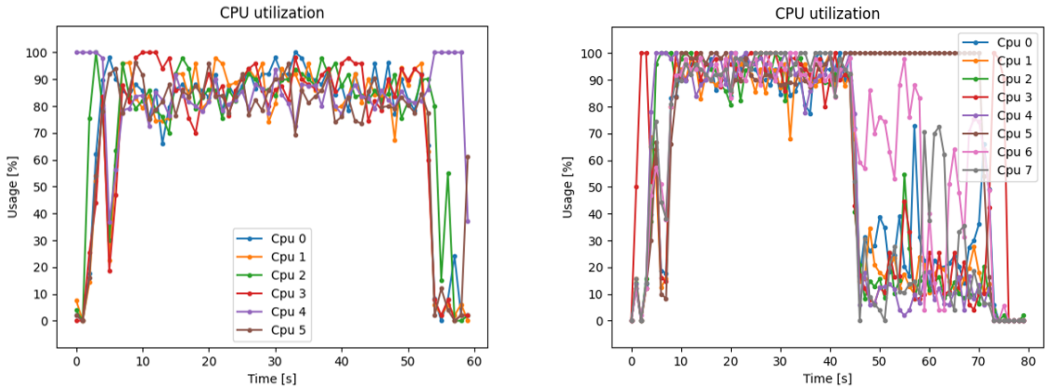


Figure 12.7: Schema (not to scale) of how the different pipeline stages are executed over time. The different rectangles indicate a batch of 20 frames per camera in the *Locate*, and single frames in the other steps

12.3 Resource usage

A further test was conducted to measure the CPU usage of the device running the pipeline. GPU usage was not measured, since the value would spike to 100% as soon as there was any task running on the GPU: the values would therefore not be indicative of whether other computations could be performed at the same time. The CPU results are shown in figure 12.8.

In the Jetson Orin Nano, all CPU cores spend the full processing time standing at almost 100%, indicating that all resources of the device are devolved to the task. On the Jetson AGX Xavier, the situation is slightly different: the most notable fact is that the bottleneck shifts from the *Locate* step to the *3D Matching*, and the overall processing speed is lower. This is visible in the graph as a sudden de-loading of the device: when all steps are running, the full device is in use; after the *Locate* finishes processing all the video, some cores reduce their usage. This may be an indication that, being a more premium device, the Xavier is faster than the Orin at transferring data to the RAM; however the fact of being older penalizes it on CPU computational speed.



(a) (b)

Figure 12.8: CPU usage per core, while running the pipeline on (a) the Jetson Orin Nano and (b) the Jetson AGX Xavier

Chapter 13

Conclusions

As discussed in the previous chapter, our solution is a pipeline able to track 3D bubbles moving in the air, at a speed up to 38 FPS. The output quality is excellent with up to 100 bubbles in the field of view, while it decreases with denser setups.

13.1 Future work

13.1.1 Improving the speed

While the speed is currently up to the requirements on the Jetson Orin Nano, an even faster system could run on less powerful and cheaper devices. As discussed in section 12.2, the main bottleneck for the speed is the loading of the images to the RAM of the processing device. This could be improved using dedicated hardware, or by spawning different processes to load the different images concurrently. For this last proposal, a study on the specific final device should be done, to be able to choose the right amount of processes. It would be crucial to balance the added parallelism with the cost of context switching, in case there are not enough cores to execute all the processes concurrently.

13.1.2 Improving the quality

Depending on the nature of the experiment, more than 100 bubbles may be required for the complete understanding of the phenomenon under observation. In order to increase the amount of bubbles reconstructed with good quality, the *3D Matching* step should be improved.

Since the "Epilines + brute force" approach is the one with the best results, and it is not far from the target speed, some more experiments could be performed to try

to accelerate it to an acceptable speed. This would improve the result, by having a better matching.

Another option would be to add more information to the data given to the *3D Matching* algorithm. An example would be if the bubbles could have different colors: if, for example, half the real bubbles were yellow and half were blue, the matching would only need to choose among half of the current candidates, the ones with the same color. In general, adding an extra piece of information that splits the real bubbles into N sets would on average enable to multiply by N the number of bubbles that can be reconstructed with good quality. Another alternative to the color could be the thermal information, if the temperature of the bubble changes during the time it spends in the air. For both these ideas, however, a different hardware setup should be constructed: RGB or thermal cameras would be required, and the images could not be simply binary, but would need more information.

Bibliography

[1] Arup Bhattacharya, Ali Ghahramani, and Ehsan Mousavi. «The effect of door opening on air-mixing in a positively pressurized room: Implications for operating room air management during the COVID outbreak.» In: *Journal of Building Engineering* 44 (2021), p. 102900. doi: https://doi.org/10.1016/j.jobe.2021.102900 (cit. on p. 3).

[2] Raja Singh and Anil Dewan. «Rethinking use of individual room air-conditioners in view of COVID 19.» In: *Creative space* 8.1 (2020), pp. 15–20. doi: https://doi.org/10.15415/cs.2020.81002 (cit. on p. 3).

[3] Inc. Sage Action. *Model 5 Console*. [Accessed: Sept. 2025]. url: https://sageactioninc.com/model-5-console/ (cit. on p. 3).

[4] Sage Action Inc. *SAI 1035 BFS*. [Accessed: Sept. 2025]. url: https://sageactioninc.com/1035-bubble-film-solution/ (cit. on p. 4).

[5] SMA-RTY. *SMA-RTY website*. [Accessed: Sept. 2025]. url: https://sma-rty.com/ (cit. on p. 4).

[6] OpenCV. *Camera calibration*. [Accessed: Sept. 2025]. url: https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html (cit. on p. 7).

[7] OpenCV. *Detection of ArUco Markers*. [Accessed: Sept. 2025]. url: https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html (cit. on p. 8).

[8] Kyle Simek. *Dissecting the Camera Matrix, Part 3: The Intrinsic Matrix*. [Accessed: Sept. 2025]. url: https://ksimek.github.io/2013/08/13/intrinsic/ (cit. on p. 9).

[9] NumPy team. *NumPy*. [Accessed: Sept. 2025]. url: https://numpy.org/ (cit. on p. 15).

[10] SciPy team. *SciPy*. [Accessed: Sept. 2025]. url: https://scipy.org/ (cit. on p. 15).

[11] Preferred Networks. *CuPy*. [Accessed: Sept. 2025]. url: https://cupy.dev/ (cit. on p. 15).

[12] OpenCV team. *OpenCV*. [Accessed: Sept. 2025]. url: https://opencv.org/ (cit. on p. 15).

[13] Anaconda. *Numba*. [Accessed: Sept. 2025]. url: https://numba.pydata.org/ (cit. on p. 15).

[14] Open3D. *Open3D: A Modern Library for 3D Data Processing*. [Accessed: Sept. 2025]. url: https://www.open3d.org/ (cit. on p. 15).

[15] Trackpy Contributors. *Trackpy: Fast, Flexible Particle-Tracking Toolkit*. [Accessed: Sept. 2025]. url: https://soft-matter.github.io/trackpy/v0.6.4/ (cit. on pp. 16, 24, 26, 44, 46, 55, 67).

[16] Ron Shnapp. «MyPTV: A Python Package for 3D Particle Tracking.» In: *Journal of Open Source Software* 7.75 (2022), p. 4398. doi: 10.21105/joss.04398. url: https://doi.org/10.21105/joss.04398 (cit. on pp. 16, 24, 30, 44, 47).

[17] Joris Heyman. «TracTrac: A fast multi-object tracking algorithm for motion estimation.» In: *Computers & Geosciences* 128 (2019), pp. 11–18. issn: 0098-3004. doi: https://doi.org/10.1016/j.cageo.2019.03.007. url: https://www.sciencedirect.com/science/article/pii/S0098300418310665 (cit. on pp. 16, 24, 35).

[18] Turbulence group at ENS Lyon. *4d-ptv*. [Accessed: Sept. 2025]. url: https://gitlab.in2p3.fr/turbulence/4d-ptv (cit. on pp. 16, 24, 38).

[19] PyTorch. *PyTorch*. [Accessed: Sept. 2025]. url: https://pytorch.org/ (cit. on p. 16).

[20] Unity Technologies. *Unity*. [Accessed: Sept. 2025]. url: https://unity.com/ (cit. on p. 17).

[21] NVIDIA Corporation. *NVIDIA Jetson Orin Nano Super Developer Kit*. [Accessed: Sept. 2025]. url: https://nvdam.widen.net/s/zkfqjmtds2/jetson-orin-datasheet-nano-developer-kit-3575392-r2 (cit. on p. 18).

[22] NVIDIA Corporation. *Jetson AGX Xavier and The New Era of Autonomous Machines*. [Accessed: Sept. 2025]. url: https://info.nvidia.com/rs/156-OFN-742/images/Jetson_AGX_Xavier_New_Era_Autonomous_Machines.pdf (cit. on p. 18).

[23] OpenCV team. *Hough Circle Transform*. [Accessed: Sept. 2025]. url: https://docs.opencv.org/3.4/d4/d70/tutorial_hough_circle.html (cit. on p. 36).

[24] OpenCV team. *Contours: Getting Started*. [Accessed: Sept. 2025]. url: https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html (cit. on p. 39).

[25] Francesco Risso. *2D Link - Original video*. [Accessed: Sept. 2025]. url: https://youtu.be/rR7RxGfCmbQ (cit. on p. 45).

[26] John C. Crocker and David G. Grier. «Methods of Digital Video Microscopy for Colloidal Studies.» In: *Journal of Colloid and Interface Science* 179.1 (1996), pp. 298–310. issn: 0021-9797. doi: https://doi.org/10.1006/jcis.1996.0217. url: https://www.sciencedirect.com/science/article/pii/S0021979796902179 (cit. on pp. 46, 68).

[27] Francesco Risso. *2D Link - Trackpy approach*. [Accessed: Sept. 2025]. url: https://youtu.be/JPp4lYocvmU (cit. on p. 46).

[28] Nicholas T Ouellette, Haitao Xu, and Eberhard Bodenschatz. «A quantitative study of three-dimensional Lagrangian particle tracking algorithms.» In: *Experiments in Fluids* 40.2 (2006), pp. 301–313 (cit. on p. 47).

[29] Francesco Risso. *2D Link - Kalman CPU approach*. [Accessed: Sept. 2025]. url: https://youtu.be/yskenLo0Cu8 (cit. on p. 50).

[30] Francesco Risso. *2D Link - Kalman GPU approach*. [Accessed: Sept. 2025]. url: https://youtu.be/SaCWMWsg73g (cit. on p. 51).

[31] Philipp Lindenberger, Paul-Edouard Sarlin, and Marc Pollefeys. «Lightglue: Local feature matching at light speed.» In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2023, pp. 17627–17638 (cit. on p. 57).

[32] Francesco Risso. *Unity renderer example*. [Accessed: Oct. 2025]. url: https://youtu.be/dAjsVrN2Zuk (cit. on p. 72).

[33] Francesco Risso. *Open3D renderer example*. [Accessed: Oct. 2025]. url: https://youtu.be/JQWxi5Jmnpc (cit. on p. 74).