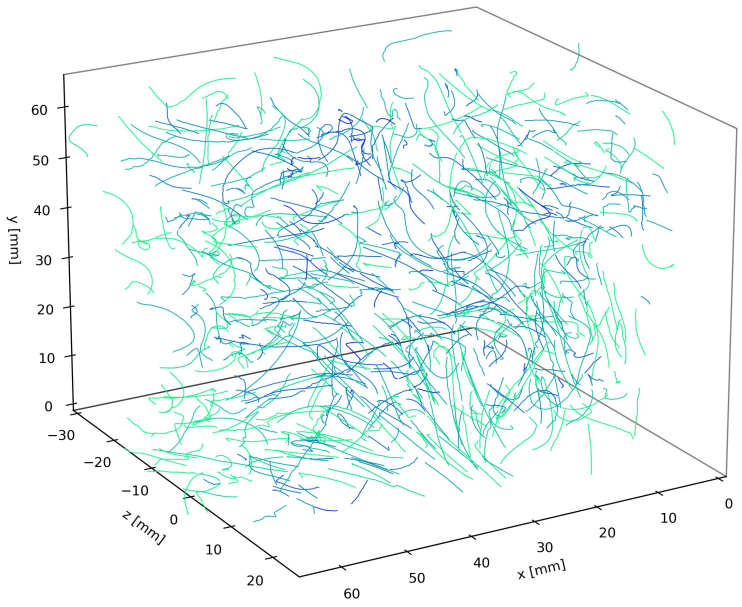
User Manual



Version: 0.6.5
Last updated: 17th October 2023
Github repository: https://github.com/ronshnapp/MyPTV
Get help & interact with our community: https://github.com/ronshnapp/MyPTV/discussions
Contact the developers: ronshnapp@gmail.com

Contents

1 Introduction

1.1 3D-PTV principles

The 3D-PTV method is used to measure the trajectories of particles in 3D space. It utilizes the principles of stereoscopic vision to reconstruct 3D positions of particles from images taken from several angles. A scheme of a typical 3D-PTV experiment using a four-camera system is shown in Fig. 1a. The "workhorse" behind the 3D-PTV method is the co-linearity condition, the 3D model. In principle, if we know what the position is and what is the orientation of the camera in 3D space (O' and θ in Fig. 1b), we can use the pin-hole camera model to relate the image space coordinates of a particle (η, ζ in Fig. 1b) to the ray of light connecting the imaging center and the particle. Then, if we have more than one camera, the particle will be located at the intersection of the two rays. Detailed information is given in [1, 2].

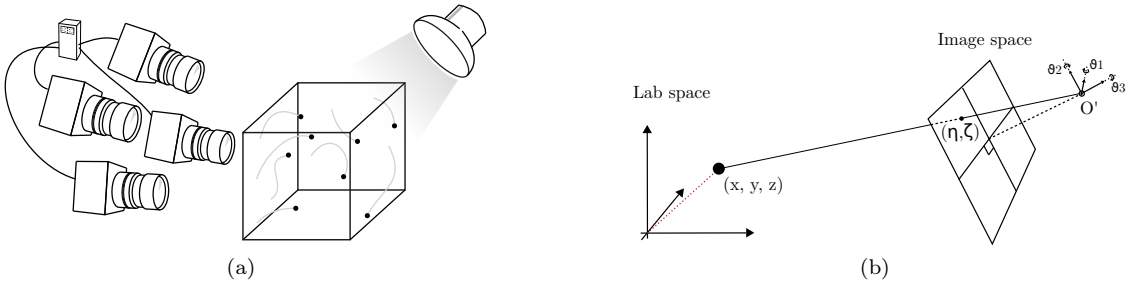


Figure 1: (a) A schematics of a 3D-PTV experiment. (b) A schematic description of the 3d model, the pin-hole camera model.

Once the experiment, namely data acquisition, is done, there are six intrinsic steps to follow in order to complete the analysis. The six steps are outlined in Fig. 2. In the camera calibration step, we use images of known calibration targets to estimate the position, orientation, and internal parameters of the cameras. In particle segmentation, we use image analysis to obtain the particles' image space coordinates (η, ζ). In the Particle matching step, we use the ray crossing principle to decide which particle image in each camera corresponds to the same physical particle, and triangulate their positions through stereo matching. In particle tracking, we connect the positions of particles in 3D space to form trajectories. In data conditioning, we might use smoothing and re-tracking algorithms to enhance our data quality according to some physical heuristics. Lastly, we can analyze the data to obtain information on the physics of the particles we are studying. The MyPTV package is meant to handle the first five of these steps.

The sections that follow outline the code used to handle the 3D-PTV method in MyPTV.

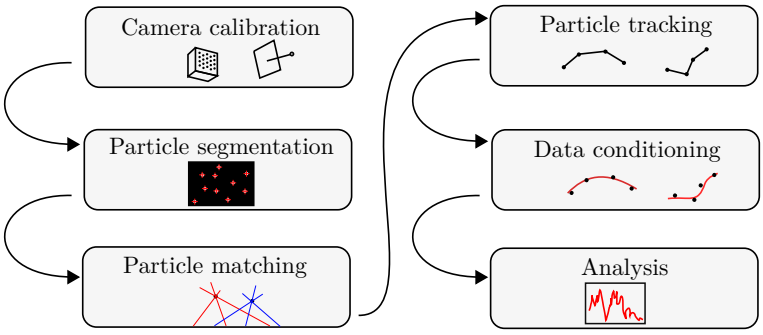


Figure 2: Basic steps in the analysis of PTV raw data into particle trajectories and scientific output. The first five steps are handled by MyPYV.

1.2 3D model

We use a pinhole camera model to transform the 2D particles seen in the camera images. In particular, the following expression relates the coordinates in the 2D image space coordinates and a direction vector pointing out from the camera imaging center:

$$\vec{r} - \vec{O} = \left(\begin{bmatrix} \eta + x_h \\ \zeta + y_h \\ f \end{bmatrix} + \vec{e}(\eta, \zeta) \right) \cdot \Big[R \Big] \tag{1}$$

where the description of the notations is given in Table 1. The matrix $[R] = [R_1] \cdot [R_2] \cdot [R_3]$ is the rotation matrix calculated with the components of the orientation vector, $\vec{\theta} = [\theta 1, \theta 2, \theta 3]$. In addition, the correction term $\vec{e}$ is assumed to be a quadratic polynomial of the image space coordinates:

$$\vec{e}(\eta, \zeta) = [E] \cdot P(\eta, \zeta) = \begin{bmatrix} E_{11} & E_{12} & E_{13} & E_{14} & E_{15} \\ E_{21} & E_{22} & E_{23} & E_{24} & E_{25} \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \eta \\ \zeta \\ \eta^2 \\ \zeta^2 \\ \eta\,\zeta \end{bmatrix} \tag{2}$$

where $[E]$ is a 3×5 matrix that holds the correction coefficients; the last row is filled with zeros because we do not attempt to correct f.

Table 1: Description of mathematical notation.

Symbol	Description
$\vec{r}$	Particle position in the lab space coordinates
$\vec{O}$	Position of a camera's imaging center
η, ζ	image space coordinates (pixels) of a particle
x_h, y_h	Correction to the camera's imaging center (in pixels)
f	The camera's principle distance divided by the pixel size
$\vec{e}(\eta, \zeta)$	A nonlinear correction term to compensate for image distortion and multimedia problems.
$[R]$	The rotation matrix which corresponds to the camera orientation vector.

1.3 MyPTV usage and structure

MyPTV is used to conduct the post-processing steps in which the experimental raw data (images) are transformed into particles' trajectories. The five steps include camera calibration, image segmentation, particle matching, particle tracking, and, data conditioning which is divided into smoothing and stitching of broken trajectories. This section outlines how these steps are to be followed and how MyPTV is structured to facilitate them. Section 2 gives instructions on how to efficiently use MyPTV.

At the current stage, MyPTV does not include a graphical user interface. Instead, the package includes a script that can be used to efficiently and conveniently run the various post-processing steps. Knowledge of Python is not required to operate MyPTV.

The internal structure of MyPTV is made of separate "modules", each of which deals with a particular step out of the list described above. Each module consists of a Python script file that includes one or several classes. Detailed information on each of these classes is given in Sections 3–9, and the code itself includes documentation to a level appropriate for development. In addition to the source code, MyPTV includes a *workflow* script that is used to operate the software in an organized fashion. Thus, MyPTV can be used either by using the workflow script or by directly evoking classes from the source code.

2 The Workflow: guidelines for a 3D-PTV experiment

MyPTV can be operated using the `workflow.py` script. The script is found under the *example* folder under the home directory of the MyPTV package. The workflow is used through command line given instructions and a file that outlines the parameters to be used in each step.

2.1 Preparation of an experiment folder

To begin post processing of the 3D-PTV results, prepare the following directory structure:

1. Make a new directory; in it, make a new directory called `Calibration` and place inside it the calibration images, and make another set of directories, one for each camera, and place inside it the particle images of each camera.
2. Locate the `example` under the root folder of the MyPTV package. This can be used as a template to the experiment directory made in the previous step.
3. Copy the files `workflow.py` and `params_file.yml` from the `example` folder into your experiment's folder.

2.1.1 The `workflow.py` script

`workflow.py` is a Python script used by the user to run the various steps of the post-processing. The script is used through the terminal or command line. To use MyPTV's various functions, go to the experiment's directory with:

```
cd \path\to\experiment\folder
```

and then, use the following syntax:

```
python workflow.py params_file.yml "command"
```

where `"command"` should be replaced with one of the following options, depending on the particular step of the post processing:

```
help
initial_calibration
final_calibration
calibration_with_particles
segmentation
matching
tracking
smoothing
stitching
2D_tracking
manual_matching
```

Each of these commands initiates one of MyPTVs functionalities, and a detailed explanation on them is found in Sections 2.2–2.8. Use `help` to print the available list of commands.

2.1.2 The `params_file.yml` file

The `params_file.yml` outlines all the parameters needed in order to run MyPTV using the workflow script. In every step conducted, go through the lists of parameters and make sure that their values are correct. The definitions per every step are tabulated in the sections below.

Table 2: The `params_file.yml` parameters for the calibration step. All paths to files are relative to the `workflow.py` script.

Parameter	Description
`camera_name`	the name of the camera to be calibrated; for example, `cam1`, etc.
`target_file`	path to the file that lists the calibration target's lab points
`calibration_image`	path of the calibration image
`resolution`	camera resolution; for example: 1280, 1024

2.2 Calibration guide

A good calibration is key to having success in your PTV experiment! Thus, follow this guide to calibrate your camera using MyPTV.

In the process called camera calibration, our goal is to determine what are the external ($\vec{O}$, $\vec{\theta}$) and internal (f, x_h, y_h, $[E]$) parameters of each of our cameras. This process is done by taking an image, a *calibration image*, of an object that has points with known coordinates marked on it - *calibration target*. Therefore, its important to make sure that for each experiment and each camera we have a calibration image ready, as seen, for example, in Fig. 7a.

Importantly, since our goal in calibration is to find the cameras' positions, once calibration images were taken in an experiment it is crucial that the cameras are not moved. Even minor vibrations can ruin an experiment. Also, the calibration target needs to be in the same medium in which the experiment will be performed to ensure that any refraction is corrected by the error terms (for example, if you are using a tank of water, place the calibration target inside the tank of water).

The calibration in MyPTV is performed in two steps: *initial calibration* and *final calibration*. The initial calibration is used to obtain a rough estimation of some of the camera's parameters using a small number of calibration points, as well as to extract the information for all the calibration points on the calibration target. In the final calibration we fine tune the camera's parameters in order to minimize the *calibration error*. The process is explained in detail in the following sub sections.

2.2.1 Preparing a `target_file`

The target file is a text file (tab separated values), in which each row shows the lab space coordinates of a calibration point. An example for the format is shown in Fig. 6b. If there is no target file in the experimental folder, this is the time to generate one. Prepare such a file and save it in the experiment's calibration folder.

2.2.2 Preparing the parameters in the `params_file`

As in any step of MyPTV, we make sure that the `params_file.yaml` in our experiment folder has the correct parameters, namely file names under the calibration tab; see Tab. 2. Each camera is calibrated separately, and so, for the calibration of each camera we need to change the file names in `params_file.yaml` appropriately (alternatively, one could have separate parameter files for each camera).

2.2.3 Initial calibration

To start the initial calibration, we start the *initial calibration GUI*. Using the workflow file in the terminal, use the following command:

```
python workflow.py params_file.yml initial_calibration
```

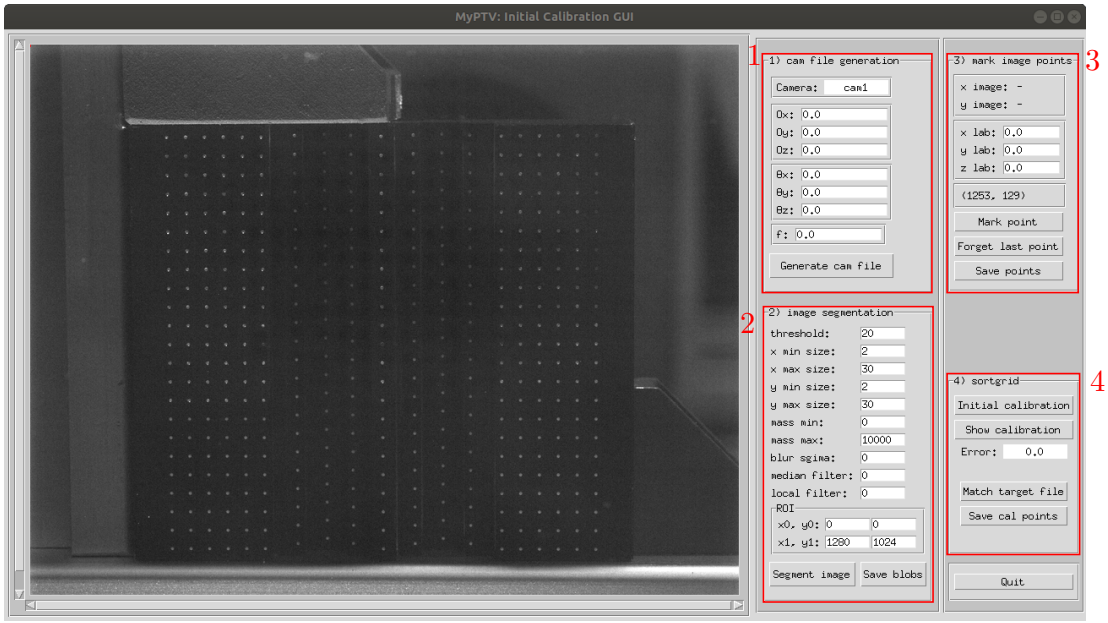


Figure 3: The initial calibration GUI.

which should open the a window like the one shown in Fig. 3. The left most panel shows the calibration image; we can zoom in and out by pressing the "+" and "-" keys.

The initial calibration is composed of four steps, and each one has a dedicated panel in the GUI. The steps should, in general, be followed along the order of the steps: $1 \rightarrow 4$.

Here are the details regarding each of the initial calibration steps:

1. `cam file generation`: Here we produce a text file that stores the camera parameters. We first have to give a first guess for the six different external parameters, $\vec{O}$ and $\vec{\theta}$, and the focal length, f. It's important to give a reasonably good initial guess, because the calibration problem often has local minima that make the convergence to solution difficult. To understand the how to choose the values, consult Section 2.2.4. When done, click the `Generate cam file` button.

2. `image segmentation`: In this step we use MyPTV to detect the points on the calibration image and to store their pixel coordinates in a file. Choose and tweak the segmentation parameters until all the calibration points are detected (or at least most of them if the images are not ideal). Detected points are indicated by green squares on the image. Also, a reference for the segmentation parameters can be found in Section 2.3. Once done, save the file using the `save` button.

3. `mark image points`: Here, we start by choosing six points (at least) of the calibration target. The points should not be coplanar and it is best to choose them such that there is some asymmetry to reduce the chance of false convergence. For each point chosen, we mark it by clicking on it on the image and we write its' lab space coordinates in the `x lab`, `y lab`, `z lab` fields, and then click the `Mark point` button. The marked points will be shown as blue crosses on the image. After marking all the chosen points, we save their coordinates using the `Save points` button.

4. `sortgrid`: In this last step, MyPTV tries to calibrate the camera using only the points we marked in step 3, and then to match the calibration points we segmented in step 2 to the points we gave in the target file. Thus, click the `initial calibration` button, make sure that the calibration error is low (aim for about 1.0), and click `show calibration`. If the results are good, click the `Match target file` button, and you should see the segmented

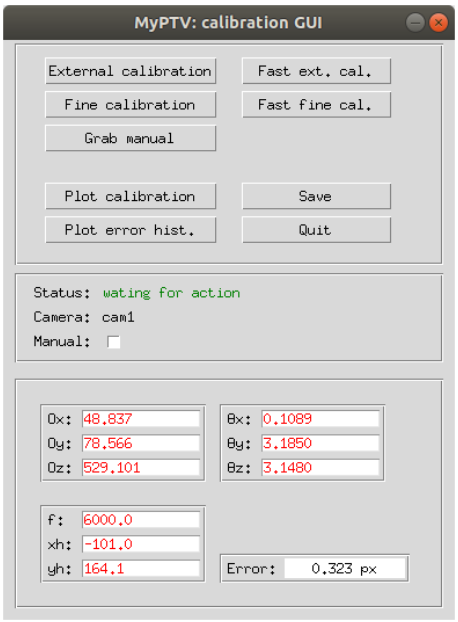


Figure 4: An image of the calibration GUI. The calibration GUI has several functionalities to run the minimization functions, plot the calibration error and the calibration particle's images, as well as change manually the camera parameters. See Tab. 3 for a description of the operation of each button.

points and the associated points from the target file paired by a red line on the image. Make sure that there are no errors in the pairing, and then save the results. If you detect errors or the initial calibration did not reach low enough error, it might be needed to generate an improved initial guess.

This concludes the initial calibration step.

2.2.4 Understanding camera external parameters

The external parameters of a camera in MyPTV include its position, its orientation, and its focal length, $\vec{O}$, $\vec{\theta}$, and f, respectively. In the calibration process, it is important to understand these parameters to be able to give a good initial guess and judge the results of the calibration procedure. Thus, let us explain how these parameters are defined.

Each camera is associated with an image space reference frame, shown in Fig. 5 as the x', y', and z' axes. The z' axis is normal to the imaging plane (the camera's sensor), while x', and y' are tangent to it; the system is right-handed.

The position parameter, $\vec{O}$, is defined as a vector pointing from the origin of the lab space coordinate system to the origin of the camera's reference frame. It is measured in lab-space units (e.g. millimeters). Therefore, one could estimate $\vec{O}$ directly by using a ruler or a measuring tape to measure where the camera is in the lab-space coordinates.

The orientation parameter, $\vec{\theta}$, is a vector of three angles that describe the rotation of the camera's reference frame with respect to the lab space frame, namely the camera's Euler angles. The first angle describes the rotation around the x' axis, the second describes rotation around y', and the third angle describes rotation around z'. Rules of thumb in trying to guess $\vec{\theta}$ are: 1) ask yourself "how should one rotate the lab's frame of reference so that it aligns with the camera's frame of reference?"; 2) we first rotate around the x' axis, then around y' and then around z'. Note: in many cases, the origin of the images is on the top left corner, so half a turn rotation around the z' axis is often needed. See Fig. 5 for an example.

Table 3: The various commands available in the calibration GUI (Fig. 4).

Parameter	Description
`External calibration`	Will minimize the calibration error by optimizing the external parameters of the camera using all the calibration points. This might require several runs to reach good results.
`Fast ext. cal.`	Will optimize the external camera parameters using a faster stochastic optimizer. This option is good for rapidly obtaining a reasonable calibration when there are a lot of calibration points.
`Fine calibration`	Will minimize the calibration error by optimizing the nonlinear error term of the camera parameters. This might require several runs to reach good results.
`Fast fine cal.`	Will optimize the nonlinear error terms of the camera parameters using a faster stochastic optimizer. This option is good for rapidly obtaining a reasonable calibration when there are a lot of calibration points.
`Grab manual`	This allows to manually change camera parameters. to do this, first tick the `Manual` checkbox. Then, manually change the values of the camera parameters. Finally, click the button and watch the effect on the calibration error.
`Plot calibration`	Will generate a plot of the calibration points projected on the camera space, and the known calibration points.
`Plot error hist.`	Will generate a histogram of the calibration errors for the various calibration points.
`Save`	Will save the results of the current session in the camera file.
`Quit`	Will quit the GUI.

The focal length parameter,f, is essentially an extension factor. In MyPTV f is measured in units of pixels. The focal length is normally more or less equal to the camera's lens focal length. For example, if one uses a camera with a 60mm lens in their experiment, and the size of one camera pixel is 10μm=0.01mm, then f would roughly be equal to 6000 (pixels). Potentially, one could also estimate f in an experiment: 1) place the camera in a known position facing a calibration target (as e.g. in Fig.4a) and take a picture 2) prepare a calibration points file for a few of the points on the calibration target; 3) preparing a camera file using the known position and orientation 4) insert some initial value for f; 5) using the `workflow.py` file to compare the distribution of the points using this value of f; 6) repeat steps 4 and 5 until a fair agreement is found.

2.2.5 Final calibration

In the final calibration we use all the calibration points to find the parameters of the camera. We do this by minimizing the calibration error, defined as the root mean square of the distances between the image-space calibration points positions and the projection of the lab-space points on the image plane. The minimization is solved numerically using the *final calibration gui*.

To start the final calibration gui, use the workflow script with the following command:

```
python workflow.py params_file.yml final_calibration
```

which should start the interface shown in Fig. 4. The description of each of the possible functionalities are given in Table 3. In principal, the lower the calibration error, the results of the measurements will be more reliable, and it will become possible to track a higher number of particles in each frame. A good calibration will have an error lower than about 0.5, or even 0.3 pixels. Once a good error value is achieved, the calibration process is complete.

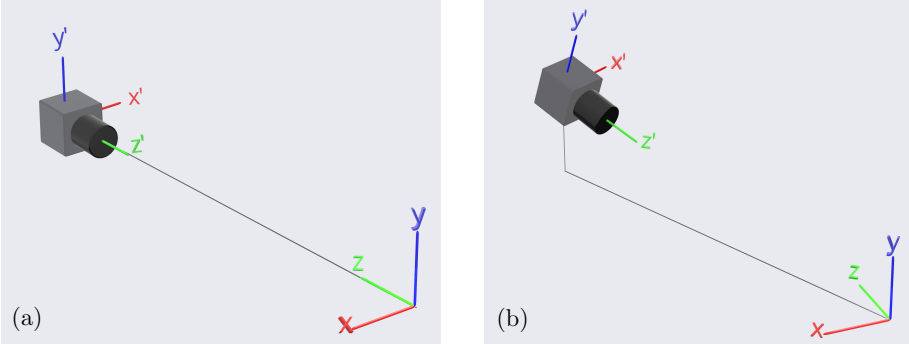
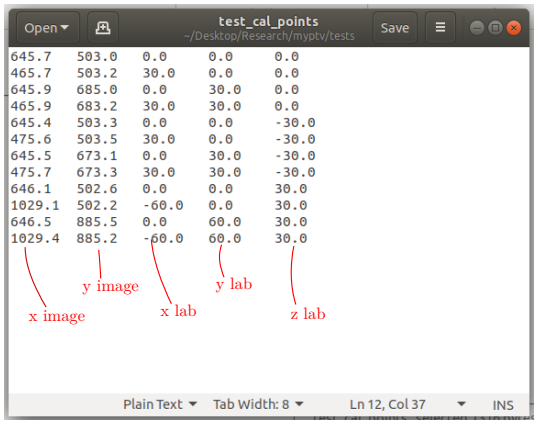
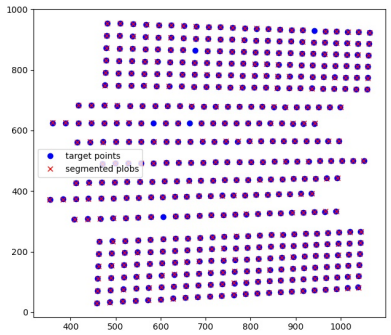


Figure 5: Two examples for camera setups. The (x, y, z) system represents the lab space, and the (x', y', z') represents the camera's system of reference. In (a) the camera's position is roughly $\vec{O} = (0, 0, 1)$, and an appropriate rotation vector could be $\vec{\theta} = (0, \pi, 0)$. In (b) the camera's position might be $\vec{O} = (1, 1, 1)$, and the orientation vector might approximately be $\vec{\theta} = (-\frac{1}{4}\pi, -\frac{3}{4}\pi, 0)$.



(a)

(b)

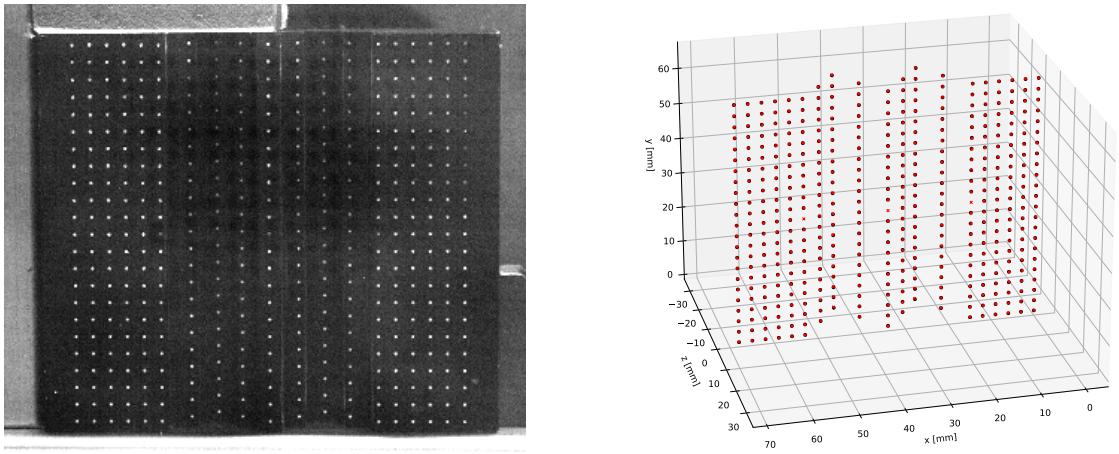


(c)

Figure 6: (a) Format of the target file, which lists the lab-space coordinates of the calibration target as tab-separated values. The order at which points are given is not relevant. (b) An example of a text file holding the calibration point data. (c) The results of matching a target file points (blue circles) and the segmented blobs (red crosses). Here, we know that the match is correct because each cross overlays a circle. In location where circles are not overlaid by a cross, the segmentation did not recognize a calibration target; having such missing points may slightly reduce the quality of the calibration but it is not critical for the target file matching.

2.2.6 Important notes

1. A "good" calibration will have a low error value - less than 0.5 pixels.
2. Optional validation - After all the cameras have been calibrated, it is also good practice to verify the calibration solution by stereo matching the calibration points. To do this, we use the `match_blob_files` (Section 6) to stereo match the files of the segmented calibration target points. We can then calculate the so-called *static calibration error* by computing the RMS of the distance between the triangulated calibration points and the real data from the target file. See Fig. 7b for an example. This validation step is not yet implemented in the workflow script.



(a) (b)

Figure 7: (a) An example of a calibration image. The points on the target have known lab space coordinates. Not also that the points are distributed over several plains (3 different z values in this case). (b) An example plot of a calibration error estimation. The red crosses represent the known positions of the calibration points given in the target file, and the black circles mark the positions of the stereo-matched segmented calibration points. In this case, the root mean square of the static calibration error was $84\mu m$.

2.2.7 How to perform calibration with several images (sometimes called multiplane calibration)

Calibrating the PTV cameras requires having calibration points spanning a full 3D volume, with points being positioned on different planes. Traditionally, this is done by constructing the calibration target with multiple planes (Fig. 7a). However, in some PTV systems another solution is taken such that the calibration target is moved to different locations and images of the target are taken there. In OpenPTV, this is called multiplane calibration. The following steps outline the procedure to follow when using such a multiplane calibration in MyPTV:

1. Place the calibration target in its first position; make sure you the location of the target and take an image of it. Then, move the target to another position and take an image of it. Repeat until the whole domain is covered by your calibration points, and save the sequence of images you just took.
2. Prepare a target file for each one of the images in the calibration sequence (see Seq. 2.2.1). Note that the target file for each image needs to hold only the information for the points that are seen in its particular image.
3. Perform an initial calibration for each of the calibration images, each with respect to its own target file. When you do so, make sure that the you are using a different "camera name" in

the params_file (for example, "cam1_1", "cam1_2", "cam1_3", ..., cam1_i).

4. After finishing the previous step, for each calibration image you should have a file called "XXXXX_i_cal_points" where "XXXXX_i" stands for the camera names you put in the params file.
5. Make a new empty file called as the base name of the previous files (the "XXXXX" part of the "XXXXX_i"); in the params_file, change the camera name to the same value ("XXXXX"). Inside the new file just generated, copy and paste the data from all the "XXXXX_i_cal_points" files.
6. Make a copy of one of the camera files, and change its name to "XXXXX". This will make sure you have a camera file from which we start the final calibration.
7. Run the final calibration as done in the regular calibration procedure.

2.3 Segmentation

2.3.1 How segmentation works

In the segmentation step, the program extracts the location of particles in the image. Segmentation in MyPTV is performed in one of two methods: *labeling*, or *dilation*, and includes several optional image processing steps.

1. In the labeling method, we first choose all pixels in an image whose grey value is higher than a given threshold. Then, such pixels that are touching each other are considered to be "blobs", so we group them together. For each blob, we calculate a *center of mass*, being the grey value weighted average of the blob, a *bounding box* size, and a *mass*, being the sum of grey values of the blobs' pixels.
2. In the dilation method, we are looking for pixels that their grey value is higher than that of all their neghbours within a box of a given size (`particle_size`), and thet their grey value is higher than a given threshold. After that, all the neighbouring pixels inside the particle size box are considered to be a blob. We then calculate the center of mass of the blob to achieve a better estimation of its position, thus redifining the blob. This process is iterated a maxinum of 3 times untill convergence is obtained. Finally, we calculate the same blob's *center of mass*, *bounding box* size, *mass*.

To improve the results of segmenting over non-ideal images, MyPTV can be set to apply image processing before the blob extraction. The filters that can be used are a Gaussian blur filter, a median noise removal filter, local mean subtraction filter, and a mask. The first three filters may increase the computational time, however they may be necessary depending on the experimental conditions, such as uneven illumination or noisy images. Examples of the results of using several combinations of the filters on a calibration image is shown in Fig. 8.

In this step we extract the particles' image-space coordinates from the images and save the coordinates in dedicated files. Use the `workflow.py` script with the `segmentation` command to start the process:

1. We perform the segmentation first on a single image by specifying `Number_of_images: 1`, and giving an image name with `single_image_name`. We then tune the segmentation parameters and the image processing filters used (e.g. `ROI`, `threshold`, `blur_sigma`, `method` etc; the full list is given in Table 4) to obtain optimal results; optionally repeat over several more images to ensure the results are satisfactory. To view the segmented particles over the image, use `plot_result: True`, and make sure the results are not saved by using `save_name: None`.
2. Once the parameters are determined, we set `Number_of_images: None`, and run the segmentation workflow again. This will then loop over all the images in the given folder (under `images_folder`), and save the results in a text file by setting `save_name: /file/name/to/use`

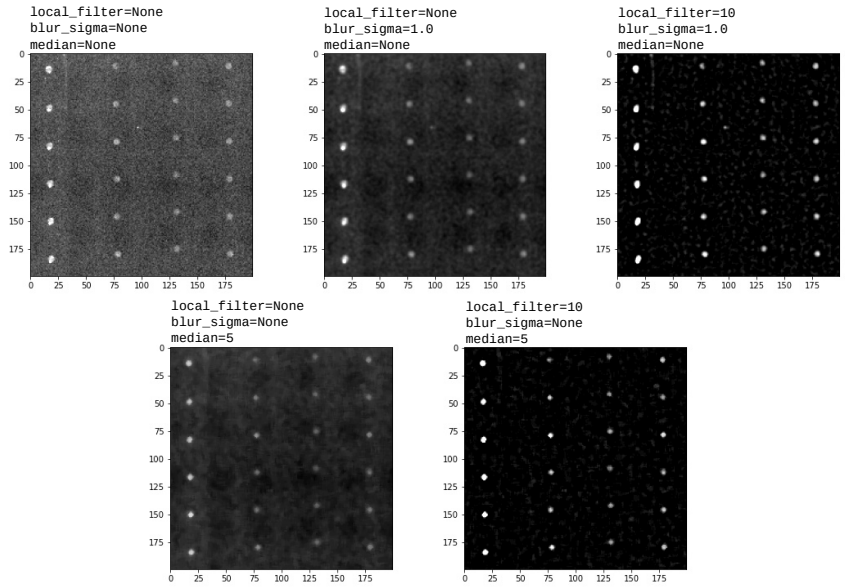


Figure 8: The effect oft various image filters on the segmentation.

2.4 Matching

In the matching step, particles images in the various images are used to stereo-locate (triangulate) the 3D positions of particles in the lab-space coordinates. As before, run first only on several images to find optimal parameter values, and only then iterate over all frames by setting `N_frames: None` and save the results.

Note: to perform easy stereo matching of points in images, use the manual matching GUI. See Section 2.10 and Fig. 10.

2.5 Tracking

The tracking step is used to link particle in the lab space coordinates in time, thus forming the 3D trajectories.

2.6 Calibration with particles

After some trajectories have been obtained, we can refine the calibration of cameras by using this obtained data and re-run the calibration procedure with the measured particles. For more detailes see Sec. 4.2. To start calibration with particles enter the command `calibration_with_particles` in the command line following the workflow command. Once this is done, a calibration sequence is initiated that uses the trajectory data.

2.7 Smoothing

The smoothing step is used to smooth trajectories in time and to calculate the velocity and acceleration of particles. This is done by fitting polynomials over sliding windows of the trajectory where velocities and acceleration are calculated through a direct differentiation of the polynomial (see [3]).

Table 4: The `params_file.yml` parameters for the **segmentation** step. All paths to files are relative to the `workflow.py` script.

Parameter	Description
`image_start`	The number of the image from which the loof begins. If this is `None` (default) the first image is set to 0.
`Number_of_images`	Number of images over which to do the segmentation. If it is 1, segmentation is done only on the image with the name given in `single_image_name`. If this is an integer, segmentation is performed on the first N images found in the directory `images_folder`. If this is `None`, segmentation is performed over all the images in the directory `images_folder`
`images_folder`	path to the folder containing the images
`single_image_name`	If `Number_of_images` is 1, the segmentation will be done on the image whose name is given here. If `Number_of_images` is any other value, this parameter is not used.
`image_extension`	extension of the images; for example, .tif
`plot_result`	if `False` will not plot the results; if `True` the results will be plotted but only if number of images is 1
`mask`	if this is 1.0, no mask is used; if this is set to a path to a file, the file will be used as a mask for the segmentation
`ROI`	region of interest; specify by indicating the xmin, xmax, ymin, ymax coordinates
`threshold`	the brightness value for which a point is considered a particle after a local mean subtraction is performed
`median`	the intger window size of a median noise removal filter; set to `None` in order to not apply the filter
`blur_sigma`	the float standard deviation of a Gaussian blur filter; set to `None` in order to not apply the filter
`local_filter`	the integer widow size of a local mean subtraction filter; set to `None` in order to not apply the filter
`min_xsize`	minimum particle size (pixels) in x direction
`min_ysize`	minimum particle size (pixels) in y direction
`min_mass`	minimum particle mass (mass is the sum of pixel grey values)
`max_xsize`	maximum particle size (pixels) in x direction
`max_ysize`	maximum particle size (pixels) in y direction
`max_mass`	maximum particle area (mass is the sum of pixel grey values)
`plot_result`	if `True` the segmented particles will be plotted over the processed image (after applying blur, local mean subtraction and masking); if `False` will not plot the results
`method`	ihe name of the method used for segmenting the blobs; can be either 'labeling' or 'dilation'
`particle_size`	in the 'dilation' method, this gives the diameter of blobs in the image (an integer number of pixels); this parameter is not used when method is set to 'labeling'
`save_name`	if `None` the results will not be saved in a file; if `path/to/file` will save the results in the given file name

Table 5: The `params_file.yml` parameters for the **matching** step. All paths to files are relative to the `workflow.py` script.

Parameter	Description
`blob_files`	names of files that hold the segmented particles for each camera, separated by commas; for example: `blobs_cam1, blobs_cam2, blobs_cam3`
`frame_start`	if `None` will start matching from the first available frame. If an integer it will start the matching from this given number.
`N_frames`	if `None` will match particles in all available frames; if an integer will match only particles in the first `N` frames.
`camera_names`	names of cameras used separated by commas; for example, `cam1, cam2, cam3`
`cam_resolution`	camera resolution; for example, `1280, 1024`
`ROI`	region of interest for the matching in lab space coordinates, and according to the format of `xmin, xmax, ymin, ymax, zmin, zmax`
`voxel_size`	the side length of voxels used in the *Ray Traversal* algorithm; note - too high values lead to long computation times and too low value result in matching errors and long computation times.
`max_blob_distance`	the distance particle usually undergo during each frame in image space coordinates (pixels)
`max_err`	maximum value of the RMS triangulation error in lab space coordinates
`save_name`	path name used for saving the results; if `None` the results are not saved

Table 6: The `params_file.yml` parameters for the **tracking** step. All paths to files are relative to the `workflow.py` script.

Parameter	Description
`particles_file_name`	path name of the file which holds the 3D coordinates of particles, namely the results of the matching step
`frame_start`	if `None` will start tracking from the first available frame. If an integer it will start the tracking from this given number.
`N_frames`	if `None` will iterate over particles in all frames; if an integer will only track particles in the first `N` frames of the particles file
`d_max`	the maximum translation in lab space coordinates
`dv_max`	the maximum allowable change in velocity in lab space coordinates per frame (e.g. mm/frame)
`mean_flow`	Adds a constant mean flow parameter that helps the tracking algorithm to track particles in flows with a constant drift; the format is [vx, vy, vz] in e.g. mm/frame.
`plot_candidate_graph`	If this is set to `True`, once tracking is done, it will show a graph of all the linking candidates. Red lines are links that were made and blue lines were candidates to be linked but they were rejected.
`save_name`	name of the file in which the results shall be saved; if `None` the results are not saved on the drive

2.8 Stitching

This step is used to re-track trajectories once velocities are known and to "stitch" broken trajectories following the algorithm of Ref. [4].

Table 7: The `params_file.yml` parameters for the calibration with particles step.

Parameter	Description
`traj_filename`	the name of the file containing the trajectories from which the calibration points are taken
`camera`	an instance of the camera we wish to try and re-calibrate
`cam_number`	int, >= 1; the number (index) of the camera to be calibrated. For example, to calibrate camera2, this should be set to 2
`blobs_fname`	The name of the file that contains the segmented particles' data
`min_traj_len`	Only trajectories longer then this number will be used in the calibration
`max_point_number`	the maximum number of points that shall be taken to re-calibrate the camera. Note that too many points (say above 1000) might lead to long calculation times

Table 8: The `params_file.yml` parameters for the smoothing step. All paths to files are relative to the `workflow.py` script.

Parameter	Description
`trajectory_file`	file name of the trajectories file, namely the results of the tracking step
`window_size`	window size of the sliding polynomial (must be an odd number); if a trajectory is shorter than `window_size` but longer than `min_traj_length`, then we use a window the size of the trajectory length
`polynom_order`	degree of the polynomial used in the smoothing
`min_traj_length`	the length of the shortest trajectories to be smoothed; has to be larger than `min_traj_length`
`repetitions`	the smoothing operation will be repeated this many times over each trajectory
`save_name`	file name to save the results; if `None` the results will not be save on the disk

Table 9: The `params_file.yml` parameters for the smoothing step. All paths to files are relative to the `workflow.py` script.

Parameter	Description
`trajectory_file`	name of the smoothed trajectory file used in the stitching
`max_time_separation`	maximum time separation over which to stitch broken trajectories
`max_distance`	maximum distance in the position-velocity space
`save_name`	file name to save the results; if `None` the results will not be save on the disk

2.9 2D tracking guide

In addition to the 3D particle tracking capabilities, MyPTV can also be used to track particles in 2D, namely using only one camera view of the particles. The main advantages of 2D experiments are that they are easier and since only one camera is needed, and that the tracking is easier in 2D because there are no particle occlusions. Note that 2D particle tracking in MyPTV doesn't simply track the particle's pixel locations, since this might give off errors due to camera aberrations or

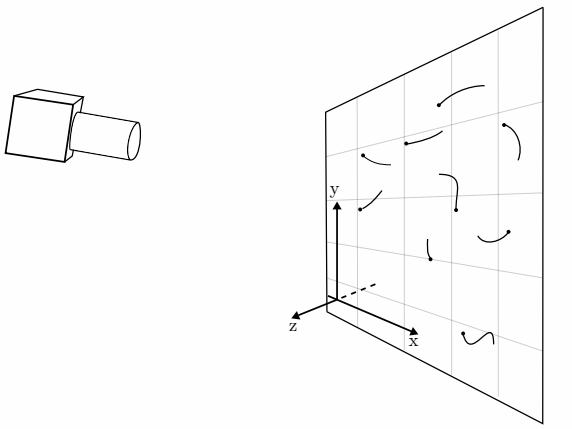


Figure 9: An example for experimental setup for 2D tracking. Note that the camera doesn't have to be perpendicular to the particles' plane and that the particles don't have to be at $z = 0$.

miss-alignment of the camera with the imaging plane. In practice this is done by making use of the optical model used in the 3D version, so this methodology is advantageous over simpler, pixel location, tracking approaches.

To track particles in 2D in MyPTV, we assume that all the particles shown in the image are located on a two-dimensional plane. In practice, this is done by setting a single z coordinate (given by the user) for all particles and then tracking their motion over the x and y coordinates only (see Fig. 9).

The 2D particle tracking procedure requires following these steps:

1. Prepare a calibration folder as described in Sec. 2.1
2. Go through camera calibration as shown in Sec. 2.2
3. Perform particle segmentation of the particle images, as described in Sec. 2.3
4. Track the particles' coordinates using the workflow script using the `2D_tracking` command (see Sec. 2.1.1). The various parameters of the `params_file` associated with 2D tracking are given in Tab. 10
5. If needed, follow smoothing (Sec. 2.7) and stitching (Sec. 2.8) of the results

2.10 Manual Matching GUI

In certain applications a user might wish to probe the 3D positions of elements in the images manually. Examples might be, to test the quality of the calibration on regions outside of the calibration target, or to measure certain geometrical features of objects in the images. For that purpose MyPTV features a graphical user interface dedicated to doing this easily. To run the GUI, use the `manual_matching` command of the `workflow.py` script.

To match a point we need to identify it from the different camera views. In the GUI we do this by clicking on the point we are after in the various images. See Fig. 10 for instructions.

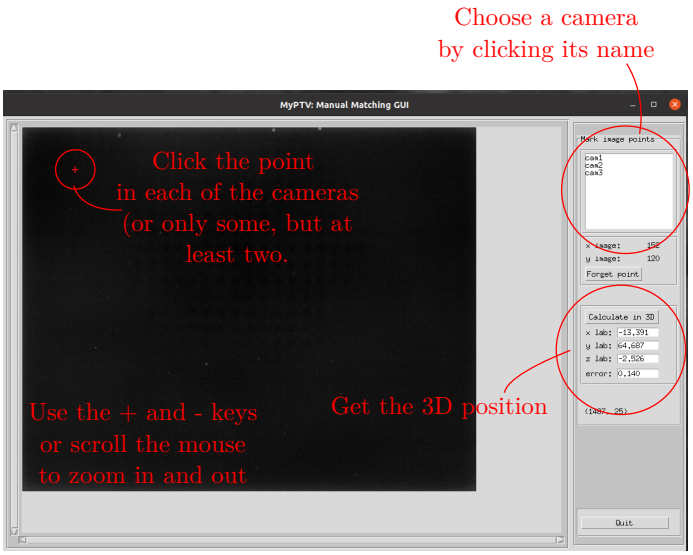


Figure 10: Instructions of how to use the manual matching GUI

Table 10: The `params_file.yml` parameters for the 2D tracking step. All paths to files are relative to the `workflow.py` script.

Parameter	Description
`blob_file`	The name of the file that contains the segmentation results
`frame_start`	The number of the first frame from which we want to perform 2D tracking
`N_frames`	The number of frames to be analyzed; If `None`, then all the frames will be analyzed
`camera_name`	the name of the camera from which the images were taken (must be a calibrated camera)
`camera_resolution`	A tuple representing the camera resolution, for example, (1280, 1024)
`z_particles`	The value of the z coordinate of the plane on which the particles are found, e.g. 0.0
`d_max`	The maximum allowable particle translation between frame in lab-coordinates
`dv_max`	The maximum allowable change of velocity for the particles in lab-space coordinates per frame (e.g. mm/frame)
`save_name`	The name of the file used to save the results

Table 11: The `params_file.yml` parameters for the manual matching operation. All paths to files are relative to the `workflow.py` script.

Parameter	Description
`cameras`	a list that holds the names of the cameras used; for example [cam1, cam2, cam3]. Note that the camera files need to be in the same folder where we are running the workflow script
`images`	the paths of the images from which matching should be performed.

3 Imaging module - `imaging_mod.py`

The imaging module is used to handle the translation from 2D image space coordinates to lab space coordinates and vice versa through the 3D model.

3.1 The `camera` object

An object that stores the camera external and internal parameters and handles the projections to and from image space and lab space. Inputs are:

1. `name` - string, name for the camera. This is the name used when saving and loading the camera parameters.
2. `resolution` - tuple (2), two integers for the camera number of pixels
3. `cal_points_fname` - string (optional), path to a file with calibration coordinates for the camera. The format fro the calibration point file is given in Section 3.3 (see Fig. 6a).

The important functionality options are:

1. `get_r(eta, zeta)` - Will solve eq. 1 for the orientation vector $\vec{b} = \vec{r} - \vec{O}$, given an input of pixel coordinates (η, ζ).
2. `projection(x)` - Will reverse solve equation (1) to find the image space coordinates (η, ζ), of an input 3D point, (`x`=$\vec{r}$).
3. `save(dir_path)` - Will save the camera parameters in a file called after the camera name in the given directory path, see Fig. 11.
4. `load(dir_path)` - Will load the camera parameters in a file called after the camera name in the given directory path, see Fig. 11.

After calibration we can save the camera parameters on the hard disc. The camera files have the structure shown in Fig. 11.

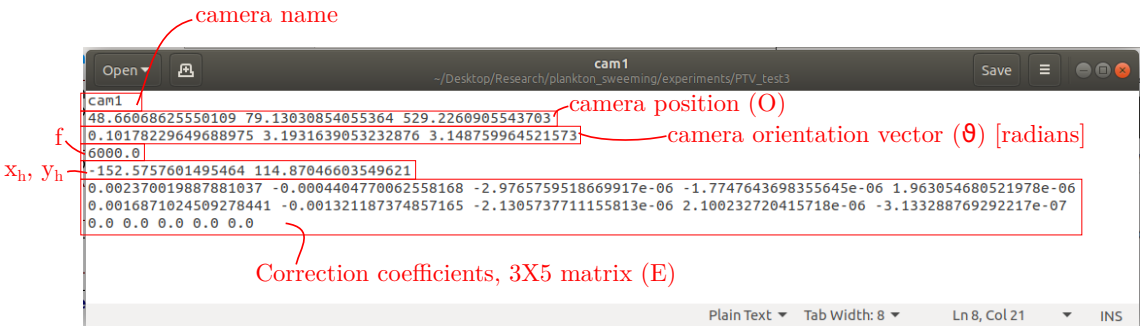


Figure 11: The structure of a camera file. The files are simple text files where each row corresponds to a specific parameter and the values in each row are separated by a white space.

3.2 The `imsys` object

An object that holds several camera instances and can be used to perform stereo-matching. The important functionalities are:

1. `stereo_match(coords, d_max)` - Takes as an input a dictionary with coordinates in image space from the several cameras and calculates the triangulation position.
   The coordinate dictionary has keys that are the camera number and the values which are the coordinates in each camera. d_max is maximum allowable distance for the triangulation.

3.3 The `Cal_image_coord` object

This is a class used for reading information given in the optional argument `cal_points_fname` of the `camera` class (Sec. 3.1). It is used internally and generally users will not have to deal with this. This class will read and interpret text files with tab separated valued, where the columns' meanings are: [x image space, y image space, x lab space, y lab space, z lab space], and each row is a single point of some known calibration target.

The input for this class is:

1. `fname` - String, the path to your calibration point file. The file is holds tab separated values with the meaning of: [x image, y image, x lab, y lab, z lab], see Fig. 6a.

4 Camera calibration - `calibrate_mod.py`

The `calibrate_mod.py` module, with the `calibrate` object, is used to find the camera calibration parameters. We calibrate each camera by taking an image of a *calibration target* - a body with markings of known coordinates in lab space - and search for the camera parameters that minimize the distance between the projection of the known points in image space and the image taken with the camera. In addition to that, after a calibration is made and trajectories have been obtained, the calibration might be refined by using the `calibrate_with_particles` object; this is particularly helpfull in cases where some of the cameras slightly moved in between the taking the calibration images and the recording of the tracer particles.

4.1 The `calibrate` object

Used to solve for the camera parameters given an input list of image space and lab space coordinates. The inputs are:

1. `camera` - An instance of a `camera` object which we would like to calibrate.
2. `lab_coords` - a list of lab space coordinates of some known calibration target.
3. `img_coords` - a list of image space coordinates that is ordered in accordance with the lab space coordinates.

The important functionalities are:

1. `searchCalibration(maxiter=5000, fix_f=True)` - When this is run, we use a nonlinear least squares search to find the camera parameters that minimize the cost function (item 3 below). This function is used to find the $\vec{O}$, $\vec{\theta}$, f, and x_h, y_h parameters (in case `fix_f=False`, it will not solve for f. `maxiter` is the maximum number of iterations allowed for the least squares search.
2. `fineCalibration(maxiter=500)` - This function will solve for the coefficients of the quadratic polynomial used for the nonlinear correction term ($[E]$).
3. `mean_squared_err` - This is our cost function, being the sum of distances between the image space coordinates and the projection of the given lab space coordinates.

To find an optimal calibration solution, we might need to run each function several times, and run the coarse and fine calibrations one after the other until a satisfactory solution is obtained. Once it is obtained, we should keep in mind to save the results using the `save` functionality of the `camera` object.

4.2 The `calibrate_with_particles` object

A class used to refine the calibration using particles data. In short, after the primary clibration is done, matching and tracking can be used to obtain trajectories from the experimental data. Assuming that this resulted in trajectories that were successfully measured, we can leverage the trajectories obtained to minimize further the calibration error. The refinement is done by assuming that the positions of particles along the resolved trajectories are "true" positions in lab-space, and thus, the blobs corresponding to these particles are used in a calibration sequence to minimize the calibration error. The assumption is that longer trajectories with low triangulation error are considered more reliable as compared to shorter trajectories, so we use only "long enough" trajectories in this process.

The inputs are:

1. `traj_filename` - the name of the file containing the trajectories from which the calibration points are taken.
2. `camera` - an instance of the camera we wish to try and re-calibrate
3. `cam_number` - int, >= 1; the number (index) of the camera to be calibrated. For example, to calibrate camera2, this should be set to 2.
4. `blobs_fname` - The name of the file that contains the segmented particles' data.
5. `min_traj_len` - Only trajectories longer then this number will be used in the calibration
6. `max_point_number` - the maximum number of points that shall be taken to re-calibrate the camera. Note that too many points (say above 1000) might lead to long calculation times.

The important functionalities are:

1. `get_calibrate_instance` - This function returns an instance of the `calibrate` object with the calibration points taken from the trajectory file. We then use this object to refine the calibration using the regular procedure (Sec. 4.1).

4.3 The `gui_final_cal.py` file

The `gui_final_cal.py` script is used to launch a `tkinter` based graphical user interface that is utilized to run various functionalities of the calibration process. The GUI is shown in Fig. 4, and the various buttons are explained in Tab. 3.

4.4 The `gui_initial_cal.py` file

The `gui_initial_cal.py` script is used to launch a `tkinter` based graphical user interface that is utilized to run various functionalities of the calibration process. The GUI is shown in Fig. 3 and it is explained in section 2.2.3.

5 Particle segmentation - `segmentation_mod.py`

This module handles the image analysis part of MyPTV, taking in raw camera images containing particles and output their image space coordinates. For the segmentation we first blur the image to remove salt and pepper noise, then we highlight particles using a local mean subtraction around each pixel, and then use a global threshold to mark foreground and pixels. Finally, the connected foreground pixels are considered to be particles, and we estimate the blob's center using a brightness weighted average of blob pixels.

5.1 The `particle_segmentation` object

Used to segment particles in a given image. This class is used internally to iterate over frames in a single folder by the `loop_segmentation` class. However, it is recommended to check the segmentation parameters manually using this `particle_segmentation` over several images in order to tune the particle searching parameters. The inputs are:

1. `image` - the image for segmentation
2. `threshold=10` - the global filter's threshold brightness value, so pixels with brightness higher than this number are considered foreground
3. `mask=1.0` - A mask matrix can be used to specify rigions of interest within the image
4. `sigma=None` - If float, this is taken as the standard deviation of a Gaussian blurring filter; if it is None, no bluring is performed
5. `median=None` - If integer this is taken as the window size for a median noise removal filter; if None, no median filter is performed
6. `local_filter=None` - Parameters for a local mean subtraction. If it's an interger it is taken as the window size; is it None, no mean subtraction is performed.
7. a bunch of threshold pixel sizes and mass in all directions and in area.
8. `method` - String. The name of the method used for segmentation (either 'labeling' or 'dilation', see Sec. 2.3). Default is 'labeling'.
9. `particle_size=3` - The particle size used in the dilation method.

The important funcitonalities are:

1. `get_blobs` - Will return a list of blob centers, their box size and their area.
2. `plot_blobs()` - Uses matplotlib to plot the results of the segmentation. A very useful functionality in the testing of segmentation parameters!
3. `save_results(fname)` - Will save the segmented particles in a text file. The file is arranged in six columns with the following attributes: (x center position, y center position, x size, y size, area, image number), see Fig. 12.

5.2 The `loop_segmentation` object

An object used for looping over images in a given directory to segment particles and save the results in a file. The inputs are nearly identical to those of `particle_segmentation`.

important functionalities are:

1. `segment_folder_images()` - Will loop over the images in the given directory and segment particles according to the given parameters
2. `save_results(fname)` - Will save the segmented particles in a text file. The file is arranged in six columns with the following attributes: (x center position, y center position, x size, y size, area, image number), see Fig. 12.

6 Particle matching - `particle_matching_mod.py`

The module used to link particles in the different images through stereo matching and estimating their 3D positions. One of the main issue in this process is that stereo matching all possible candidates is an NP hard problem, so to track numerous particles in each frame we have to choose which particles are likely to produce a 3D particle coordinate. Thus, particle matching in MyPTV

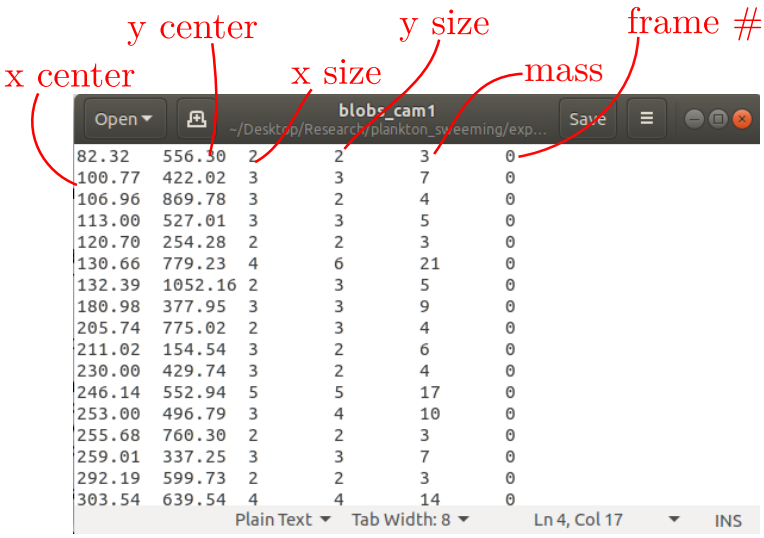


Figure 12: An example of a text file holding the segmentation resuls and the description of the different columns.

uses two algorithms in conjunction. First is a novel algorithm that uses 2D time tracking of blobs to deduce which candidates are more likely to produce traceable particles in 3D. Second is the Ray Traversal algorithm proposed in Ref [5], in which the lab space volume is divided to voxels and stereo matching is attempted for rays within each voxel. Using the two algorithms in conjunction was found to yield a 50% reduction in computational time and more traceable trajectories (45% more trajectories were found in a test).

6.1 The `match_blob_files` object

This is the object that we use to get triangulated particles results from the segmented blob files (a file as the one in Fig. 12 for each camera). For each frame it first runs the first algorithm using time information, and only then uses the Ray traversal algorithm on the blobs that were not successfully connected. The inputs are:

1. `blob_fnames` - a list of the (srting) file names containing the segmented blob data. The list has to be sorted according the order of cameras in the `img_system`.
2. `img_system` - an instance of the `img_system` class with the calibrated cameras.
3. `RIO` - A nested list of 3X2 elements. The first holds the minimum and maximum values of x coordinates, the second is same for y, and the third for z coordinates.
4. `voxel_size` - the side length of voxel cubes used in the ray traversal algorithm. Given in lab space coordinates (e.g. mm). Note - a too large voxel size will result in high computational times due a high number of candidates, while a too small voxel size might lead to erroneous intersection of rays, leading to matching errors. Thus, this parameter should be optimized.
5. `max_blob_dist` - the largest distance for which blobs are considered neighbours in the image space coordinates (namely, the largest permissible blob displacement in pixels).
6. `max_err=None` - Maximum acceptable uncertainty in particle position. If None, (defult), than no bound is used.
7. `reverse_eta_zeta=False` - Should be false if the eta and zeta coordinates need to be in reverse order so as to match the calibration. This may be needed if the calibration data points were given where the x and y coordinates are transposed (as happens, e.g., if using matplotlib.pyplot.imshow).

The important functionalities are:

1. `get_particles()` - Use this to match blobs into particles in 3D.
2. `save_results(fname)` - Save the results in a text file. The format has 4 + number of cameras columns separated by tabs: (x, y, z, [N columns corresponding to the blob number in each camera] , frame number, see Fig. 13).

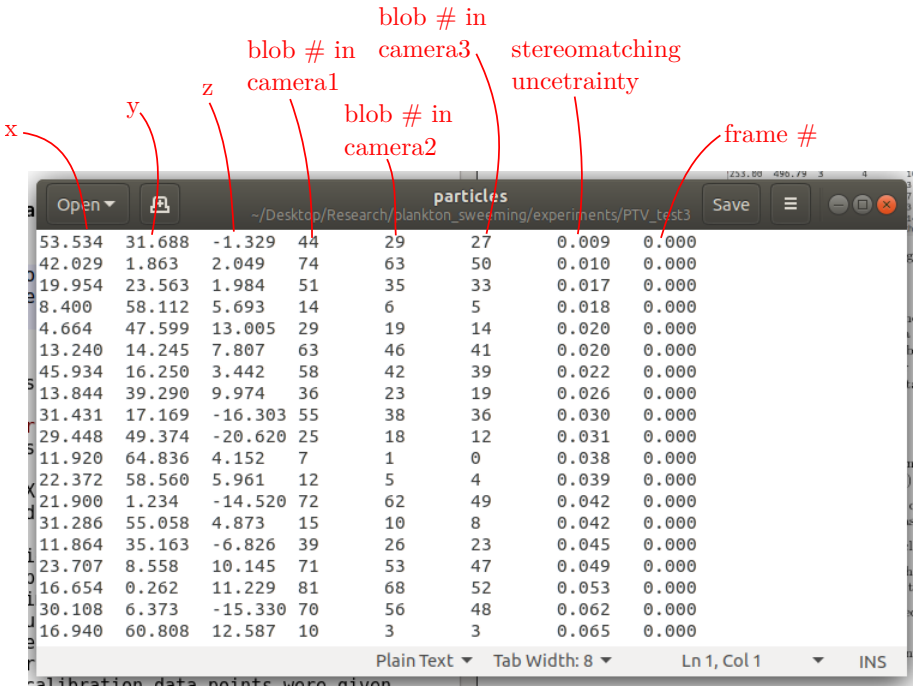


Figure 13: An example of a text file holding the triangulated particles' resuls and the description of the different columns. In this example there were three cameras. The blob number columns give the index of the blobs corresponding to any particle at the this specific frame number; a value of -1 in one of the rows means that no blob was used to stereo-match the particle in this row for this particular camera.

6.2 The `matching` object

This object is the "engine" used to match particles using the Ray Traversal algorithm. In practice we run the relevant functions: `get_voxel_dictionary()` → `list_candidates()` → `get_particles()`, and after that the results are held in the attribute `matched_particles`.

6.3 The `matching_using_time` object

This object performs the matching of blobs using the 2D tracking heuristic. In principle, it is given a list of blobs that were successfully used to form 3D particles in the previous frame. Then, for each of the given blobs it searches for nearest neighbours in the current frame, and stereo-matches those blobs that were found (using the `.triangulate_candidates()` method).

6.4 The `initiate_time_matching` object

This object is used to initiate the time searching algorithm on the first frame. It goes over the blobs at the first frame and searches for blobs that have nearest neighbours at the second frame.

Those that have neighbours are used in a first run of the Ray Traversal algorithm, thus they are given priority in the search.

7 Tracking in 3D - `tracking_mod.py`

This is the module that is used to track particles in 3D. There are currently three tracking methods implemented, nearest neighbour, two-frame, and four-frame, see Ref. [6]. Users are welcome to choose their preferred method and use it.

7.1 The `tracker_four_frames` object

An object used to perform tracking through the 4-frame best estimate method [6]. Input:

1. `fname` - a string name of a particle file (e.g. Fig. 13
2. `mean_flow=0.0` - either zero (default) of a numpy array of the mean flow vector, in units of the calibrations spatial units per frame (e.g. mm per frame). The mean flow is assumed not to change in space and time.
3. `d_max_=1e10` - maximum allowable translation between two frames for the nearest neighbour search, after subtracting the mean flow.
4. `dv_max_=1e10` - maximum allowable change in velocity for the two-frame velocity projection search. The radius around the projection is therefore dv_max/dt (where dt = 1 frame^{-1})
5. `store_candidates = False` - Boolean indicator. If it is true, then the tracker stores all the possible links that correspond to the tracking thresholds. In this case we can then plot a particle linking graph to analyze the tracking quality. If it is False, the tracker does not keep this information.

The important functionality is:

1. `track_all_frames()` - Will track particles through all the frames.
2. `return_connected_particles()` - Will retun the list of trajectories that were established.
3. `save_results(fname)` - Will save the results on the hard drive. The results are saved in a text file, where each row is a sample of a trajectory. The columns are specified as follows: [trajectory number, x, y, z, frame number], see Fig 14.
4. `plot_candidate_graph()` - If the candidate links were stored, this method will plot a graph that shows the links that were made and the candidate links that could have been made but were eventually not made due to duplicate links for the same particle.

7.2 The `tracker_two_frames` object

An object used for tracking through the 2-frame method. The description is the same as in Section 7.1

7.3 The `tracker_nearest_neighbour` object

An object used for tracking through the nearest neighbour method. The description is the same as in Section 7.1

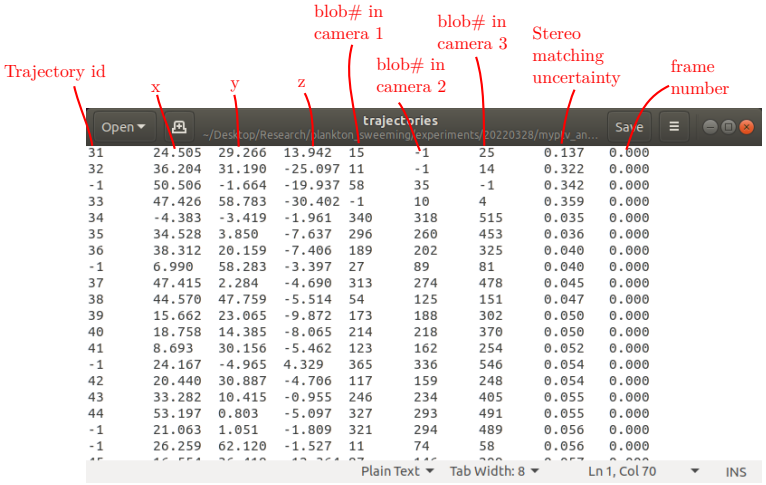


Figure 14: Example of a trajectory file and the column definitions. For Trajectory id being a non-negative integer, rows with the same Trajectory id correspond to the same trajectory; rows with Trajectory id being -1 are samples that could not be linked with the given tracking parameters.

8 Trajectory smoothing - `traj_smoothing_mod.py`

This module is used to smooth trajectories and to calculate the velocity and acceleration of the particles. For the smoothing we are using the polynomial fitting method proposed and used in Refs. [3, 7]. In short, each component of the particle's position is fitted with a series of polynomials with a sliding window of fixed and the derivatives are calculated by analytically differentiating the polynomial. The end result is a new file with smoothed trajectories. However note that we smooth and calculate velocities and accelerations only for trajectories longer than the window size for the smoothing (a user decided parameter).

8.1 The `smooth_trajectories` object

A class used to smooth trajectories in a list of trajectories. Due to the smoothing we also calculate the velocity and acceleration of the trajectories. The input trajectory list structure is the same as the files produced by the classes in `tracking_mod.py`.

Note - only trajectories whose length is larger than the window size will be smoothed and saved. Shorter trajectories are saved with zero velocity and accelerations.

The inputs are:

1. `traj_list` - a list of samples organized as trajectories. This should have the same data structure used in the saving function of the tracking algorithms (see Section 7.1).
2. `window` - The window size used in the sliding polynomial fitting.
3. `polyorder` - The order of the polynomial used in the fitting.

The important functionality is:

1. `smooth()` - performs the actual smoothing
2. `save_results(fname)` - Saves the results on the hard drive using the given (string) file name. The resulting file is a text file such that each row is a sample of a trajectory, and with 11 columns. The columns have the following meaning: [traj number, x, y, z, v_x, v_y, v_z, a_x, a_y, a_z, frame number], where v_i and a_i denote components of the velocity and acceleration vectors respectively, see Fig. 15.

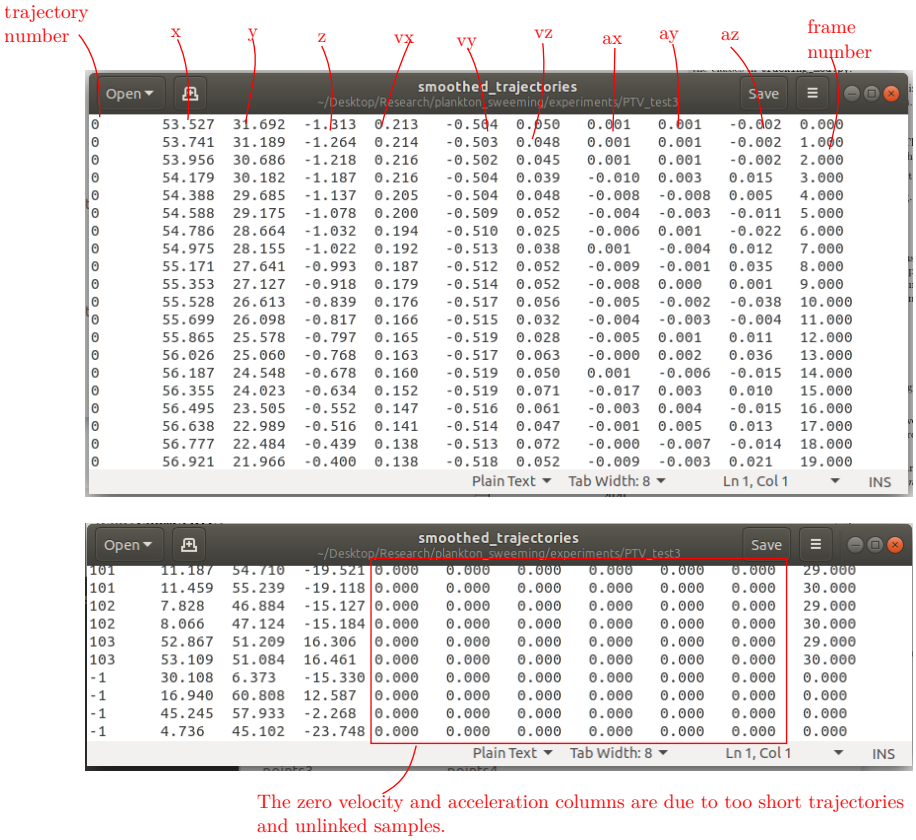


Figure 15: Example file holding the results of smoothed trajectories, and the description for each column. Note also the unsmoothed samples at the bottom of the file.

9 Trajectory stitching - `traj_stitching_mod.py`

This module applies the algorithm by Ref. [4] to connect trajectories that were broken along the process by tracking the trajectories again in the position–velocity space. We also extend this by interpolating the missing samples using a 3rd order polynomial, that is fitted to the existing 4 data points at the tips of the broken trajectories. This module is applied after the trajectory smoothing step.

9.1 The `traj_stitching` object

This object performs the stitching process. Inputs:

1. `traj_list` - the list of smoothed trajectory, given as a Numpy array of shape (N,11), where N is the number of samples. The format is the same as the format generated after the smoothing process.
2. `Ts` - The maximum number of broken samples allowed in the connection.
3. `dm` - The maximum distance between the trajectory for which connections are made.

The important functionalities are:

1. `stitch_trajectories()` - Will search for candidates and stitch the best fitting candidates. Run this function to perform the stitching. After running the new trajectory list is held as the attribute `new_traj_list`.

2. `save_results(fname)` - Will save the stitched trajectories in a text file with a given file name. The format for the saved file is the same as the one used in the smoothing trajectories (Fig. 15).

References

[1] M. Virant and T. Dracos. 3D PTV and its application on Lagrangian motion. *Measurement Science and Technology*, 8:1552–1593, 1997.

[2] H. G. Maas, D. Gruen, and D. Papantoniou. Particle tracking velocimetry in three-dimensional flows Part I: Photogrammetric determination of particle coordinates. *Experiments in Fluid*, 15:133–146, 1993.

[3] B. Lüthi, A. Tsinober, and W. Kinzelbach. Lagrangian measurement of vorticity dynamics in turbulent flow. *Journal of Fluid mechanics*, 528:87–118, 2005.

[4] H. Xu. Tracking Lagrangian trajectories in position–velocity space. *Measurement Science and Technology*, 19(7):075105, 2008.

[5] M. Bourgoin and S. G. Huisman. Using ray-traversal for 3D particle matching in the context of particle tracking velocimetry in fluid mechanics. *Review of scientific instruments*, 91(8):085105, 2020.

[6] N. T. Ouellette, H. Xu, and E. Bodenschatz. A quantitative study of three-dimensional Lagrangian particle tracking algorithms. *Experiments in Fluids*, 40(2):301–313, 2006.

[7] R. Shnapp, E. Shapira, D. Peri, Y. Bohbot-Raviv, E. Fattal, and A. Liberzon. Extended 3D-PTV for direct measurements of Lagrangian statistics of canopy turbulence in a wind tunnel. *Scientific reports*, 9(1):1–13, 2019.