MyPTV user manual

Ron Shnapp
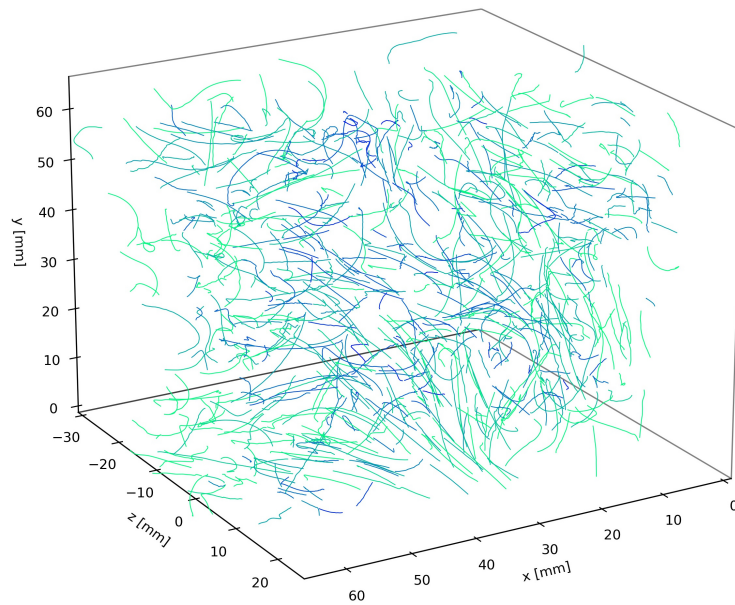
Contents

1 Introduction

1.1 3D-PTV principles

The 3D-PTV method is used to measure trajectories of particles in 3D space. It utilities the principles of stereoscopic vision in order 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 "work horse" behind the 3D-PTV method is the co-linearity condition, the 3D model. In principle, if we know what is the position 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].
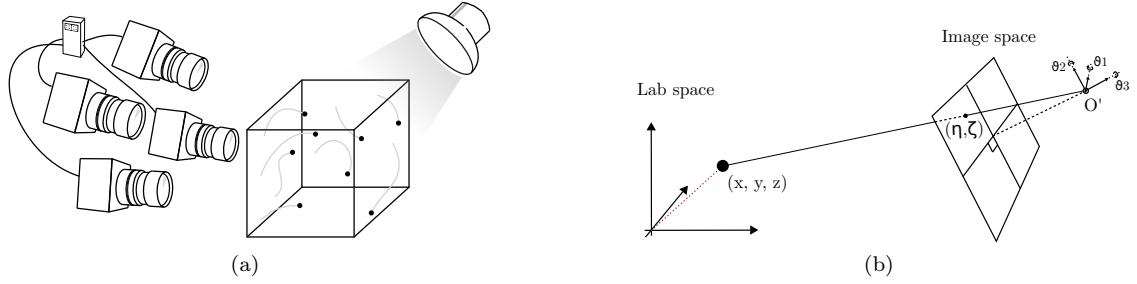
Figure 1: (a) A schematics of a 3D-PTV experiment. (b) A schematic description of the 3d model, the pin-hole camera model.

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 Camera calibration, 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 of the cameras correspond 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 the quality of our data 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

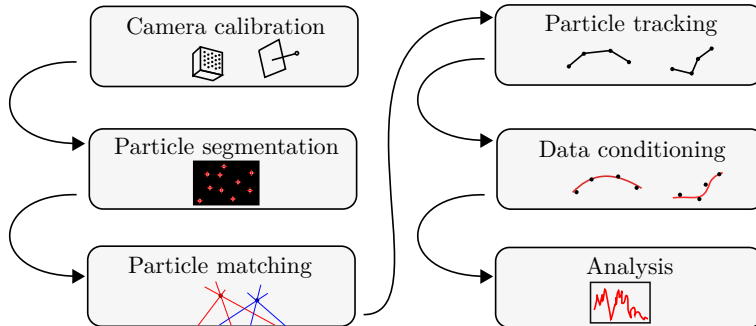To transform the 2D particles seen in camera images we use a pinhole camera model. In particular, the following expression relates the coordinates in the 2D image space coordinates and a direction vector pointing out from the camera imaging center:

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

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 steps in the post processing. Knowledge in Python is not required to operate MyPTV.

Each of the steps described above is dealt with in MyPTV as a separate "module". Each module consists of a Python script file which 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 either be used either by using the workflow script, or by directly evoking classes from the source code.

2 The Workflow

Operation of MyPTV can be facilitated 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, please follow these steps:

1. Prepare a new directory; place inside of it the experiment data: the calibration images in a directory called `Calibration` and the particle images under separate folders named after the name 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 `workfolw.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 experiment. It is used through the terminal or commend line. To use the various functions, use the following syntax:

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

```
calibration
match_target_file
segmentation
matching
tracking
smoothing
stitching
```

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. Thus, in every step conducted go through the lists of parameters and make sure that their values are correct. The definitions are tabulated per every step in the sections below.

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.

Camera calibration consists of several steps in which we determine the external $(\vec{O}, \vec{\theta})$ and internal $(f, x_h, y_h, [E])$ parameters of our camera system. Each camera in MyPTV is calibrated separately using a *calibration image* - an image of an object with marked points on it whose coordinates are
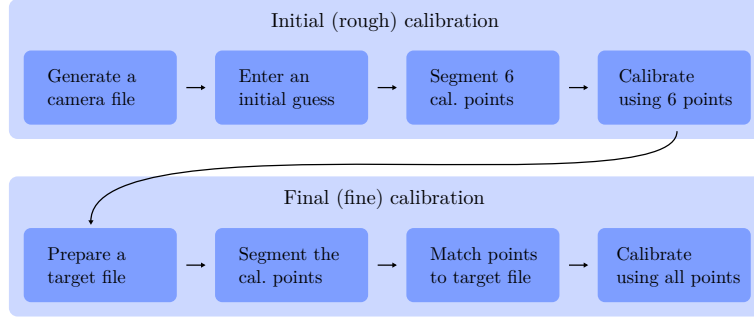
Figure 3: A block diagram representing the steps to follow in the calibration of each camera using MyPTV.

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
calibration_points_file	path to the file that lists the correlated lab space and image space coordinates
target_file	path to the file that lists the calibration target's lab points
segmented_points_file	path of the file with segmented calibration points
calibration_image	path of the calibration image
resolution	camera resolution; for example: 1280, 1024

known to us, a *calibration target*. Therefore, to calibrate a camera, we make sure that we have a calibration image ready, as seen, for example, in Fig. 9.

Importantly, since the goal is finding the cameras' positions, once a calibration image was taken in an experiment it is critical that the cameras are not moved. 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 (e.g. if you are using a tank of water, place the calibration target inside the tank of water).

A schematic block diageam for the calibration process of each camera is given in Fig. 3. The calibration is divided into two sub-processes: 1) initial calibration, in which we use only 6 calibration points to obtain an approximate calibration estimation, and, 2) final calibration, in which we use all the calibration points on the calibration target to improve the accuracy of the initial calibration. The various steps of the process are performed in MyPTV through the workflow script, and detailed instructions are given below.

2.2.1 Preparing the parameters in the `params_file`

As in any step of MyPTV, we first 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).

2.2.2 Initial calibraiton

For the initial calibration, follow these steps:

1. *Generate a camer file:* The camera file holds the camera parameters (see Fig. 11). An empty camera file can be generated automatically by the workflow script when we start the

calibration squence in a directory that doesn't havea camera file in it (by running `python workflow.py params_file.yml calibration` in the command line).

2. *Enter an initial guess:* Open the camera file just gnerated and insert an initial guess for the camrea's position ($\vec{O}$ in the first line, and $\vec{\theta}$ in the second line). See Fig. 4 for a reference on how camera translations and orientations work.

3. *Segment 6 calibration points:* Choose six points of the calibration target and prepare a file that lets MyPTV know their coordiantes in image space and in lab space (see Fig. 7 for the format). The name of the file should be inserted under `calibration_points_file` in the `params_file.yml`.
   The calibration points file can be generated easilly using a dedicated user interface (GUI). To run the GUI, enter `python workflow.py params_file calibration_point_gui` in the command line. Alternatively, if MyPTV can't find the `calibration_points_file`, the GUI will start automatically. Instruction are given in Fig. 5. Note, for best performance, it is best to have some asymmetry in the points chosen (e.g. see the blue crosses in Fig. 5).

4. *Calibrate using 6 points:* We now calibrate the camera (i.e. improve the initial guess), using the calibration sequence. To start the sequence enter `python workflow.py params_file.yml calibration` in the command line. Then, the following actions can be chosen by the user:

```
1 - external parameters minimization
2 - correction error term calibration
3 - print current camera parameters
4 - plot the projection of the calibration points
8 - save the current calibration results
9 - quit
```

   To calibrate, use options 1 and 2 in iterations and observe the calibration error decreasing (error is given as RMS deviation in pixel units). Once the error had converged to a low level, lower than 1.0 and prefferably lower than 0.5, save the results (using 8).

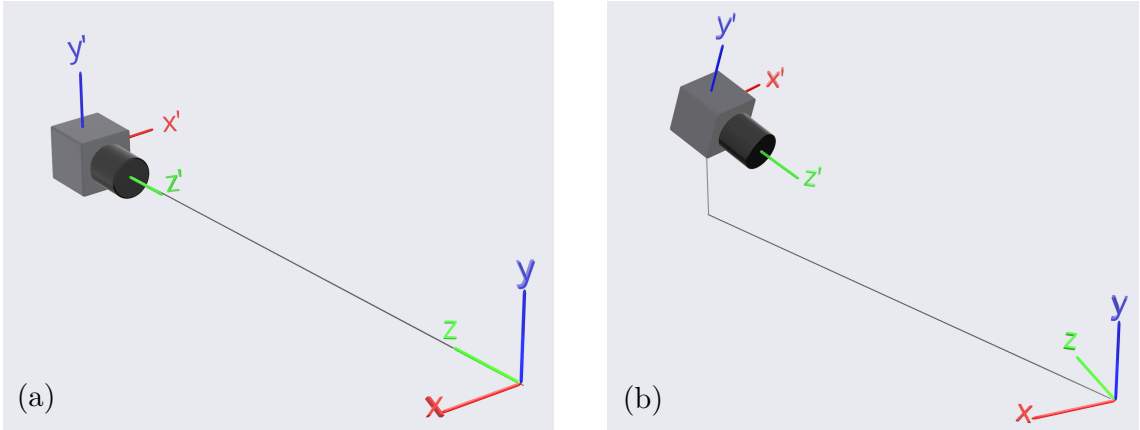This concludes the initial calibration step.

Figure 4: 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 $\vec{\theta} = (-\frac{1}{2}\pi, -\frac{3}{4}\pi, 0)$. 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 alighns with the camera's frame of reference; 2) we first rotate around the x axis, then around y and then around z; 3) in many cases the origin of the images is on the top left corner, so a half a turn rotation around the z axis is often needed.
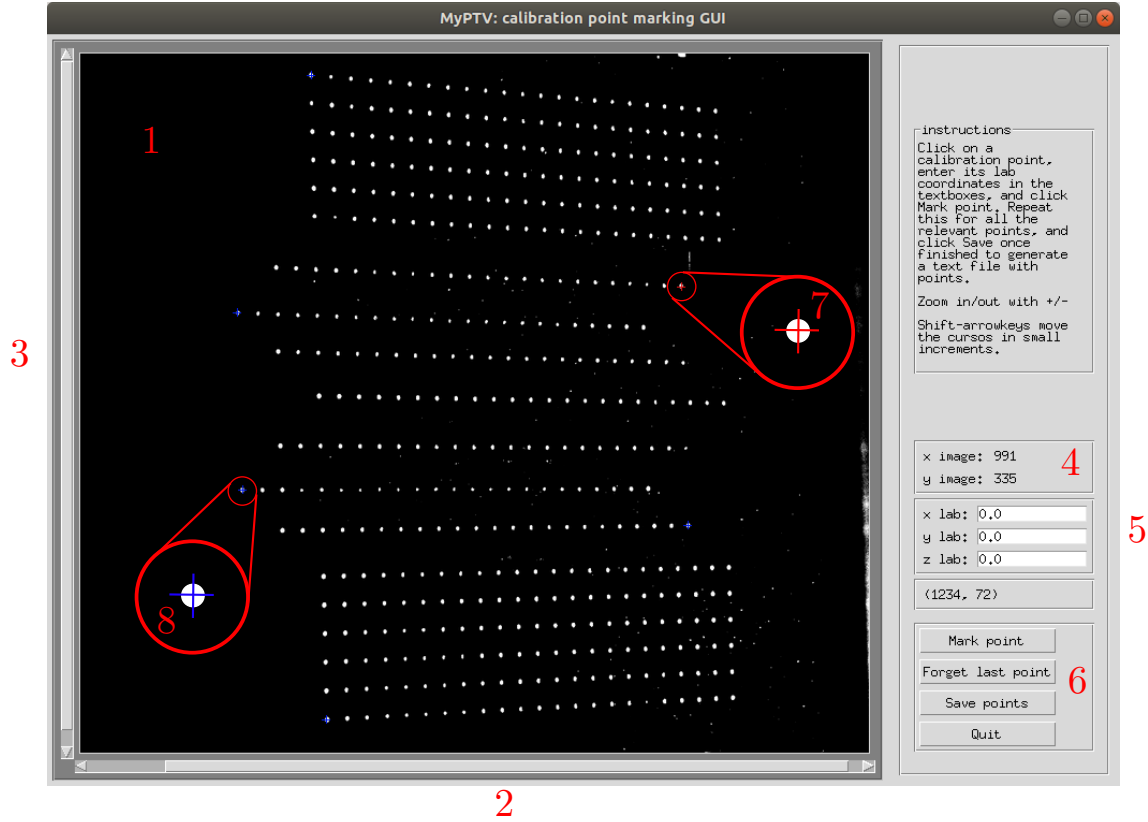
Figure 5: The calibration point segmentation GUI. 1 - the calibration image, with points to be segmented. 2 and 3 - scroll bars to explore the image. When the image is clicked a red cross apears (7) and its coordinates in image space are pronted in 4; to mark a calibration point click on it. 5 - to specify the lab space coordinates of the point, enter them in the correct tabs. 6 - once 'mark point' is clicked, the cross becomes blue (8), which means that the point's data was registered in the memory. to remove points from the register click 'Forget last point'. To save the points in the register click 'Save points' (the file name is the one given in `calibration_points_file`). To zoom in use the $+/-$ buttons. To move the red cross more accurately, use Shift-Arrowkey.

2.2.3 Final calibraiton

For the final calibration, follow these steps:

1. *Prepare a target file:* If there is no target file in the experimental folder, this is the time to generate one. A *target file* is a text file which lists the lab-space coordinates of all the points on our calibration target according to the format shown in Fig. 6. Prepare such a file and save it in the experiment's calibration folder.

2. *Segment the calibration points:* To obtain the image space coordinates of all the calibration points we use the segmentation operation of the workflow script. To do so: 1) insert the name of the calibration image under the `Segmentation -> single_image_name` tab; 2) run the segmentation command - `python workflow.py params_file.yml segmentation`. If the results are not satisfactory, change the segmentation parameters and re-run the segmentation command until you get it; then save the results (the file name used is the one under the `segmented_points_file` field in the `params_file`). See also Sec. 2.3 for more details on the segmentation process.

3. *Match points to target file:* Here we let MyPTV know which of the segmented points corresponds to which point in the target file. To do this, we used the command `python workflow.py params_file.yml match_target_file`. This will generate a figure, highlight-
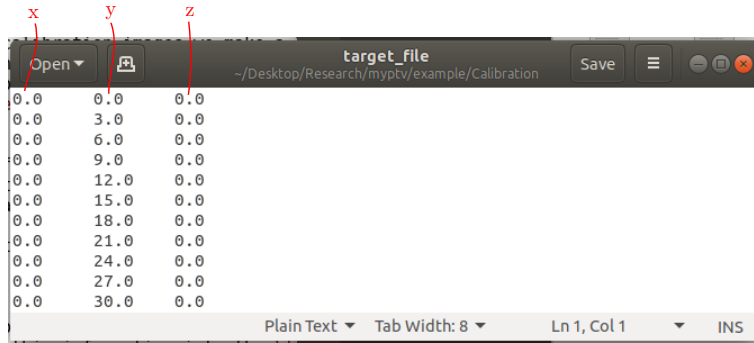
ing the target file and the segmented results overladi by one another. Make sure that the two data match each other (see Fig. 8. If the results are good, save them by quitting the figure and entering 1 when prompted. If the match is not good, then the initial calibration was not good enough; enter 2 when prompted, and return to Sec. 2.2.2 to improve the initial calibration.

4. *Calibrate using all points:* Now that we have segmented the necessary data, we return to the calibration sequence in the workflow to finish the calibration. Therefore, run `python workflow.py params_file.yml calibration` in the command line. Then, the following actions can be chosen:

```
1 - external parameters minimization
2 - correction error term calibration
3 - print current camera parameters
4 - plot the projection of the calibration points
8 - save the current calibration results
9 - quit
```

To calibrate, use options 1 and 2 in iterations and observe the calibration error decreasing (error is given as RMS deviation in pixel units). Once the error had converged to a low level, lower than 1.0 and prefferably lower than 0.5, save the results (using 8).

Congratulations, this step completes the calibration process.

Figure 6: 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.

2.2.4 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. 10 for an example. This validation step is not yet implemented in the workflow script.

2.3 segmentation

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

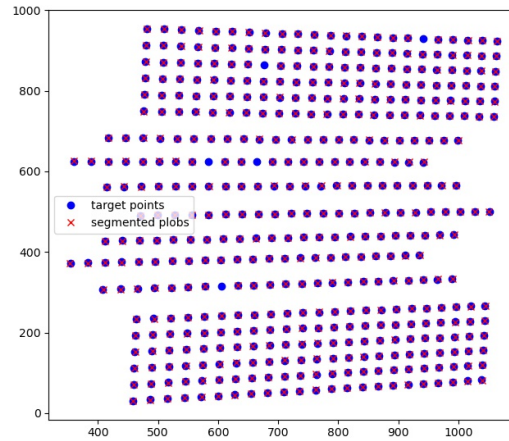Figure 7: An example of a text file holding the calibration point data.

Figure 8: 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 quiality of the calibration but it is not critical for the target file matching.
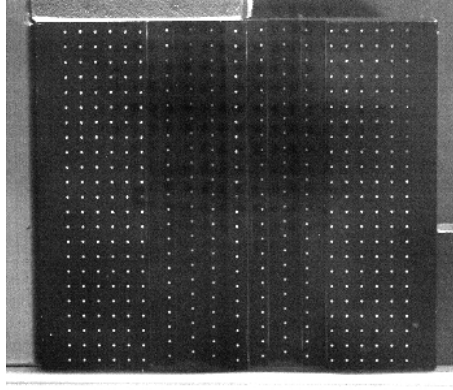
Figure 9: An example of a calibration image. The points on the target have known lab space coordinates. Not also that the points are distributed over several plains (3 different z values in this case).
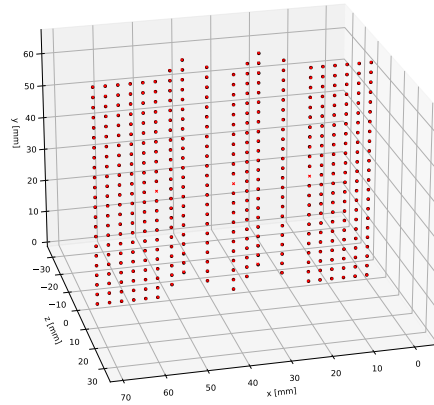

Figure 10: An example plot of a calibration error estimation. The red crosses represent the known positions of the calibration points given in the target file, and the black circles mark the positions of the stereo-matched segmented calibration points. In this case, the root mean square of the static calibration error was $84\mu m$.

9

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 (e.g. `ROI`, `threshold`, `blur_sigma`, etc.) to obtain optimal results (optionally repeat over several more images to ensure the results are satisfactory). To view the results, 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`

Table 3: The `params_file.yml` parameters for the segmentation step. All paths to files are relative to the `workflow.py` script.

Parameter	Description
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
blur_sigma	the standard deviation of a Gaussian blur filter
local_filter	the size of a local mean subtraction filter
min_xsize	minimum particle size (pixels) in x direction
min_ysize	minimum particle size (pixels) in y direction
min_area	minimum particle area (pixels^2)
max_xsize	maximum particle size (pixels) in x direction
max_ysize	maximum particle size (pixels) in y direction
max_area	maximum particle area (pixels^2)
plot_result	if `False` will not plot the results; if `True` the results will be plotted
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

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.

Table 4: 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`
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

2.5 tracking

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

Table 5: 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
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)
save_name	name of the file in which the results shall be saved; if `None` the results are not saved on the hard

2.6 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 [5]).

Table 6: 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
`polynom_order`	degree of the polynomial used in the smoothing
`save_name`	file name to save the results; if `None` the results will not be save on the disk

2.7 stitching

This step is used to re-track trajectories once velocities are known and to "stitch" broken trajectories following the algorithm of Ref. [7].

Table 7: 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

3 Imaging module - `imaging_mod.py`

The imaging module is used to handle the translation from 2D image space coordinates to lab spcae 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 saveing 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. 7).

The important functionalities 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, $(\text{x}=\vec{r})$.

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.
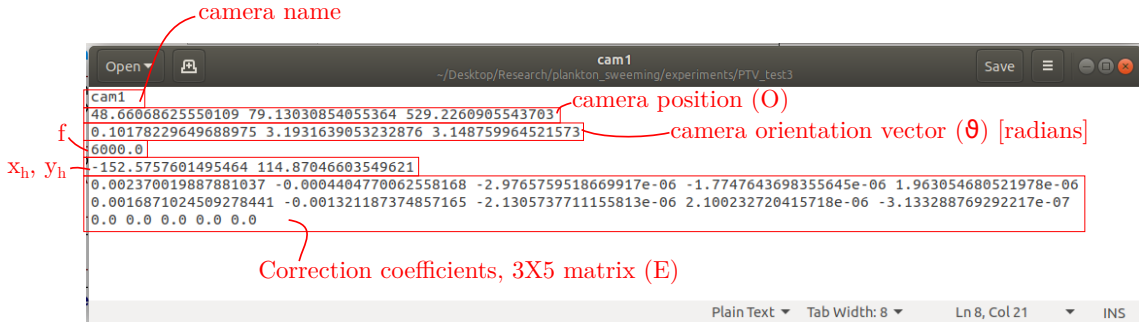

Figure 11: The structure of a camera file. The files are simple text files where each row corresponds to a specific parameter and the values in each row are separated by a white space.

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.

13

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. 7.

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.

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.

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 usefull to check the segmentation parameters manually using this `particle_segmentation` over several images in order to tune the particle searching. The inputs are:

1. `image` - the image for segmentation

2. `sigma=1.0` - the standard deviation of the blurring filter

3. `threshold=10` - the global filter's threshold brightness value (pixels with brightness higher than this number are considered foreground)

4. `mask=1.0` - A mask matrix can be used to specify rigions of interest within the image

5. `local_filter=15` - The window size (pixels) for the local filter.

6. a bunch of threshold pixel sizes in all directions and in area.

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 usefull 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.

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 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 [3], 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

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

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 neighbors 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 particlesin 3D.

16

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.

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 neighbors 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 neighbors at the second frame. Those that have neighbors 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 neighbor, two-frame, and four-frame, see Ref. [4]. 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 [4]. 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 neighbor 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})

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.

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 neighbor method. The description is the same as in Section 7.1

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. [5,6]. 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).

Figure 14: Example of a trajectory file and the column definitions.

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.
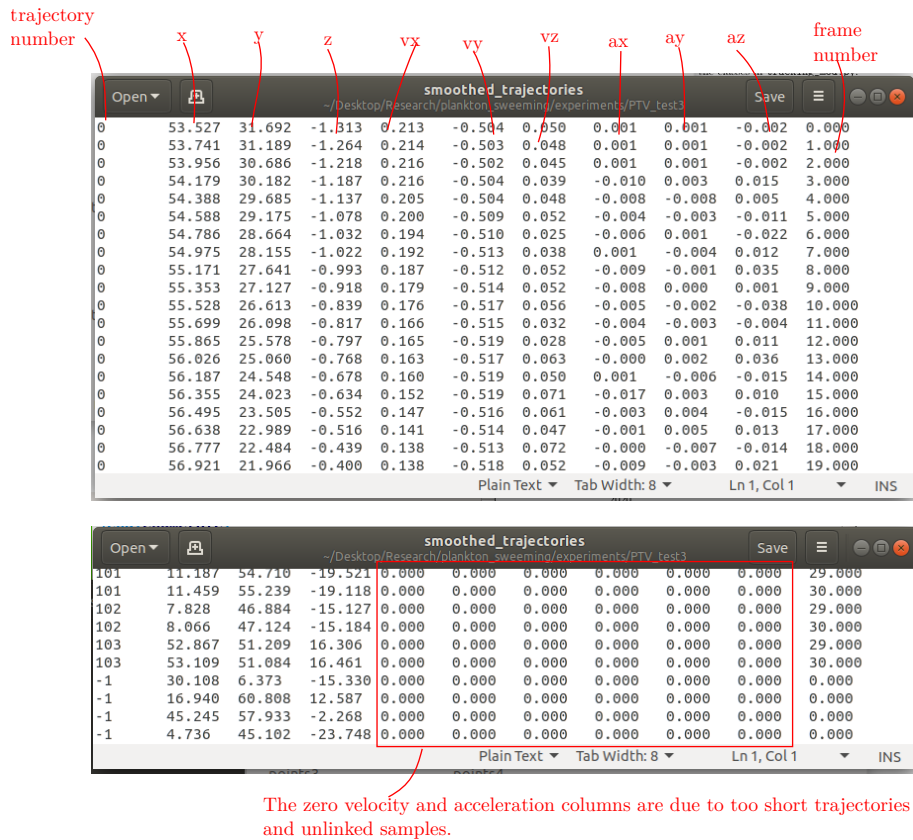

Figure 15: Example file holding the results of smoothed trajectories, and the description for each column. Note also the unsmoothed samples at the bottom of the file.

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.

9 Trajectory stitching - `traj_stitching_mod.py`

This module applies the algorithm by Ref. [7] 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*, 8:1552–1593, 1997.

[2] H. G. Mass, 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] 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.

[4] 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.

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

[6] 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.

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