MyPTV user manual

Ron Shnapp*

3rd April 2022

Version 0.1

Contents

*ronshnapp@gmail.com

1 3D-PTV principles

The 3D-PTV method is used to measure trajectories of particles in 3D space. It utilises the pronciples of stereoscopic vision in order to reconstruc 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 colinearity 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].

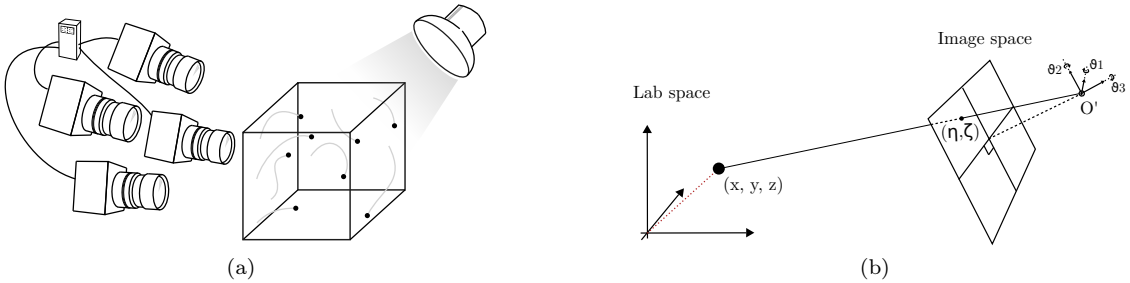

Figure 1: (a) A schematics of a 3D-PTV experiment. (b) A schematic description of the 3d model, the pin-hole camera model.

Once the experiment, namely data aquisition, 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 paritlces' 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 triagulate their positions through stereo mathcing. 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.

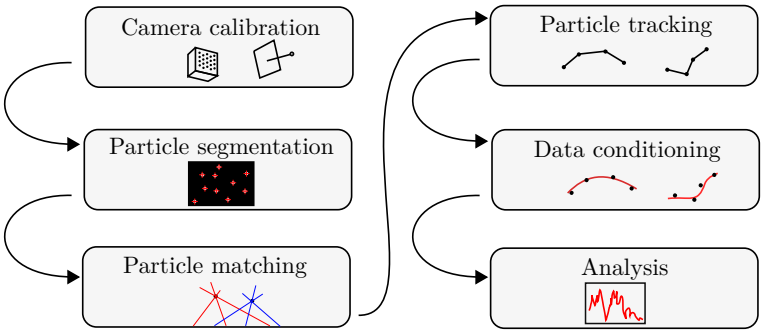

Figure 2: Basic steps in the analysis of PTV raw data into particl trajectories and scientific output. The first five steps are handled by MyPYV.

Table 1: Description of mathematical notation.

Symbol	Description
$\vec{r}$	Particle position in the lab space coorindates
$\vec{O}$	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 roation matrix which corresponds to the camera orientation vector.

2 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. For that, we use the following mathematical model:

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

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 temr $\vec{e}$ 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.

2.1 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 2.3 (see Fig. 4).

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, (`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. 3.

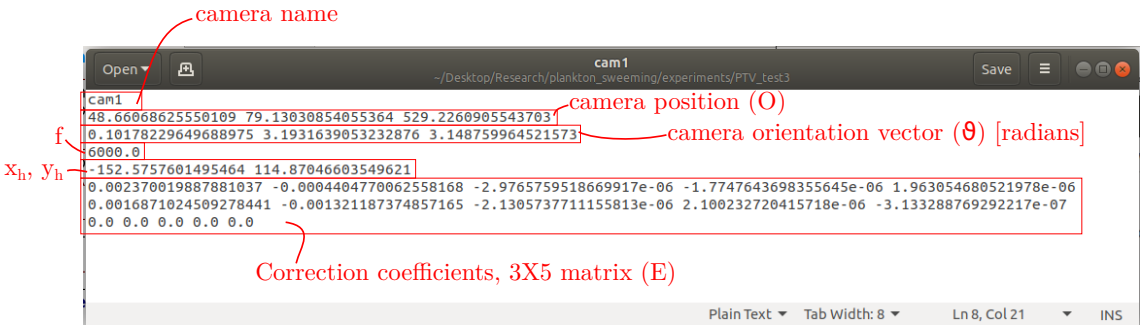

Figure 3: The structure of a camera file. The files are simple text files where each row corresponds to a specific paramter and the values in each row are separated by a whitespace.

4. `load(dir_path)` - Will load the camera parameters in a file called after the camera name in the given directory path, see Fig. 3.

After calibration we can save the camera parameters on the hard disc. The camera files have the structure shown in Fig. 3.

2.2 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 dicionary with coordinates in image space from the several cameras and calculates the triagulation position.
   The coordinate dicitonary 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.

2.3 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. 2.1). 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. 4.

3 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.

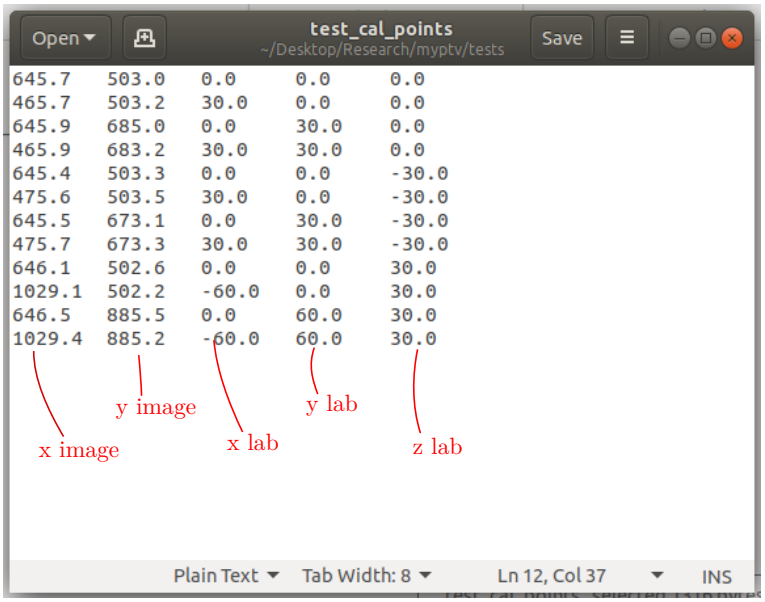

Figure 4: An example of a text file holding the calibration point data.

3.1 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 optomal 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.

3.2 A guide to camera calibration

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

To calibrate a camera, we first make sure that we have a calibration image ready (for example, see Fig. 5). Once this is done, we follow the following steps:

1. Generate a calibration point file, using the format given in Fig. 4. There are two ways of doing this:

   (a) The file can be generated by manually typing in the coordinates of the calibration points. This option is good in cases where the calibration target does not have too many points, or when an initial calibration is needed using only a small number of the calibration points. An initial calibration can be performed with as low as about six points.

   (b) When calibration is made using numerous calibration points (e.g. 437 points in Fig. 5), the file can be generated by combining a *target file* and the `particle_segmentation` object. This procedure is performed with the following steps:

      i. Manually prepare an initial calibration point file with only a small subset of the points on the calibration target ((a) above).

      ii. Proceed with the calibration steps below (2., etc.) to obtain a rough initial calibration.

      iii. Generate a *target file* - a file that lists the lab space coordinates of all the points on the calibration target

      iv. Use the `particle_segmentation` to extract the image space coordinates of the points on the calibration image. Save the extracted coordinates using `particle_segmentation.save_results(fname)`.

      v. Use the `match_cal_blobs_and_target_points` class for the `myptv.utils.py` script to automatically match the *target file* points and the image space coordinates that were segmented. Save the results on the hard drive as a text file, which holds the calibration points data.

   Once the calibration points file is ready (either initial or full), continue with the calibration.

2. Generate a *camera file* for the camera we are about to calibrate using the format shown in Fig. 3. The camera file should hold an initial guess for the camera center $\vec{O}$ and the camera orientation, $\vec{\theta}$. For a camera that is positioned in front of a calibration target and alighned with it, the following initial guess might be used:

   `0.0 0.0` z
   `0.0 3.14 3.14`
   f `0.0 0.0`
   `0.0 0.0 0.0 0.0 0.0`
   `0.0 0.0 0.0 0.0 0.0`
   `0.0 0.0 0.0 0.0 0.0`

   where z is the estimated distance from the camera to the calibration target and f is the scale factor (a typical value might be $f \sim 6000$ for a 60mm lens).

3. In a Python console, initiate an instance of the `calibrate` object with an instance of the camera you would like to calibrate. For example:

   ```
   from myptv.calibrate_mod import calibrate
   cam = camera(cam_name, resolution, cal_points_fname=cal_points_fnams)
   cam.load('./')
   cal = calibrate(cam, cam.lab_points, cam.image_points)
   cal = calibrate(cam, cam.lab_points, cam.image_points)
   ```

4. Now we are ready to use myptv to optimize the camera's parameters. As explained above, there are two components of the camera model - a linear part, which includes $\vec{O}$, $\vec{theta}$, f, and x_h, y_h, and a nonlinear correction term E. We minimize each of these two groups separately. Specifically:

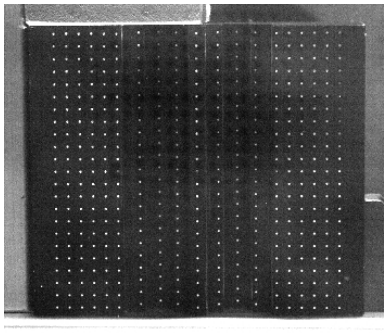

Figure 5: An example of a calibration image. The points on the target have known lab space coordinates. Not also that the points are distributed over several plains (3 different z values in this case).

(a) To optimize the linear part, we use:

```
cal.searchCalibration()
print(cal.mean_squared_err())
```

so the error is printed after each iteration. To find an optimal solution it is possibly needed to run this several times, while observing that the error is reducing (the error is given in image space coornidates, pixel).

(b) To optimize the nonlinear part, we use:

```
cal.searchCalibration()
print(cal.mean_squared_err())
```

where, again, it might be needed to run this several times to obtain a good solution.

To find a good solution, we iterate over the two steps several times, until convergence to a good solution is obtained.

$*$ A good solution will have a low error value of only a small fraction of a pixel (namely, `print(cal.mean_squared_err())` should return less than 0.5).

$**$ If a good solution is not obtained after several iterations, it is possible that the initial calibration was not good enough and that it had resulted in an error in the pairing of blobs and target file. To fix this, start the calibration again with better initial calibrations. Another issue might be that the initial guess was not very good and thus the minimization had found a wrong local minimum. To fix this, start again with a better initial guess, making sure the values of $\vec{O}$ and $\vec{\theta}$ are similar to their values during the experiment.

5. Once a good solution is found, don't forget to save the camera using:

   ```
   cam.save()
   ```

6. (Optional) 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 5) 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. 6 for an example.

4 Particle segmentation - `segmentation_mod.py`

This module handles the image analysis part of MyPTV, taking in raw camera images containing particles and outputing their image space coordinates. For the segmentation we first blur the image

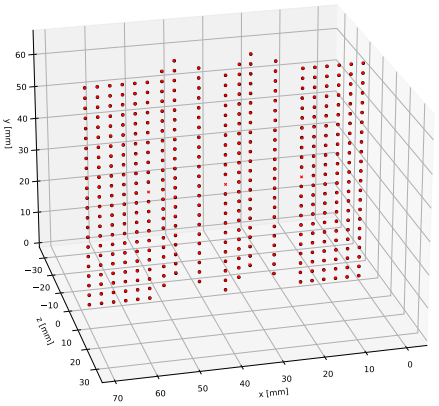

Figure 6: An example plot of a calibration error estimation. The red crosses represent the known positions of the calibration points given in the target file, and the black circles mark the positions of the stereo-matched segmented calibration points. In this case, the root mean square of the static calibration error was $84\mu m$.

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 backgroud 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.

4.1 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. 7.

4.2 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. 7.

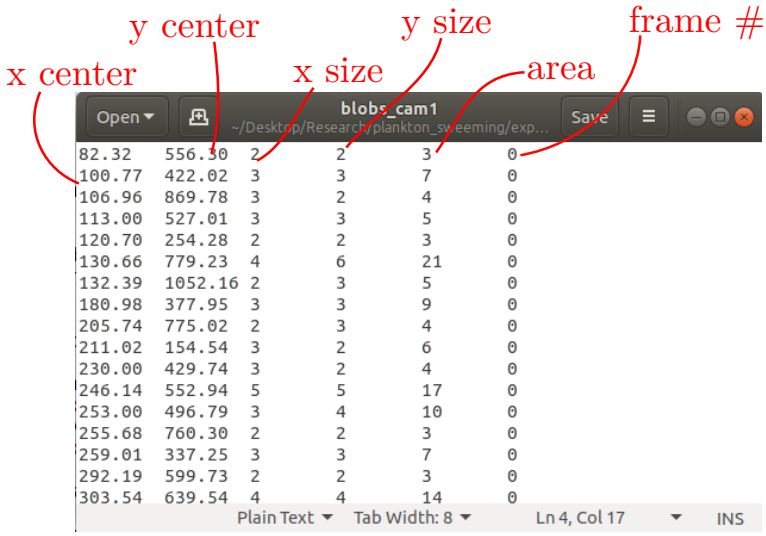

Figure 7: An example of a text file holding the segmentation resuls and the description of the different columns.

5 Particle matching - `particle_matching_mod.py`

The module used to identify the same particle in the different images and use stereo matching to estimate their 3D position. Particle matching in MyPTV uses the Ray Traversal algorith proposed in Ref [3]. In short, the 3D domain is divided into voxel cubes; then, for each segmented blob we shoot a ray throught the 3D volume and list the voxels through which the ray had passed. Finally, is more than one ray had passed through a certain voxel, we stereo match the blobs at the intersection. Lastly, for each stereomatched blob, we keep the particles corresponding to the highest number of cameras (for example, we favour particles that result from a crossing of four camera views than three) and the smallest RMS distance from the epipolar lines.

5.1 The `match_blob_files` object

This is the object that we use in order to obtain trangulated particles results from the segmented blob files (a file as the one in Fig. 7 for each camera). 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 erronous intersection of rays, leading to matching errors. Thus, this parameter should be optimized.

5. `max_err=None` - Maximum acceptable uncertainty in particle position. If None, (defult), than no bound is used.

6. `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.

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. 8).

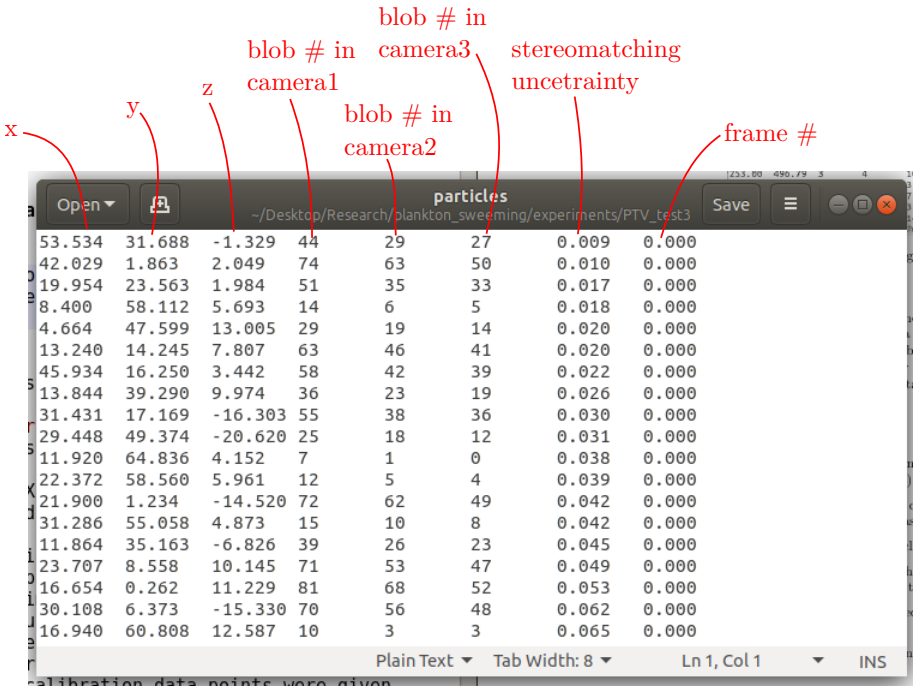

Figure 8: An example of a text file holding the triangulated particles' resuls and the description of the different columns. In this example there were three cameras.

5.2 The `matching` object

This object is the engine used backstage for matching particles. 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 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. [4]. Users are welcome to choose their perfered method and use it.

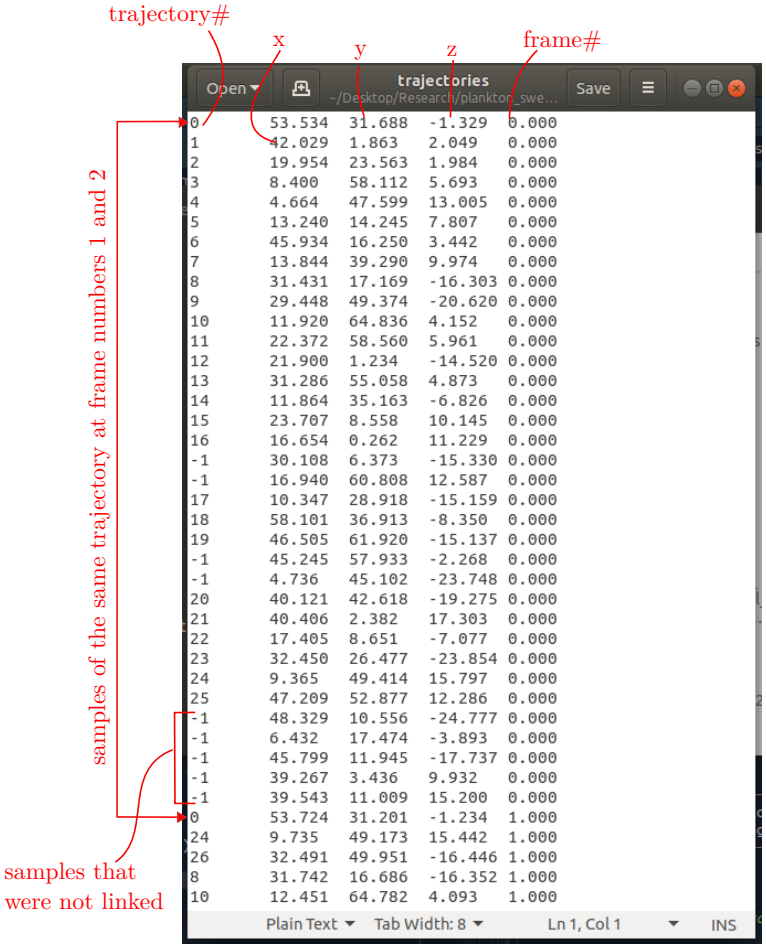

Figure 9: Example of a trajectory file and the column definitions.

6.1 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. 8
2. `mean_flow=0.0` - either zero (deafult) 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})

The important functionalities are:

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 9.

6.2 The `tracker_two_frames` object

An object used for tracking through the 2-frame method. The description is the same as in Section 6.1

6.3 The `tracker_nearest_neighbour` object

An object used for tracking through the nearest neighbour method. The description is the same as in Section 6.1

7 Trajectory smoothing - `traj_smoothing_mod.py`

This module is used to smooth trajecotries and to calculate the velocity and acceleration of the particles. For the smoothing we are using the polynomial fitting methd 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 lenght 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 valocities and accelerations only for trajectories longer than the window size for the smoothing (a user decided parameter).

7.1 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 trajecotry 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 svaed with zero velocity and accelerations.

The inputs are:

1. `traj_list` - a list of samples organized as trajecotries. This should have the same data structure used in the saving function of the tracking algorithms (see Section 6.1).
2. `window` - The window size used in the sliding polynomial fitting.
3. `polyorder` - The order of the polynomial used in the fitting.

The important functionalities are:

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. 10.

8 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.

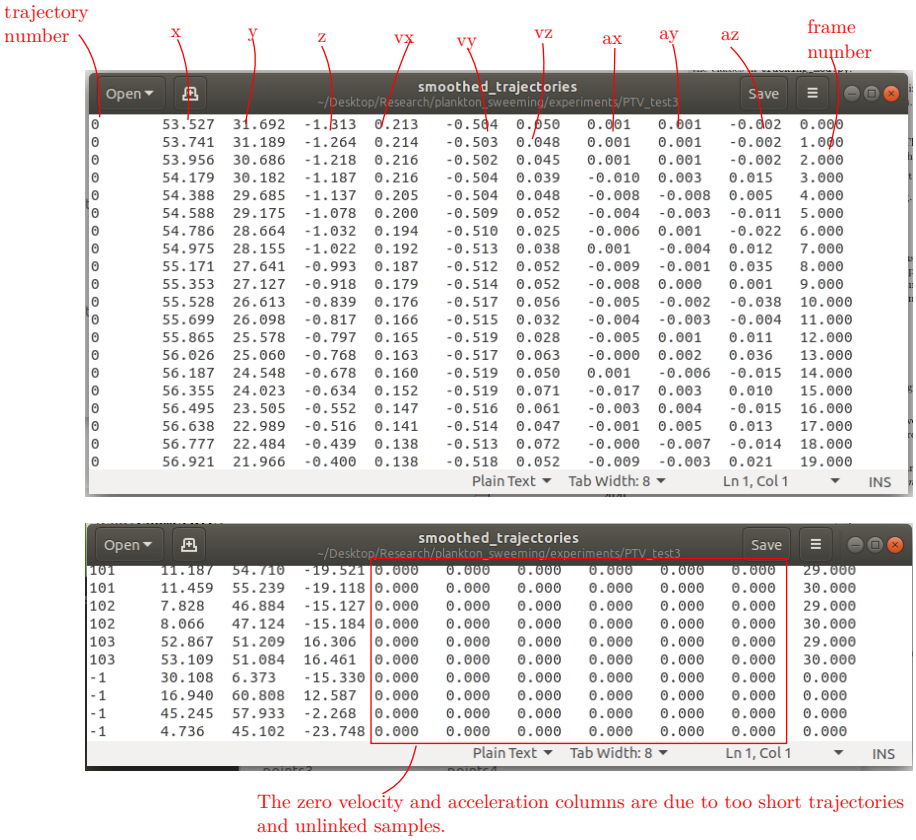

Figure 10: Example file holding the results of smoothed trajectories, and the description for each column. Note also the unsmoothed samples at the bottom of the file.

8.1 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. 10).

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.