MyPTV user manual

Ron Shnapp*

21st March 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 principles 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].

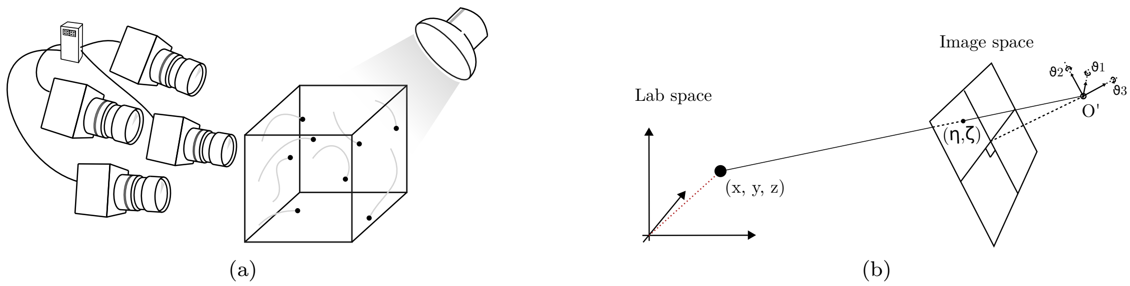


Figure 1: (a) A schematics of a 3D-PTV experiment. (b) A schematic description of the 3d model, the pin-hole camera model.

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.

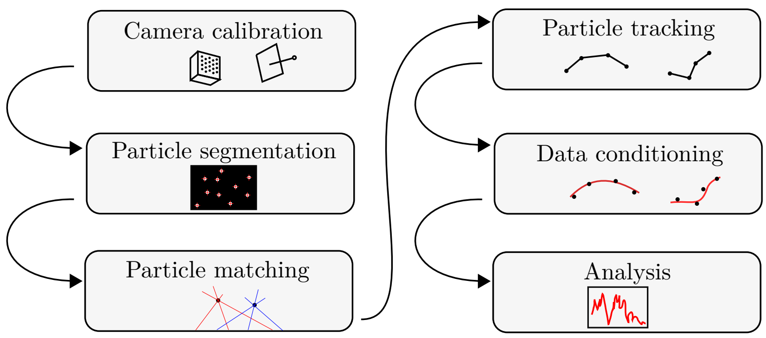


Figure 2: Basic steps in the analysis of PTV raw data into particl trajectories and scientific output. The first five steps are handled by MyPYV.

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:

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

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 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 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 $(\eta,\, \zeta)$.
2. `projection(x)` - Will reverse solve equation (1) to find the image space coordinates $(\eta,\, \zeta)$, 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.

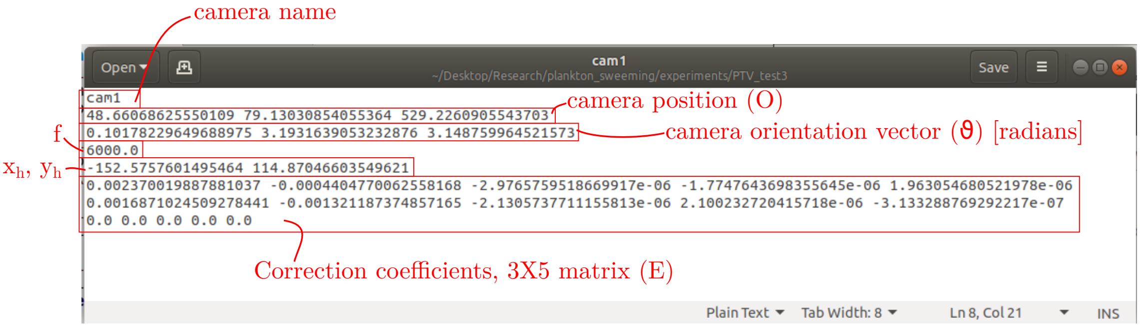


Figure 3: The structure of a camera file. The files are simple text files where each row corresponds to a specific paramter and the values in each row are separated by a whitespace.

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.

3 Camera calibration - `calibrate_mod.py`

Holds the `calibrate` object that is used to find the camera calibration parameters.

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.

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 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. Inputs:

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.

4.2 The `loop_segmentation` object

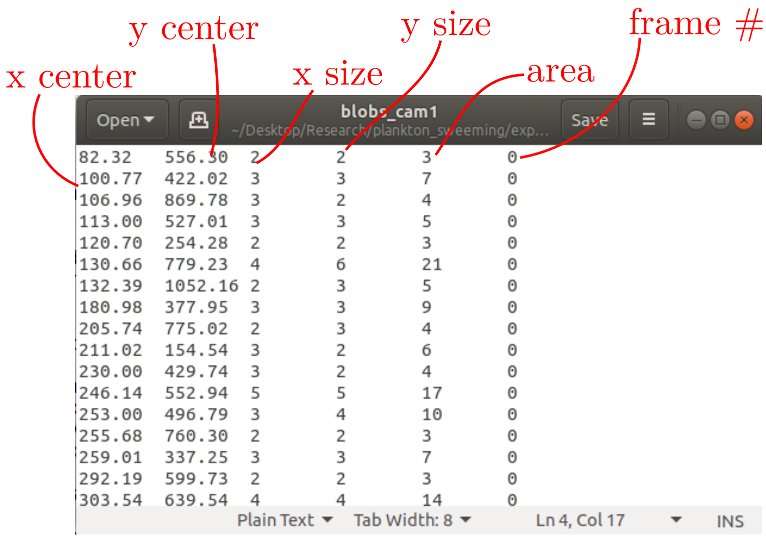
An object used for looping over images in a given directory to segment particles and save the results in a file.

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. 4.

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



82.32	556.30	2	2	3	0
100.77	422.02	3	3	7	0
106.96	869.78	3	2	4	0
113.00	527.01	3	3	5	0
120.70	254.28	2	2	3	0
130.66	779.23	4	6	21	0
132.39	1052.16	2	3	5	0
180.98	377.95	3	3	9	0
205.74	775.02	2	3	4	0
211.02	154.54	3	2	6	0
230.00	429.74	3	2	4	0
246.14	552.94	5	5	17	0
253.00	496.79	3	4	10	0
255.68	760.30	2	2	3	0
259.01	337.25	3	3	7	0
292.19	599.73	2	2	3	0
303.54	639.54	4	4	14	0

Figure 4: An example of a text file holding the segmentation resuls and the description of the different columns.

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.

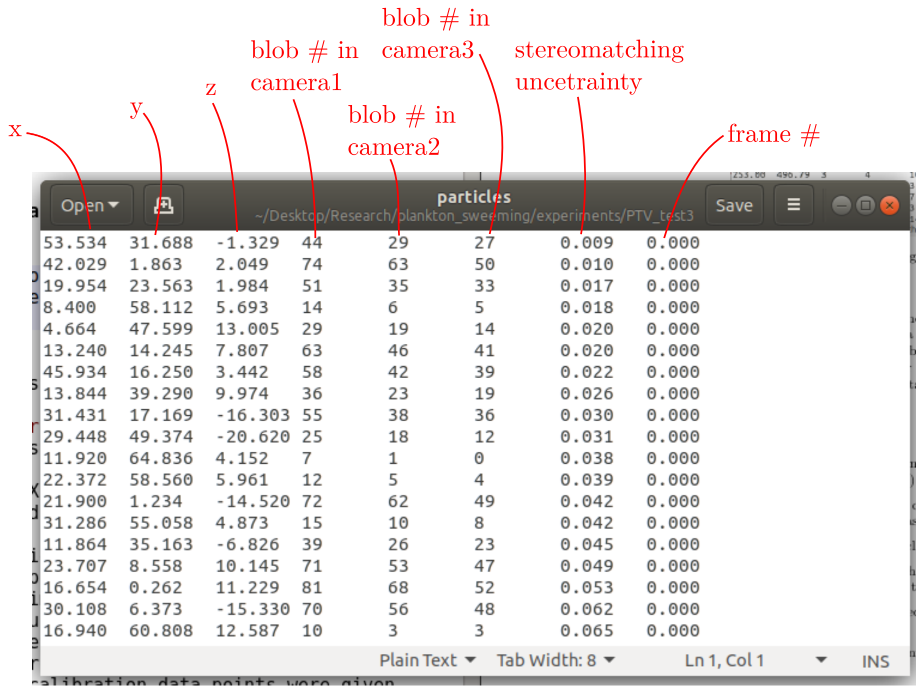
5.1 The `match_blob_files` object

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. 4 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).
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. 5).



53.534	31.688	-1.329	44	29	27	0.009	0.000
42.029	1.863	2.049	74	63	50	0.010	0.000
19.954	23.563	1.984	51	35	33	0.017	0.000
8.400	58.112	5.693	14	6	5	0.018	0.000
4.664	47.599	13.005	29	19	14	0.020	0.000
13.240	14.245	7.807	63	46	41	0.020	0.000
45.934	16.250	3.442	58	42	39	0.022	0.000
13.844	39.290	9.974	36	23	19	0.026	0.000
31.431	17.169	-16.303	55	38	36	0.030	0.000
29.448	49.374	-20.620	25	18	12	0.031	0.000
11.920	64.836	4.152	7	1	0	0.038	0.000
22.372	58.560	5.961	12	5	4	0.039	0.000
21.900	1.234	-14.520	72	62	49	0.042	0.000
31.286	55.058	4.873	15	10	8	0.042	0.000
11.864	35.163	-6.826	39	26	23	0.045	0.000
23.707	8.558	10.145	71	53	47	0.049	0.000
16.654	0.262	11.229	81	68	52	0.053	0.000
30.108	6.373	-15.330	70	56	48	0.062	0.000
16.940	60.808	12.587	10	3	3	0.065	0.000

Figure 5: 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.

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. 5
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.

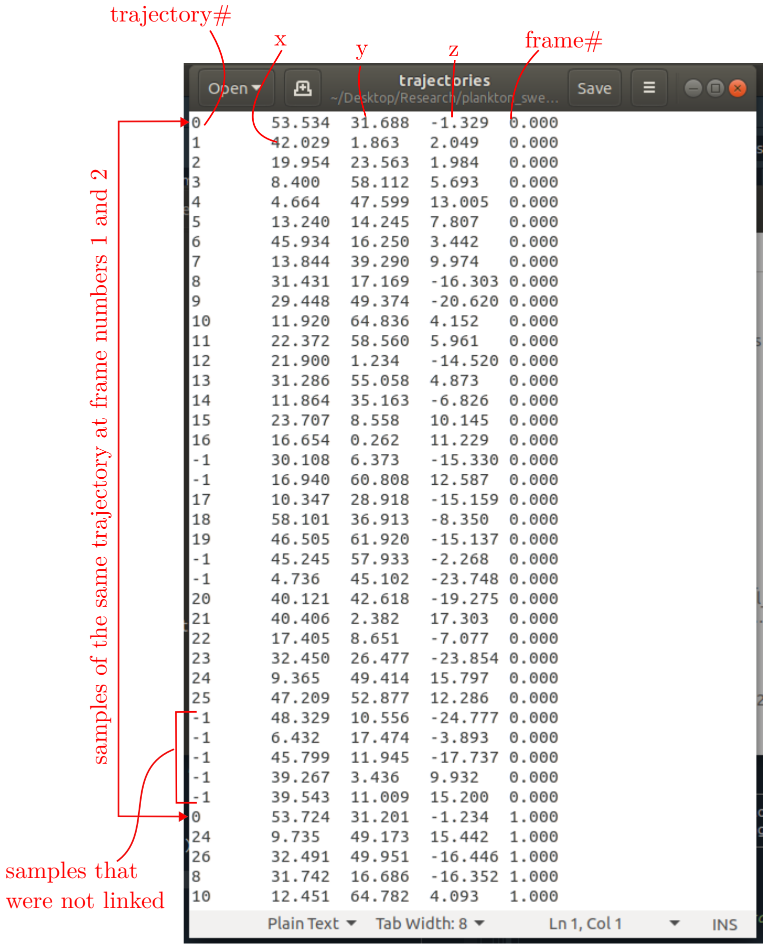


Figure 6: Example of a trajectory file and the column definitions.

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 6.

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

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.