User Guide for Quantitative Three-Dimensional Synthetic Aperture Imaging Codes

Jesse Belden

jbelden@mit.edu

May 27, 2011

Contents

Summary:
This guide describes the Matlab codes required for carrying out a 3D Synthetic Aperture (SA) imaging project. The codes are included in the file structure with top folder called SA_codes_releaseX.X.

Introduction

Figure 1 outlines the “road map" for an SA imaging fluid flow measurement project. This user guide details the first five steps, as the last step (post-processing) is application specific. Often, we will refer to [1], which is the thesis describing this work in much greater detail. All of the relevant codes are contained in the file structure with top folder called SA_codes_releaseX.X. Each code in the SA processing sequence draws from data entered by the user in the function *configdata.m*. The first cell requires the user to enter the directories that will be used for the project. The directories are contained in a struct called **paths** that contains the following fields:

code_direc: SA_codes_releaseX.X directory

calib_direc: path containing all of the calibration images (see below for organization)

save_direc: path where you want the resulting data files to be saved (refocused images will be saved elsewhere)

img_direc: path containing the images to refocus

Image Capture

This section describes the general requirements for an SA imaging system as well as some practical considerations for setting up an SA experiment. For details on a 30 Hz camera array comprising Point Grey Flea 2 machine vision cameras, as well as a high speed array using Photron cameras, please refer to Chapter 3 of [1].

Camera Synchronization

To properly refocus the data using the SA algorithms requires accurate synchronization of image capture across all cameras. Often, machine vision cameras such as the Point Grey Flea 2 offer synchronization based on a computer clock, such as the clock of the IEEE-1394 bus. However, the jitter associated with this clock is too large for SA fluid flow applications (e.g., jitter is $\pm$ 125μs for Point Grey MultiSync^{TM} software). Therefore, the SA camera array should always be synchronized using a hardware trigger, typically externally generated. All cameras should have the capability to be hardware triggered. In our experience, an externally generated timing pulse can be generated using a Berkeley Nucleonics Model 505 Pulse Generator or similar. For the machine vision cameras, each camera is set to a mode that starts frame exposure with a rising or falling pulse edge, exposes for a pre-set amount of time, and overlaps the frame readout with the next frame exposure. In the case of a high speed camera array, camera synchronization is accomplished by setting one camera to be the “master camera" and outputting a timing pulse from the master to the other cameras in the array. A hardware trigger can initiate capture.

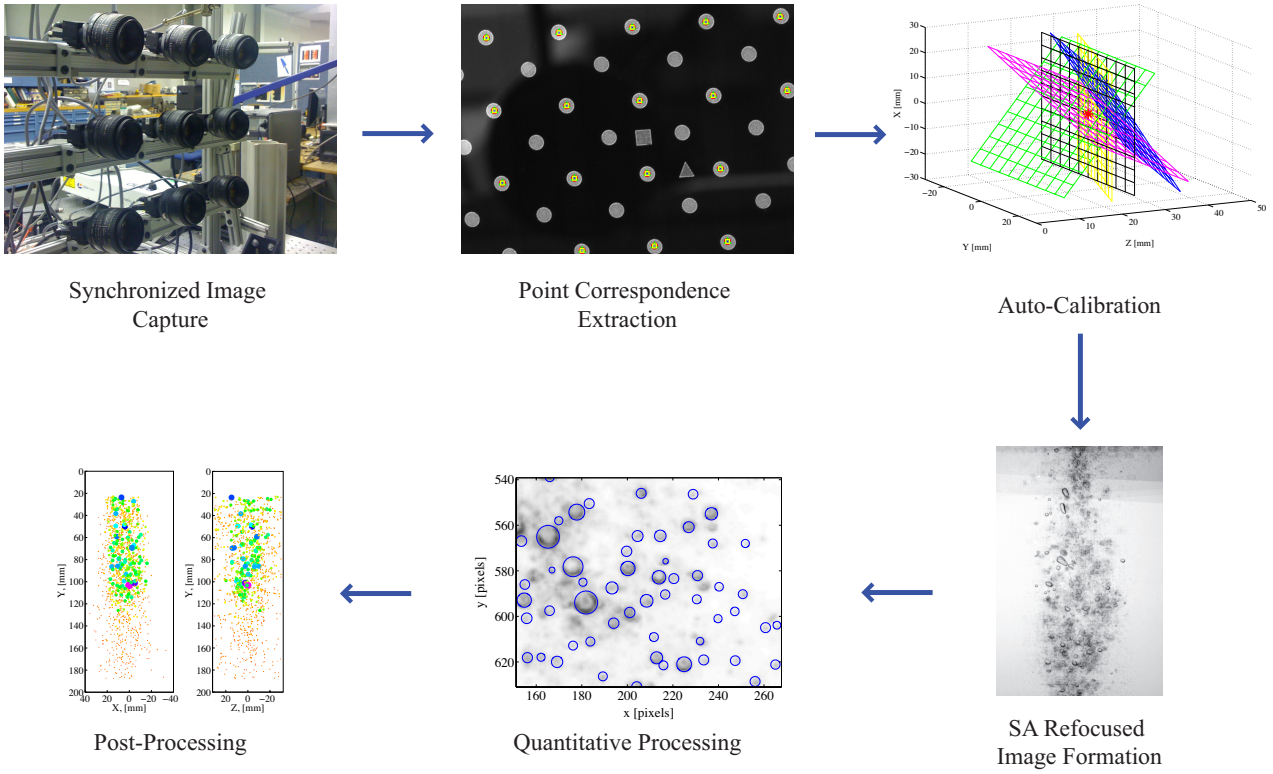

Figure 1: “Road map" for an SA imaging fluid flow measurement project.

Experimental Setup

An SA data set consists of a set of refocused planes within a volume. All cameras should be arranged so as to view the refocused planes. To accomplish this, the cameras can be arranged on a frame such as that shown in Figure 2, where cameras have degrees of freedom indicated by the arrows. As discussed in Chapter 2 of [1], the camera baseline spacing (separation between cameras relative to distance to volume) governs the depth of focus and depth resolution in the refocused volume. Therefore, within the space constraints imposed by the experiment, the cameras should be spaced as far apart as possible while still keeping the refocused planes in view. If the spacing is too wide, however, the overall viewable depth of the volume will decrease. We have found a typical number for the baseline spacing (ratio of camera spacing to distance to object plane) of 0.3-0.4 to work well in practice.

Once the cameras are approximately in place, the following procedure is recommended for alignment and focusing:

1. Place an object at the center of the target volume. This could be a calibration object or simply something that is easy to see in the image.

2. For each camera, rotate the camera until the center of the target volume appears approximately in the center of the image.

3. When each camera is in place, open the apertures as wide as possible (smallest f-number). This will minimize the depth of field.

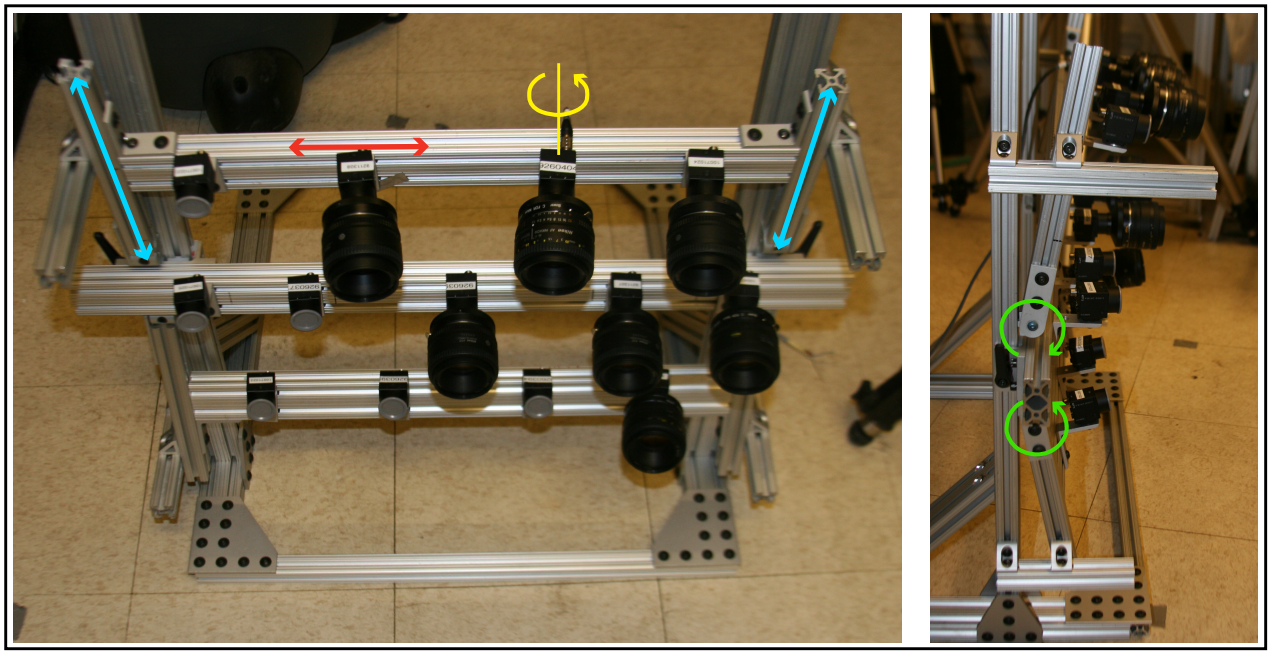

Figure 2: Point Grey Flea 2 machine vision cameras mounted on an 80/20® frame.

4. Adjust the focus of each camera.

5. Close the apertures to extend the depth of field. To determine what f-number is required, move a reference object to the front and back of the volume checking to see the object stays in focus in each image. If the object is not in focus, use a larger f-number.

Calibration

The first step in calibration involves generating image point correspondences across cameras. For a “traditional" calibration, this often involves placing a precision grid at precisely controlled locations throughout the volume. However, for the auto-calibration methods applied in SA imaging, only a small amount of well-controlled reference geometry is required and other points can be randomly distributed throughout the volume (see Chapter 4 of [1]). In practice, we have found an efficient manner for generating point correspondences is to place a precision grid in the center of the volume in a well-controlled manner to establish the world coordinate frame. From there, the grid can be moved through arbitrary locations and angles within the volume to generate point correspondences.

Point Correspondence Extraction

Main Code: calib_point_extract.m

You will first need to enter data relevant to the calibration setup in *configdata.m*. Presently, the code can accept two types of grids: checker board patterns or circle patterns. The calibration cell of *configdata.m* generates a struct called **calib_data** that contains the following fields:

grid_type: either ‘checker’ or ‘circle’

dX: spacing in X direction between grid points (units are choice of user)

dY: spacing in Y direction between grid points (units are choice of user)

nx: number of grid points in X direction on grid

ny: number of grid points in Y direction on grid

homog_definition: how you will define the points to establish the initial homography. This should be a string set as either 'user' if you know the homography defining grid points and they stay the same for each image and each plane or 'interactive' if you wish to choose different grid points for each plane.

homog_ref_points: If you select the 'user' option above, you need to include a 2 x N array of grid points defining the location of the N initial homography points. Note: N must be ≥ 4.

numcams: number of cameras

num_calib_planes: number of unique calibration plane locations

If the grid is a checkerboard pattern, then **calib_data** must also contain:

corner_win_x: x dimension size of the window for the corner finder

corner_win_x: y dimension size of the window for the corner finder

If the grid is a circle pattern, then **calib_data** must also contain:

d_circ_pix: approximate size of the circles in pixels (used in defining a template for locating circles)

Next, you need to organize the calibration images according to the file structure shown in Figure 3. The calibration directory can be named anything you want, but the plane directories **must** be named **plane1, plane2,...,planeN**. The images can also be named anything you want, but should be organized sequentially and in the same way for each plane.

Now you can run *calib_point_extract.m*. The program will load the configuration data, then the program loads and displays the first image from the first plane and asks the user to select the grid point corresponding to the origin. The origin point must be the same for every image and every plane, otherwise the points will not correspond. Then, the user is asked to select N (must be ≥ 4) points to define the initial homography. A typical result of this interaction is shown in Figure 4.

The program will then define the points in grid coordinates corresponding to the selected homography points. If you set **homog_definition** to 'interactive', then you will be asked to input the grid points for every image and every plane. For example, the four homography points selected in Figure 4 would have to be entered as (-1,-1), (-1,2), (2,2) and (2,-1) if that is the order in which the points were manually selected. If, however, you set **homog_definition** to 'user', then you will not have to input grid points at all, but will be responsible for selecting the same points in the same order for each image and each plane. In the 'user' defined case applied to the example of Figure 4, the user would provide the following array in the **homog_ref_points** field (in Matlab notation): $[-1,-1\ ;\ -1,2\ ;\ 2,2\ ;\ 2,-1]^T$. Again, more than four points can be used with either method.

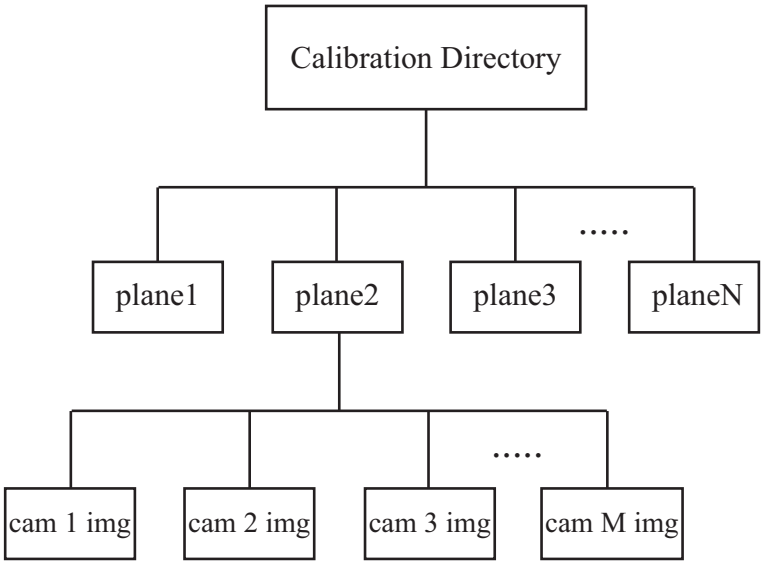

Figure 3: Calibration file structure.

Click on N points to define initial homography (at least 4), hit enter when done

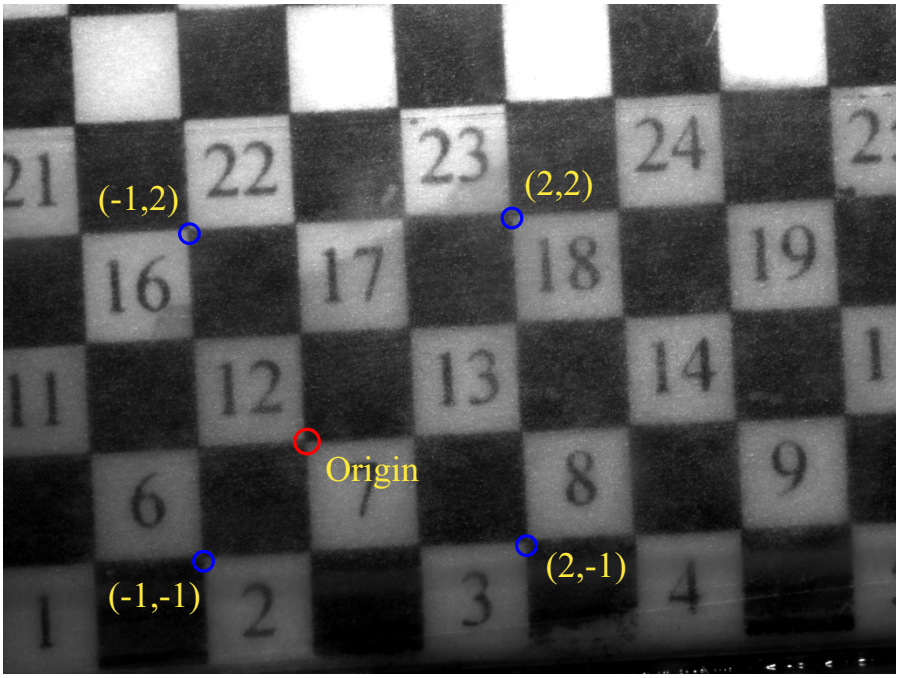

Figure 4: Example of point selection for definition of the origin and initial homography points.

Next, the program applies the initial homography to the entire set of grid points to generate the initial guesses for other grid point locations, shown by the yellow circles in Figure 5(a). As indicated by the title of Figure 5(a), the user is required to click the upper left and lower right corner of a window of points to retain. In this way, points near the edges that may not have well defined images can be excluded from the calibration. The local corner finder (or circle center detector depending on grid type) refines the estimates in the region of the yellow circles resulting in the red points locations in Figure 5(b). The resulting points are save in a .mat file in the plane directory and a single file is save for all images for this plane as well. When all images and planes have been processed, the points are saved in the calibration directory in *all_point_corrs.mat*, which contains the following fields:

umeas: 2 x (nx · ny) x numcams array containing all grid point coordinates on the grid. Any grid point that does not appear in an image is replaced with a NaN, and NaN's are dealt with internally by the SA codes.

xy_grid: 2 x (nx · ny) array containing all grid point coordinates on the grid (grid point coordinates are in integer increments). Each grid point value in **xy_grid** corresponds to the measured points contained in **umeas**.

xy_phys: 2 x (nx · ny) array that is the same as **xy_grid**, except the grid points are converted into physical coordinates.

Auto-Calibration

For this section, you need to enter data relevant to the calibration setup in *configdata.m* in the **selfcal** struct. First, enter either 'pinhole' or 'refractive' in the **model_type** field depending on the choice of model.

Pinhole Auto-Calibration

If your entire experiment is in air, or a small glass wall lies between the cameras (in air) and volume of interest (in air), then a computer vision auto-calibration method based on the pinhole camera model should work fine. We have used the Svoboda toolbox [2, 3] with very good success and this is included in the SA codes (*SA_codes_releaseX.X/Calibration/Self_Calib/pinhole/*). Please see their website for more information: http://cmp.felk.cvut.cz/ svoboda/SelfCal/.

Regardless of the method chosen to calibrate the cameras, the pinhole camera matrices should be organized as a 3 x 4 x numcams array and stored as the variable **P** and saved to the .mat file called *self_calibration_results.mat* in the calibration directory (it is up to the user to do this).

Refractive Auto-Calibration

Main Code: selfcalib_refractive_grid.m

If there are changes in the media in your imaging system, you may need to apply the refractive auto-calibration model, which is described in detail in Chapter 4 of [1]. You need to first enter some

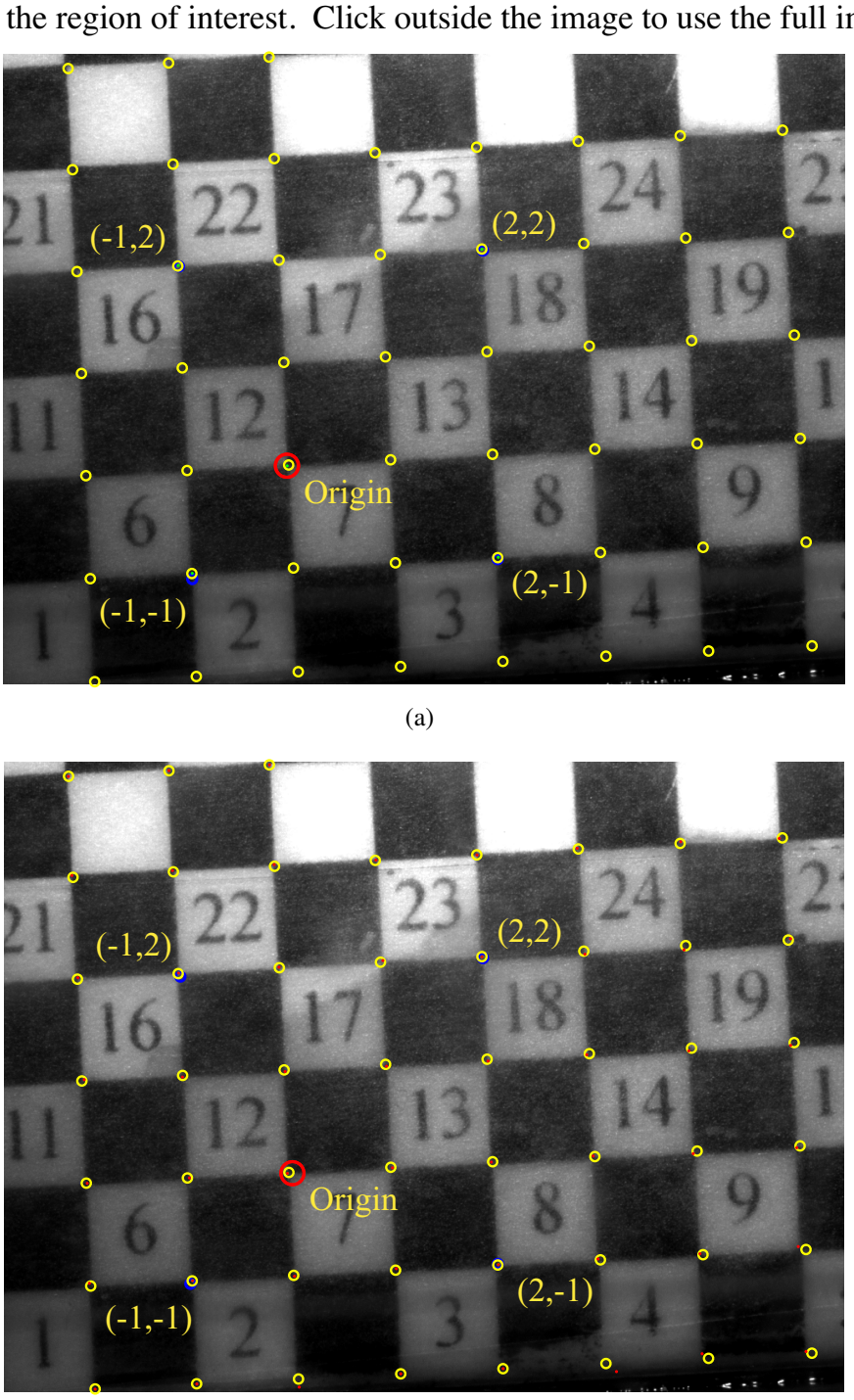

Figure 5: Results of initial homography applied to all grid points (a) and refinement of points through corner finder (b).

data in the refractive auto-calibration cell of *configdata.m*, which generates a struct called **selfcal** that contains the following fields:

rep_err_tol: target mean reprojection error in final model (note that the auto-calibration procedure may never achieve this due to image point measurement error.)

max_iter: maximum number of iterations to use in the auto-calibration routine

t: thickness of the tank wall

Zw: depth coordinate of the front of the tank wall. This doesn't influence the estimated locations of points and cameras, just sets the Z-coordinate system.

n1, n2 n3: Index of refraction of the media containing the cameras, wall and fluid, respectively.

so: Initial guess of displacement of camera centers relative to object plane (single value). This is simply to initialize the auto-calibration routine and does not have to be very accurate. **However, the sign should be correct (e.g., if Z-coordinate of cameras is less than that of the object plane, then the value of this variable should be negative).**

f: Initial guess of focal length (single value). This is simply to initialize the auto-calibration routine and does not have to be very accurate.

img_size_u, img_size_v: number of pixels in u (or x) dimension and v (or y) dimension on the image sensors, respectively. If all image sensors are the same size, then you can enter a single value for each variable. If cameras have different sensor sizes, each variable must be a 1 x numcams array.

K_signs: 3 x 3 matrix defining the sign of each term of the camera intrinsic (K) matrix (see Equations 4.11-4.12 in Chapter 4 of [1]). The signs of terms in the intrinsic matrix will depend on how your coordinates systems are defined.

image_offset: Offset in the image coordinate system. Set this to 'fixed' if you know the offset (typically half of the sensor size in pixels) or 'free' if you want these to be free parameters in the auto-calibration.

plane_params0: 6 x num_calib_planes matrix containing the initial estimates for the parameters describing the locations of the calibration planes. Each calibration plane is defined by a 6x1 vector: [phi,alpha,theta,Xp,Yp,Zp], i.e., 3 rotations angles and 3 translation coordinates. The auto-calibration will find correct models and plane locations even with inaccurate initial estimates, but will converge faster with better guesses.

refgeom: 3 x # reference points array containing the reference geometry.

refgeom_ind: 1 x # reference points array containing the indices of world calibration points corresponding to the reference geometry. This is optional, but assumes that some of the reference geometry actually comes from the calibration grids themselves.

You are now ready to run *selfcalib_refractive_grid.m*, which performs the refractive autocalibration. This code can take a little while to run due to the nature of the refractive model. The current iteration number is displayed in the command window and a plot is updated with the maximum value of mean reprojection error after each iteration.

Synthetic Aperture Refocusing

Main Codes:
pinhole_reprojection.m
refractive_reprojection.m
additive_SA_refocusing.m
multiplicative_SA_refocusing.m

For this section, you need to enter data relevant to the synthetic aperture refocusing in *configdata.m* in the **SA** struct, which contains the following fields:

pixperphys: value determining the scale (pixels per physical units) to impose in the reprojected images.

imsz_u, imsz_v: u and v dimensions of the reprojected images.

shift: 1 x 2 array defining the x and y shift of the u-v coordinate system imposed in the reprojected image (note that this has nothing to do with the synthetic aperture refocusing itself).

sign_matrix: 3 x 3 matrix defining the sign of **pixperphys** and **shift**. In Matlab notation, the matrix should be $[\pm 1, 0, \pm 1\ ;\ 0, \pm 1, \pm 1\ ;\ 0, 0, 1]$. The (1,1) entry is the sign applied to the u scale, the (2,2) entry defines the sign applied to the v scale and the (1,3) and (2,3) entries define the sign applied to the x and y shifts of the u-v coordinate system, respectively.

zmin: minimum depth coordinate of the focal planes (physical units).

zmax: maximum depth coordinate of the focal planes (physical units).

dz: spacing of the focal planes in the depth dimension (physical units).

refocus_type: either ‘add’ or ‘mult’ for the additive or multiplicative SA refocusing algorithm.

mult_expon: value of the exponent for the multiplicative SA refocusing algorithm (you can leave this blank if you are only running the additive SA algorithm).

Next, the plane to plane transformations must be established based on the calibration data. If you assumed pinhole camera models, then run *pinhole_reprojection.m*, if the refractive model was used, then run *refractive_reprojection.m*. These codes assume that the synthetic focal planes are all in X-Y planes, the range and spacing of the focal planes are set by the variables **zmin, zmax** and **dz** in the **SA** struct. The codes save each plane to plane transformation for each camera in a struct named **T** in the .mat file called called *reproj_transforms.mat* in the save directory.

The SA refocused images are then generated by running either *additive_SA_refocusing.m* or *multiplicative_SA_refocusing.m* For more information on the differences between the algorithms, see Chapters 3 & 5 of [1]. The codes read images from the **paths.img_direc** and save the refocused images to a directory named 'refocused', which is created by the code in **paths.img_direc**.

3D SAPIV

Presently, the 3D SAPIV processing isn't incorporated into the same configuration scheme as for the SA processing. However, the codes necessary to perform 3D PIV are included in the directory: *SA_codes_releaseX.X/Post_Processing/3DPIV/*. These codes are built upon the matPIV toolbox [4] and have been adapted for 3D capability.

You will need to organize the SA refocused images from adjacent time steps such that the sets of images are in directories labeled 'time1', 'time2', etc. In the *3DPIV.m* code, you need to enter data in the following fields:

img_dir: directory containing the all of the sub 'timeN' directories.

num_stdevs: number of standard deviations above the mean to set the intensity threshold at (see Chapter 2 of [1]).

winsize: 3 x 3 matrix containing the window size (in voxles) for each PIV pass. The rows correspond to pass number and the columns refer to the X, Y and Z dimensions of the windows.

overlap: value between 0 and 1 defining the percentage of window overlap.

save_dir: directory for saving the PIV data results.

The thresholding portion is run by the code *im_thresh.m*, which uses one of two methods to determine the threshold. The default method is by calculating the standard deviation of the intensity in the refocused images. The alternate method (which is commented out in the code) finds the best fit gaussian to the intensity in the refocused images, from which the standard deviation is determined. The code *3DPIV_postprocessing.m* applies several post-processing routines to the 3D PIV results; for more information on the filters, see the descriptions in the matPIV documentation [4]. You will also need to define the **direc** field to define where the 3D PIV results are saved.

References

[1] Jesse Belden. *Synthetic Aperture Imaging for Three Dimensional Resolution of Fluid Flows*. PhD thesis, Massachusetts Institute of Technology, June 2011.

[2] Tomáš Svoboda, Daniel Martinec, and Tomáš Pajdla. A convenient multi-camera self-calibration for virtual environments. *PRESENCE: Teleoperators and Virtual Environments*, 14(4):407–422, August 2005.

[3] Tomáš Svoboda, Daniel Martinec, Tomáš Pajdla, Jean-Yves Bouguet, Tomas Werner, and Ondrej Chum. Multi-camera self-calibration. http://cmp.felk.cvut.cz/ svoboda/SelfCal/.

[4] J. K. Sveen. An introduction to matpiv v.1.6.1. eprint no. 2, ISSN 0809-4403, Dept. of Mathematics, University of Oslo, 2004. http://www.math.uio.no/~jks/matpiv.