**Structure/flow of *refractiveSelfCalibration* library:**

(I only describe each function once)

*CalibrationTiff*:

Carry out the refractive autocalibration process from beginning to end, using multipage tiffs

*setupObjects*:

Initialize *planeData*, *cameraData*, *sceneData*, *refracTol* objects with experimental parameters.

(*saveCalibImagesTiff: #* currently unused

Save specific images to be used for calibration from a multipage tiff on a path.)

*getCalibImagesTiff*:

Get specific images to be used for calibration from a multipage tiff

*getSpacedFrameInd*:

Get the indices of frames to use for calibration

*findCorners*:

Find subpixel chessboard corners in images, either passed in or pointed to with a path

If the openCV cornerfinder fails:

*preprocess:*

Edit the image to make corner finding easier

If that fails:

*fixCorners*:

Try to find missing corners using the geometry of the grid

*selfCalibrate*:

Solve for camera and plane parameters, using found corners in each camera and experimental parameters

*setupCamera:*

generate initial guesses for camera parameters (7 x number of cameras)

*setupPlanes:*

generate initial guesses for plane parameters (6 x number of calibration planes)

*planar\_grid\_to\_world:*

using plane and experimental parameters, generate an initial guess for the 3D real-world positions (world points) of the found chessboard corners in the images (image points)

*P\_from\_DLT:*

Get an initial guess for the camera matrices via DLT on the guessed image points

*cam\_decomp*:

Get the camera positions (camParams 0:3) from the camera matrices

#calculate initial error

*reprojError:*

Find the difference between known image points and world points projected to the image plane (reprojection error), based on the estimated camera matrices and world points

*refrac\_proj*:

project the world points to the image plane in each camera, accounting for refraction

*cam\_decomp*:

(see above)

*img\_refrac:*

model refraction to get the length of the ray from camera to tank wall on its way to the calibration planes

if there are only 2 media:

*NR\_1eq:*

use Newton-Raphson iteration to solve the refractive equations for the length of the ray from the wall to the camera

*f\_eval\_1eq:*

evaluate the snell’s law equation n1\*sin(t1)-n2\*sin(t2) (should ideally equal 0) and its derivative

if the Newton-Raphson solver fails:

*bisection:*

Solve the refractive equation using bisection (trace the ray forwards and backwards and minimize the difference between them)

*f\_refrac:*

evaluate the refractive equation n1\*sin(t1)-n2\*sin(t2)

if there are 3 media:

if any points are on the tank wall:

solve for their ray using *bisection*

*NR\_2eq*:

use Newton-Raphson iteration to solve the refractive equation for the length of the in-wall ray for in-tank points

*f\_eval\_2eq:*

evaluate the snell’s law equations and their derivatives for a ray in 3 media

if the Newton-Raphson solver fails:

if any points are on the tank wall:

*bisection:*

(see above)

*refrac\_solve\_bisec:*

iteratively use bisection to solve the refractive equations in 3 media for the length of the in-wall ray

*bisection:*

(see above)

*f\_eval\_2eq:*

(see above)

#iteratively adjust camera and plane parameters to minimize error

while error > tolerance and iterations < max and error is decreasing:

*cam\_model\_adjust:*

find the best-fit camera model for a set of world points

*refrac\_project\_onecam:*

project world points to 2D image coordinates in one camera using 7 camera parameters

*P\_from\_params:*

Get the camera matrix from camera parameters

*refrac\_proj:*

(see above)

*planar\_grid\_triang:*

find the best-fit world points for a set of cameras

*planar\_grid\_adj:*

use plane parameters to project world points into cameras

*refrac\_proj:*

(see above)

*planar\_grid\_to\_world:*

Calculate world points based on plane parameters

*reprojError*

(see above)

*showCalibData*:

plot the reconstructed model of the experimental setup

*getScale*:

get the average physical size of a pixel in the images (useful for refocusing)

*saveCalibData*:

save the found calibration data in format for the C parser

**Parameters for *CalibrationTiff (main function):***

datapath - Path to stored images

exptpath - Path for saved output

camids - Names of cameras being calibrated

n1 - indices of refraction of 3 media in system

n2

n3

dx - spacing of grid points on calibration grid

dy

nx - number of grid points in each row, column on grid

ny

pix\_pitch - pixel pitch (size of a pixel)

so - distance of cameras from tank

f - focal length of the lens

tW - tank wall thickness (2nd refractive medium)

sx - dimensions of images being calibrated

sy

zw - z coordinate of wall in coordinate system being used

znet - total traverse of calibration plate in millimeters

ncalplanes - number of planes

nframes - number of frames in calibration video/tiff -- optional, but preferrable to provide

tol - tolerance for Newton-Raphson snell's law solver

fg\_tol - tolerance for output of snell's law function

maxiter - max number of iterations for solvers

bi\_tol - tolerance for bifurcation snell's law solver

bi\_maxiter - max number of iterations for bifurcation snell's law solver

z3\_tol - tolerance for distance of grid points from tank wall

rep\_Err\_Tol - tolerance for final reprojection error

**Useful bits for future implementation:**

The *findCorners, selfCalibrate, showCalibData* series should be broadly applicable; pass in a set of images and their associated planedata, camdata, scenedata and tolerances, and the functions will output all relevant calibration data. Other functions are primarily optimized for multipage tiffs and the current use case.

Providing the number of frames in a calibration tiff is not strictly necessary, as this can in theory be extracted from the video using tiffCapture. However, apparently there’s no easy way to do that, and I had this process crash several times. Easier just to open the video and manually copy in the number of frames.

*findCorners* will accept either images passed directly or a path to a folder containing stored images (as long as stored images are correctly named).

NaN handling has mostly survived from the MATLAB code, although it is entirely unused and untested in this code’s current form. The subfunctions of selfCalibrate \*should in theory\* be able to handle world point arrays containing NaNs.

The *planeData*, *cameraData*, *sceneData*, *refracTol* objects inherit from a *parameters* superclass and have a class method, *getInfo*, which returns the names of inputs needed to initialize the class and parameters it holds, and an instance method, *params*, which returns the names and values of parameters held by a specific object. All of these objects are iterables.

*planeData* can be initialized either with z0 (the origin of the z coordinate for each plane) or with znet (the net traverse of the calibration plane) which will be used to derive z0.

Some functions have optional Boolean parameters. These are primarily intended for debugging; for example, parameters starting with print\_ or show\_ enable additional output.

**FixCorners:**

This is the only bit of code I wrote entirely on my own – and somewhat hastily – so some explanation is probably necessary.

The code has lots of nested subfunctions. All of these should be lambda functions, I think (no real reason to do memory assignment), but some were just too long to be readable in that format.

Assume a grid missing some points (the code cannot handle extra points).

Points with no other point in a 180 degree arc are assume to be edges of the grid, and those with no point in a 240 degree arc (not 270 because of perspective) are assumed corners.

Missing corners are found by drawing lines through the edge points and checking that a corner lies on each intersection of perpendicular edges.

Then, using intact edges as starting points, all of the grid lines are drawn.

Finally, using the found grid lines, the point array is reconstructed entirely from the intersections of the perpendicular lines.

If inconsistencies are found at any point, a warning is raised and the found points at that time are returned in no order.

The last step is reordering. The grid is divided into columns using the bottom row, and points are sorted by column first, then y coordinate within the column (corresponding to row).