Assignment 6 – due 5/20/2017 at 11:59pm
Topic: Positional Tracking via Pose Estimation

This is a programming assignment. At this stage, we assume that you participated in laboratory 5 and successfully completed it.

Students are strongly encouraged to use the Arduino environment for this assignment. Other programming environments could be generally acceptable, but will not be supported in the laboratory, the office hours, or on the piazza. Students may use the computers in Packard 001, which have Microsoft VisualStudio and the Arduino IDE programming environment installed. Each team of 2 students can share a single computer for the lab. You can work in teams of no more than two for the assignment, but you can also do it by yourself. If you work with another student, please acknowledge that student in your submission. You can submit this assignment as a group on Gradescope.

Please document all your answers, plots, and insights in a **single pdf** file containing all requested results. Please submit relevant sections of your source code in that pdf file as well. When submitting to Gradescope, please have each task start on a new page and make sure to mark all pages in a PDF that pertain to the particular task. In addition, please submit a .zip file with only the source code found in root code directory as well as shaders, unless otherwise instructed. Do **not** include anything from the external_dependancies or stage directories. Please remove the dlls as well.

Task 1 of 5 (5 points)

The Arduino starter code uses the *setup()* function to initialize the tracking procedure. In the *loop()* function, which is the main callback, we first query for raw photodiode measurements using the following function

```
bool updateSuccess = photodiodes.updateClockTicks();
```

Every time you call *updateClockTicks()*, the function will try to read the latest timing values from each of the 4 diodes. Since the Teensy runs much faster than the HTC Lighthouse laser sweeps update, *updateClockTicks()* will not always be able to read new values for each photodiode, but it will try and update as many of the 8 values (x and y for each photodiode) as it can. Only when all 8 values are updated will the function return *true*. And only when it returns *true* should you continue with your calculations.

Your first task is simple: after getting the raw photodiode timings from *readPhotodiodes()*, you will implement the function *update2DPositions()* in the file “photodiodes.cpp”. This function uses an internal variable of clockTicks (ordered as {sensor0.x, sensor0.y, sensor1.x, sensor1.y, sensor2.x, sensor2.y, sensor3.x, sensor3.y}) and writes the resulting values into the projection2D_ array in the same order. In this function, you should first convert the raw clock ticks to azimuth and elevation angles for the measured x and y timings, respectively. Then, you convert this angle to a relative position at unit distance away from the camera. Follow the equations in the lecture 11 slides.

We have created a unit tester to help you evaluate your implementation of *update2DPositions*(). The unit tester is called during the setup() loop of each Arduino run and the results of the test are printed out into the serial monitor. The resulting difference between your unit test and the expected solution should be zero. Be sure that your *update2DPositions()* is passing the unit tester before moving on.

Task 2 of 5 (30 points)

Implement the homography method for pose estimation. Note that we are only interested in the resulting position and we will discard the rotation part of the homography matrix. All code for this task should be implemented in the *HomographyPose* class in *HomographyPose.cpp*. Within the setup() loop of the Arduino, we run unit tests for you of all the following functions and their expected return values, given a specific input. Be sure that the unit tests are passing before moving on.

A) First, within *updateParameters*(), create the vector *b* that is a duplicate of the *float* projection2D* array of values returned from the photoDiode class. (2 points)

B) Next, set up the matrix *A* as discussed in lecture in the *updateParameters(float* projection2D)* function. *A* describes the linear function between the homography values we are attempting to solve for, *h*, and the projection of the photodiodes onto the light house "sensor" plane, which you already computed in *PhotoDiodes::update2DPositions()*. You will need to make use of the features we are tracking in object space, defined by *object2D*. We call it 2D because our object in this case is a plane. (5 points)

C) Now, we've defined both *A* and *b* in the linear system of equations *Ah=b*, and with the 4 photodiodes we have fully constrained the problem, hence we can simply solve for *h*. Solve for the homography values, *h*, in *solveForHomography()*. Make use of the *matrixMath* library perform matrix inversions and multiplications. Check if *Matrix.Invert()* returns 0, in which case the matrix is not invertible or is ill-conditioned (i,e. something went wrong before somewhere). Make sure to return *false* in solveForHomography*()* if *Matrix.Invert()* fails. If the inversion is successful, multiply the matrix inverse with *b*, storing the result in the member variable *h*. (5 points)

D) When solving for *h* in *solveFor3DPosition()*, we only estimate the homography up to scale, meaning that we assume $h_{33} = 1$ in the computation. But the homography we are looking for actually has a $t_z = sh_{33}$ component. Estimate this normalization factor, *s*, from the estimated homography values and compute the estimated 3D position of the VRduino in *solveFor3DPosition()*. Store these three translation elements in the *position3D* array and return the normalization factor. (5 points)

E) Compute the residual in the computeResidual() function. To do this, construct the homography matrix *Rt* based on the scaled homography estimate, *h*, the normalizationFactor, and the estimated *position3D* of the VRduino. For each photodiode, forward project the 2D coordinate of the diode by the homography matrix *Rt* onto the lighthouse's "sensor plane" at unit distance 1 away from the lighthouse, represented in the lecture slides as p_x^{2D} and p_y^{2D}. Calculate the L2 residual between these estimated projections with the homography and the measured projection of the photodiodes onto the sensor plane, i.e. *projection2D*. Return the residual. (5 points)

F) Finally, implement the full homography based algorithm by combining the functions you've implemented above in computePosition(). The *computePosition()* function takes in the measured projections of the photodiodes onto the light house "sensor" plane. Make sure to return false if any matrix inversion is false because the values for that position will be invalid. Store the estimated 3D position of the board in the *position3D* member variable. (5 points)

G) The estimated positions, especially the z coordinate, can be a little noisy. In the *computePosition*() function, implement a first order IIR filter over the Z coordinate. For this IIR filter, we consider a filter $z_{filtered}[n] = \alpha z_{filtered}[n-1] + (1-\alpha) z_{unfiltered}[n]$, where n represents a time stamp. Report the

tuning value α that works well and briefly discuss the effect a larger or smaller values of α. (3 points)

Task 3 of 5 (30 points)

Implement the Levenberg-Marquardt algorithm for pose estimation in the LevenbergMarquardtPose class. All code for this task should be implemented in "LevenbergMarquardt.cpp". For each of these parts make sure to check your implementation against the unit tests provided for you before moving onto the next part!

a) Implement the function *eval_g_at_x()*. This function maps the rotation/translation values stored in x into the homography values, i.e. h = g(x). This function takes a 6-element float array *x* as input that contains the 3 rotation values theta_x, theta_y, theta_z, t_x, t_y, t_z and a 9-element float array as ouput that contains the elements of the 3x3 homography matrix in a row-major format, i.e. h[0] = h_11, h[1] = h_12, h[2] = h_13, h[3] = h_21 … Update the output h by using the input *x*. (5 points)

b) Implement the function *eval_f_at_h()*. This function evaluates the function f = f(h), which maps the homography to the projection of the photodiodes onto the light house's "sensor" plane. This function takes the 9-element array *h* representing homography matrix in row-major format as input and outputs 8-element array corresponding to the 2D positions of the photodiode projections onto the plane unit distance from the light house. The output array is stored in the member variable *f*, and is formatted as follows [pd0.x pd0.y pd1.x pd1.y …]. (5 points)

c) Implement the function *get_jacobian_g()* to compute the Jacobian matrix of g(x). Follow the derivation in the lecture 12 slides. This function takes a 6-element array *x* as input and a 9 x 6 2D float array *Jg* as output. Update the output *Jg* by using the input *x*. (5 points)

d) Implement the function *get_jacobian_f()* to compute the Jacobian matrix of f(h). Follow the derivation in the lecture 12 slides. This function takes a 9-element homography array *h* for this. Update the 8x9-element float array *Jf* by using the input array h and *object2D* variable of this class. The *object2D* variable is the 3D positions of the 4 photodiodes in object coordinates. (5 points)

e) Implement the function *compute_delta_x()* to compute the LM update step. Follow the optimal step described in Lecture 12 for the LM algorithm. (5 points)

f) Finally, implement the full LM algorithm by combining the functions you've implemented above in computePosition(), by iterating over multiple steps. The number of iterations is defined by the *kMaxIters* member variable. Make sure to initialize your estimate on your rotation/position with *positionEstimate* and then afterwards in every loop make steps towards the minimum by adding the output of *compute_delta_x()* to the current estimate. Make sure to return false if any matrix inversion is false because the values for that position will be invalid. Store the output of your best estimate on the rotation/translation in the *pose3D* member variable(). (5 points)

Task 4 of 5 (25 points)

a) A good way to see if your LM (or any optimization) algorithm is converging is to keep track of the residuals after iterations. Store the residual after every iteration in the residual member variable. Looking at residuals also a good way of comparing the performance of the algorithm under different hyper parameters like the dampening constant, λ, and the number of iterations. Let's do a small

parameter search over these variables. Report the average residual (over all iterations) and the residual of your output, for λ = 0.001, 0.1, and 1.0 and *kMaxIters* = 1, 10, 25. That's 18 values total. (9 points)

b) Report the residual due to the output of the homography based method. (1 points)

c) You might see that the residual due to the homography method is lower than due to the LM method. For the best λ setting that you found, how many LM iterations do you have to run to achieve a comparable residual to that of the homography method? You might find it helpful to plot the residuals that you collected for each iteration of the Teensy *loop()* call. Also, you should turn off auto-scrolling in the serial monitor and make sure you are fairly comparing the residuals for the outputs of the two algorithms for the exact same input.

Does LM ever achieve the same precision as the homography method? If yes, is the program still running in real-time for the number of iterations when it does? If no, does that mean that LM is worse than homography? Discuss your results and issues around comparing the residuals. (5 points)

d) In which of the following scenarios would it not be possible to estimate 3D position using the homography method? Briefly discuss as to why or why not homographies would be able to handle these scenarios.

1) 3D arrangement of photodiodes on the HMD, as opposed to the planar arrangement on the VRduino (3 points)

2) Focal length $f_{x/y}$ is not equal to 1 and principle point $c_{x/y}$ not equal to 0 (3 points)

3) Accounting for the lens distortions in the Lighthouse projector with the following forward model (4 points)

$$p^{3D,view} = K * [R|t] * p^{3D,object}$$

$$p^{2D} = (K_1 * r + K_2 * r^2)\left(\frac{p_x^{3D,view}}{p_z^{3D,view}}, \frac{p_y^{3D,view}}{p_z^{3D,view}}\right)$$

$$r = \sqrt{((p_x^{2D})^2 + \left(p_y^{2D}\right)^2)}$$

Remember that the original forward model was:

$$p^{3D,view} = K * [R|t] * p^{3D,object}$$

$$p^{2D} = \left(\frac{p_x^{3D,view}}{p_z^{3D,view}}, \frac{p_y^{3D,view}}{p_z^{3D,view}}\right)$$

Task 5 of 5 (10 points)

As a more theoretical exercise, answer the following questions in the homework write-up that you submit.

a) Describe with your own words what the *focal length* and *principle point* are in the camera matrix. (1 points)

b) Why can we use a 3x3 camera matrix here whereas the OpenGL pipeline uses a 4x4 projection matrix for their pinhole camera model? (2 points)

c) How are the entries of the 3x3 camera matrix and the 4x4 OpenGL projection matrix related? (2 points)

Questions?

First, Google it! It's a good habit to use the internet to answer your question. For 99% of all question, the

answer is easier found online than asking us. If you can't figure it out this way, post on piazza and definitely make sure you attend the lab on Fridays (in Packard 001).