Homework 6: Pose Tracking

EE267 Virtual Reality 2024

Due: 05/16/2024, 11:59pm

Instructions

Students should use the Arduino environment and JavaScript for this assignment, building on top of the provided starter code found on the course website. We recommend using the Chrome browser for debugging purposes (using the console and built-in debugger). Make sure hardware acceleration is turned on in the advanced settings in Chrome. Other browsers might work too, but will not be supported by the teaching staff in labs, piazza, and office hours.

The theoretical part of this homework is to be done individually, while the programming part can be worked on in groups of up to two. If you work in a group, make sure to acknowledge your team member when submitting on Gradescope. You can change your teams from assignment to assignment.

Homeworks are to be submitted on Gradescope (sign-up code: **RKJZDW**). You will be asked to submit both a PDF containing all of your answers, plots, and insights in a **single PDF** as well as a zip of your code (more on this later). The code can be submitted as a group on Gradescope, but each student must submit their own PDF. Submit the PDF to the Gradescope submission titled *Homework 6: Pose Tracking* and the zip of the code to *Homework 6: Code Submission*. For grading purposes, we include placeholders for the coding questions in the PDF submission; select any page of the submission for these.

When zipping your code, make sure to zip up the directory for the current homework. For example, if zipping up your code for Homework 1, find `render.html`, go up one directory level, and then zip up the entire `homework1` directory. In addition, for this homework, **delete the `node_modules` folder before zipping** the directory, which will make the upload go more smoothly on Gradescope. Your submission should only have the files that were provided to you. Do not modify the names or locations of the files in the directory in any way besides the deletion of the `node_modules` folder.

For this week's topic, course notes are available on the course website. These notes go into more detail for the mathematical concepts discussed in class. Please review the lecture slides and read the course notes before you start with this homework.

Please complete this week's lab and watch the video before you start to work on the programming part of this homework.

1 Theoretical Part

1.1 Image Formation Model (*10pts*)

Let's say we have the ground truth 6-DOF pose, i.e. orientation $\boldsymbol{\theta}$ (in degrees) and position $\vec{t}$ (in mm), of a VRduino given as

$$\boldsymbol{\theta} = \begin{pmatrix} \theta_x \\ \theta_y \\ \theta_z \end{pmatrix} = \begin{pmatrix} 45^\circ \\ 0^\circ \\ 45^\circ \end{pmatrix}, \quad \vec{t} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} = \begin{pmatrix} 10 \\ 10 \\ -50 \end{pmatrix}.$$

Assume that the lighthouse is located at the origin in world coordinates. We also know that the four photodiodes are mounted on the VRduino on the following positions (specified in mm):

$$p_0 = \begin{pmatrix} -42 \\ 25 \\ 0 \end{pmatrix} \quad p_1 = \begin{pmatrix} 42 \\ 25 \\ 0 \end{pmatrix} \quad p_2 = \begin{pmatrix} 42 \\ -25 \\ 0 \end{pmatrix} \quad p_3 = \begin{pmatrix} -42 \\ -25 \\ 0 \end{pmatrix}.$$

As a Lighthouse base station sweeps its invisible horizontal and vertical laser lines through the room, all four photodiodes would be triggered at a particular time stamp, measured in clock ticks of the microcontroller. Given the parameters above, what are these time stamps for all photodiodes? List both horizontal and vertical time stamp for each photodiode, i.e., 8 values total.

Round the clock ticks to the nearest integer. Assume that the microcontroller runs at 48 MHz. Similar to the lecture and course notes, we define the sequence of rotations from local to world coordinates as yaw-pitch-roll, i.e. $R = R_z(\theta_z) R_x(\theta_x) R_y(\theta_y)$ (see Appendix C of course notes). You can use your favorite tool for the computation (Matlab, Python, etc.) in this problem. There is no need to submit code, but show your derivations/logic in your writeup.

1.2 Robustness of the Inverse Method (*15pts*)

Usually, we do not know the ground truth pose of a VRduino. To compute it from the measured time stamps, we can use the homography method as discussed in class and in the course notes. For this purpose, we need to construct a linear system of equations $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{h}$ and solve it for $\boldsymbol{h}$. This requires the matrix $\boldsymbol{A}$ to be inverted. As you know, the robustness of a linear inverse problem with respect to sensor noise or slight errors in the measurements is defined by the conditioning of the matrix. You can use your favorite tool for the computation (Matlab, Python, etc.) in this problem. There is no need to submit code, but show your derivations/logic in your writeup.

(i) Show the matrix $\boldsymbol{A}$ constructed from the eight measurements you calculated in Section 1.1. (*5pts*)

(ii) Compute the singular values $\sigma_{1,\ldots,8}(\boldsymbol{A})$ and condition number $\kappa(\boldsymbol{A}) = \frac{\sigma_{max}(\boldsymbol{A})}{\sigma_{min}(\boldsymbol{A})}$ of this matrix. Briefly discuss what this number means for the robustness of this inverse problem with respect to noise and measurement errors. Feel free to use Matlab, Python, or whatever other tool you like for computing the singular values. (*10pts*)

1.3 Arranging Photodiodes in 3D (*20pts*)

On the VRduino, all photodiodes are arranged in a planar configuration. While this makes the related calculations a bit easier, it may not be the best approach when robustness and precision of the tracking matter. Indeed, the arrangement of the photodiodes on the controllers and the HMD that the HTC Lighthouse usually tracks is a much

more complex 3D configuration. Let's look at such a 3D photodiode arrangement in more detail! You can use your favorite tool for the computation (Matlab, Python, etc.) in this problem. There is no need to submit code, but show your derivations/logic in your writeup.

(i) What is the minimum number of photodiodes arranged in a non-planar 3D configuration that results in a square or tall matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $m \geq n$ in the linear equation system? How did you come up with that number? (*10pts*)

(ii) Assume that we have the same four planar photodiodes listed in Section 1.1 as well as four additional photodiodes that extrude from that plane and are located at:

$$p_4 = \begin{pmatrix} 0 \\ -25 \\ 10 \end{pmatrix} \quad p_5 = \begin{pmatrix} 0 \\ 25 \\ 10 \end{pmatrix} \quad p_6 = \begin{pmatrix} 0 \\ -25 \\ -10 \end{pmatrix} \quad p_7 = \begin{pmatrix} 0 \\ 25 \\ -10 \end{pmatrix}.$$

On this VRduino, we have measured the following horizontal and vertical time stamps on all 8 photodiodes and converted them into normalized coordinates:

photodiodes	x^n	y^n
p_0	-0.2926	-0.0822
p_1	0.3296	0.9955
p_2	0.6459	0.4924
p_3	0.1919	-0.2940
p_4	0.5138	0.0796
p_5	0.0948	0.4814
p_6	0.3403	0.1154
p_7	-0.0404	0.4376

Using the least square solution $\boldsymbol{h} = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\boldsymbol{b}$ to the linear system, retrieve translation vector $\vec{t}$ and report yaw, pitch, and roll angles in degrees. Round translations and angles (in degrees) to the nearest integer. (*10pts*)

Hint: First derive the linear system, invert it to obtain rotation matrix R and translation vector $\vec{t}$ based on $\boldsymbol{h}$. Then refer to Appendix C of course notes for details on how to calculate yaw, pitch, and roll from R.

Programming Part PDF Deliverables

There are no additional deliverables from the programming part in this homework.

2 Programming Part

In this part of the homework, you will implement homography-based pose estimation with the VRduino and the Lighthouse.

The structure of the starter code is similar to the code in Homework 5. Here is a brief overview:

1. `vrduino.ino`: Initializes all tracking variables and calls tracking functions during each loop iteration.
2. `PoseTracker.cpp`: Manages the pose tracking. Inherits the orientation tracking functionalities from `OrientationTracker` in Homework 5.
3. `PoseMath.cpp`: Implements the math for the pose tracking algorithms.
4. `Lighthouse*.cpp` Handles the photodiode interrupts to record pulse timings from the Lighthouse. These are implemented for you.

Implementation hints:

- You do not need a Lighthouse base station to implement the following tasks. We provide pre-recorded timing data from the Lighthouse so you can implement and test your code off-line. Make sure that `simulateLighthouse = true` in `vrduino.ino` (this should be the default setting), so you are able to work with the pre-recorded data instead of live-captured photodiode timings.
- Before implementing a function, read the documentation in the header file for additional details.
- We provide a framework for unit testing in `TestPose.cpp`. Feel free to add your own tests to this file. Set `test = true` in `vrduino.ino` to run the tests. Each part in this assignment depends on the previous part, so make sure each part works before moving on.
- Use the provided visualizer with the "6D pose" mode selected. See Lab 6 for instructions on how to run it.

The goal of the remaining programming questions is to implement the function `updatePose()` in `PoseTracker.cpp`. Here, you will have to call several other functions, all included in the file `PoseMath.cpp`, with the correct input and output variables. In the following, we will go through each of these functions step by step. You will implement each of the functions in `PoseMath.cpp` and then call them with the correct arguments in `updatePose()`. Before starting the homework, please make sure that the CPU speed of Teensy is set to 48 MHz in `Tools` of Arduino IDE.

After implementing all functions, you can run `render.html` to dive into the VR world with 6-DOF capabilities. As we unfortunately cannot provide the access to the Lighthouse this year, you may not be able to try it with the actual physical measurements (if you have it, you can!). However, you can still feel the functionality with the pre-recorded measurements. To match the coordinate system between the lighthouse and the virtual world, reset the orientation tracking by pressing `r` in Terminal while holding your headset. If your pose estimation is jittery, you can perform denoising by temporal filtering, which is already provided in the starter code. To increase and decrease the denoising amount, press `2` and `3` in the browser window.

2.1 Clock Ticks to Normalized 2D Coordinates (*10pts*)

In every loop iteration, we query the photodiodes to see if new timing data is available for all 4 diodes. If new data is available, the `clockTicks` variable in `PoseTracker` is populated with these timings. We have implemented this functionality for you.

Your first task is simple: implement the function `convertTicksTo2DPositions()` in `PoseMath.cpp`. In this function, you should first convert the raw clock ticks measured for horizontal and vertical sweep directions to azimuth and elevation angles, respectively. Then, you convert this angle to normalized coordinates x^n and y^n on an imaginary plane at unit distance away from the base station. Follow the equations of lecture 11. Remember that division of two integers in C returns another integer.

Once you have implemented `convertTicksTo2DPositions()`, call it with the correct arguments in `updatePose()` in `PoseTracker.cpp`. Remember that the clock ticks are available in the `clockTicks` array (sorted in the following order: `photodiode0.x`, `photodiode0.y`, `photodiode1.x`, ...) and you want to write the normalized coordinates into the variable `position2D` (same order as `clockTicks`), so these should be the arguments for `convertTicksTo2DPositions()` in `updatePose()`.

You can run the JavaScript visualizer provided to you in the starter code to visualize your normalized coordinates (upper left browser sub-window with black background, should look like Figure 1 but without labels). Make sure you see these coordinates update correctly in the browser window before continuing.

Note that the main file `vrduino.ino` will automatically stream values of the `position2D` array with the prefix 'PD' to the serial port. This is how the JavaScript visualizer reads and renders them, but you don't have to do anything for this.

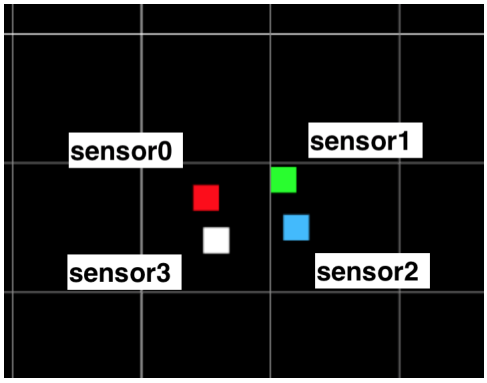

Figure 1: *Visualization of the normalized 2D coordinates x^n and y^n for all 4 photodiodes.*

2.2 Populate Matrix A (*10pts*)

Now, set up the matrix A as discussed in the lecture. For this purpose, implement the function `formA()` in `PoseMath.cpp`. Once you have implemented this function, call it with the appropriate arguments in `updatePose()`.

Remember that A describes the linear mapping between the 8 homography values h that we need to estimate and the normalized coordinates of the photodiode timings, which you computed in the last subsection. The physical reference positions of the photodiodes on the VRduino board are listed in the lecture slides and provided in the starter code in the array `positionRef` (defined in `PoseTracker.h`), so this should be one of the arguments for `formA()`. Also, create an empty 8×8 array in `updatePose()` to pass in as the output variable A. Keep in mind that you can access 2D arrays in C/C++ as `A[i][j]`, where i is the row index and j is the column index.

2.3 Solve for h (*10pts*)

With the matrix A and the vector of measurements b in place, we have all the information we need to solve for the homography values h. The measurements, called b in the lecture slides, are just your array `position2D` that you have already computed above. Implement the function `solveForH()` in `PoseMath.cpp` to compute h. Make use of the `MatrixMath` library perform matrix inversions and multiplications. Check if `Matrix.Invert()` returns 0, which indicates that the matrix is not invertible or is ill-conditioned. This is usually only the case if you made a mistake somewhere else before calling `Matrix.Invert()`. Make sure to return `false` in `solveForH()` if `Matrix.Invert()` fails. If the inversion is successful, multiply the matrix inverse with `b`, storing the result in the output variable `hOut`.

Now call `solveForH()` in `updatePose()` with the appropriate arguments; `updatePose()` should return 0 if `solveForH()` returns false.

2.4 Get Rotation Matrix and Translation from h (*15pts*)

Next, you will implement the function `getRtFromH()` in `PoseMath.cpp` to estimate the rotation and translation from the homography matrix. Once implemented, call `getRtFromH()` in `updatePose()`. The input to `getRtFromH()` is the array of 8 homography values h and the output should be a 3×3 rotation matrix and a 3-element translation vector. In `updatePose()`, make sure to initialize the latter two variables and pass them into `getRtFromH()` where they should be populated with the correct values. The output translation vector passed into `getRtFromH()` by `updatePose()` should be called `position` – please use this variable, because we already implemented the code that streams this variable to JavaScript.

The 8 elements of h have so far only been estimated up to a scale factor s (i.e, we assumed that $h_9 = 1$). So in `getRtFromH()`, you want to estimate the scale factor s first, then use that to calculate the translation vector from $h_{3,6,9}$ and then estimate the 3×3 rotation matrix from s and $h_{1,2,4,5,7,8}$. Make sure that the columns of the rotation matrix are orthogonal and unit length, as discussed in lecture.

2.5 Convert Rotation Matrix to Quaternion (*10pts*)

Now that we have estimated the 3×3 rotation matrix representing the orientation of the VRduino with respect to the base station, we want to convert it to a rotation quaternion before we stream it via the serial port to the host computer for rendering. For this purpose, implement the function `getQuaternionFromRotationMatrix()` in `PoseMath.cpp`. Refer to the appendices of this week's course notes for all relevant math. Call the function `getQuaternionFromRotationMatrix()` in `updatePose()` with the rotation matrix you computed in the previous subsection and write the resulting quaternion into the variable `quaternionHm`.

Questions?

First, Google it! It is a good habit to use the Internet to answer your question. For 99% of all your questions, the answer is easier found online than asking us. If you cannot figure it out this way, post on piazza or come to office hours.