ECE 276A Project 1: Orientation Tracking

Jean Gorby Calicdan
Department of Electrical and Computer Engineering
University of California, San Diego
La Jolla, United States
jcalicda@ucsd.edu

***Abstract*—This report presents the implementation of an orientation tracking system utilizing Inertial Measurement Unit (IMU) data and the construction of panoramic images using camera data. A projected gradient descent algorithm was used to estimate the orientation of a rotating body, leveraging IMU gyroscope and accelerometer readings. The accuracy of the estimated orientation was evaluated using ground-truth VICON data, and the computed trajectory was applied to generate panoramic images.**

***Index Terms*—IMU, orientation, projected gradient descent, optimization**

I. Introduction

Accurate orientation tracking is fundamental for applications in robotics and motion estimation. Monitoring orientation enables tasks such as navigation, object tracking, and augmented reality. This report addresses the problem of estimating the orientation of a rotating body using IMU data and applies these estimates to construct panoramic images from camera data. The goal is to leverage both motion models and optimization techniques to achieve precise orientation estimation.

II. Problem Formulation

A. Orientation Tracking

The task is to estimate the orientation q_t of a body over time, given IMU readings (angular velocity and linear acceleration). The problem can be mathematically defined with these models:

Motion model:

$$q_{t+1} = q_t \circ \exp\left([0, \frac{\tau_t \omega_t}{2}]\right)$$

where **exp** is the quaternion exponential map.

Observation model:

$$[0, a_t] = q_t^{-1} \circ [0, 0, 0, -g] \circ q_t$$

where **g** is the earth's gravity.

The task is to find the quaternion trajectory $q_{1:T}$ that minimizes a cost function combining errors from the motion and observation models. This can be done by minimizing the following cost function:

$$c(q_{1:T}) = \frac{1}{2} \sum_{t=0}^{T-1} \left\| 2 \log\left(q_{t+1}^{-1} \circ f(q_t, \tau_t \omega_t)\right) \right\|^2 + \frac{1}{2} \sum_{t=1}^{T} \left\| [0, a_t] - h(q_t) \right\|^2$$

subject to $\|q_t\| = 1$.

B. Panorama

Given the orientation trajectory over time, q_t, and images captured by a camera, stitch together the images into a panorama.

III. Technical Approach

A. IMU Calibration

Bias Calculation: Static portions of IMU data were identified using timestamps, during which the IMU remained stationary. Biases for the gyroscope and accelerometer were computed as the mean of the sensor readings over this interval.

$$\text{Bias}_{\text{gyro}} = \frac{1}{N} \sum_{i=1}^{N} \omega_{\text{raw},i}, \quad \text{Bias}_{\text{accel}} = \frac{1}{N} \sum_{i=1}^{N} a_{\text{raw},i}$$

Expected Static Values: During the static period, the accelerometer readings were expected to align with the direction of gravity in the body frame. Assuming the sensor is stationary and oriented upright, the accelerometer should measure $[0, 0, g]$, where g is the magnitude of gravity (approximately $9.81 m/s^2$). To normalize for scale, the IMU data was expected to output $[0, 0, 1]$. Deviations from this value indicated the presence of biases or miscalibration in the accelerometer.

Scale Factor Adjustment: Calibration scale factors were determined using sensor datasheets and reference voltages to convert raw sensor readings into physical units (e.g., radians per second for gyroscopes, for accelerometers).

B. Motion Model Implementation

The motion model was implemented using quaternion mathematics. The gyroscope readings were integrated over time to compute incremental rotations using the exponential map:

$$\Delta q_t = \exp\left([0, \frac{\omega_t \Delta t}{2}]\right)$$

Successive orientations were calculated via quaternion multiplication:

$$q_{t+1} = q_t \circ \Delta q_t$$

All quaternion operations were implemented rather than using library functions to do the same. This was done in order to allow for vectorization of operations in order to speed up run time. In these functions, helper library functions from transformds3d and jax were used for efficient numerical computation.

C. Solving the Optimization Problem

To refine orientation estimates, we employed an iterative gradient descent approach:

1) Initialize: Set an initial estimate for the quaternion trajectory $q_{1:T}$.
   - For the first time step, this can actually be initialized, given that we know $q_0 = [1, 0, 0, 0]$. We can roll out the entire motion model to initialize $q_{1:T}$.
2) Compute Gradient: Evaluate the gradient of the cost function with respect to each quaternion.

$$\nabla_q c(q_{1:T}) = \frac{\partial}{\partial q}\left(\frac{1}{2}\left\|2\log(q_{t+1}^{-1} \circ f(q_t, \tau_t\omega_t))\right\|^2 + \frac{1}{2}\left\|[0, a_t] - h(q_t)\right\|^2\right)$$

   - Rather than solving the gradient by hand and implementing the resulting function, we use jax.grad function to utilize their numerical differentiation methods for efficiently and reliabitity.
3) Update Quaternions: Adjust the quaternion values iteratively using:

$$q_{1:T}^{(t+1)} = q_{1:T}^{(t)} - \alpha\nabla_q c(q_{1:T})$$

   where α is the learning rate
4) Normalize: Ensure that each quaternion remains unit-norm by normalizing after each update. This ensures that we are minimizing within the given constraint which transforms the gradient descent into a projected gradient descent.

$$q_{1:T}^{(t+1)} = \Pi\left(q_{1:T}^{(t)} - \alpha\nabla_q c(q_{1:T})\right)$$

5) Repeat: Iterate until convergence or end of iteration count.

Tuning hyperparameters was a method used within this step as well in order to find the appropriate learning rate and max number of iterations.

D. Panorama Construction

The panorama construction process was implemented by projecting camera images onto a spherical surface and then transforming those coordinates to a cylindrical panorama for unwrapping.

- Projection onto a Sphere:
  1) For each pixel in the image, compute the spherical coordinates (λ, ϕ) using the pixel's row and column indices along with the camera's horizontal $(60°)$ and vertical $(45°)$ fields of view. The depth is assumed to be 1m.
  2) Convert spherical coordinates (λ, ϕ) to Cartesian coordinates assuming unit depth where

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos(\phi)\cos(\lambda) \\ \cos(\phi)\sin(\lambda) \\ \sin(\phi) \end{bmatrix}$$

- Rotate the Cartesian coordinates to the world frame using the camera-to-world rotation matrix R.

$$\begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} = R \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix}$$

- Projection to a Cylinder:
  1) Transform the Cartesian coordinates back into spherical coordinates.
  2) Map the spherical coordinates onto a cylinder where the longitude (λ) is along the cylinder's circumference and the latitude (ϕ) determines the height.
  3) Unwrap the cylinder into a rectangular image with a width of 2π radians and height π radians. This results in a panorama that has a 2:1 aspect ratio. In code, this was roughly 2000x1000 pixels.

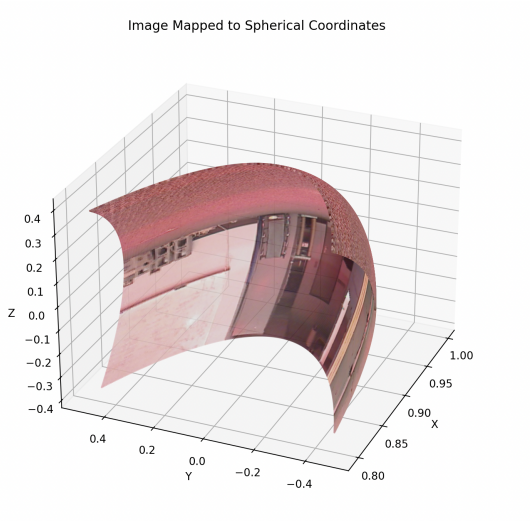

Fig. 1. Example projection of one image into a sphere

IV. Results

A. Orientation Estimation

See next page

B. Panorama Construction

See next page

V. Discussion

The project involved iterative development, debugging, and refinement to achieve the final results. Several challenges were encountered and resolved during the process:

A. Calibration

In the early stages, figuring out the static period of the robot was a tricky problem. I originally had some code which dynamically looked for static periods in the VICON data independently of the data. This was done by looking at the rotation matrices and looking the changes between consecutive timesteps. Logically, this made sense but it was rather tricky

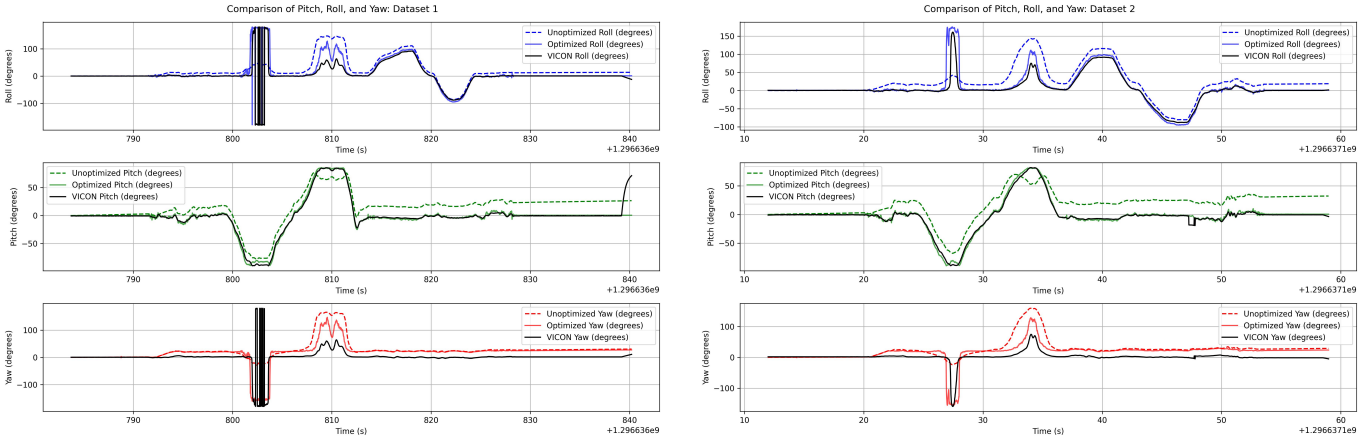

Fig. 2. Orientation Comparisons for Datasets 1 and 2

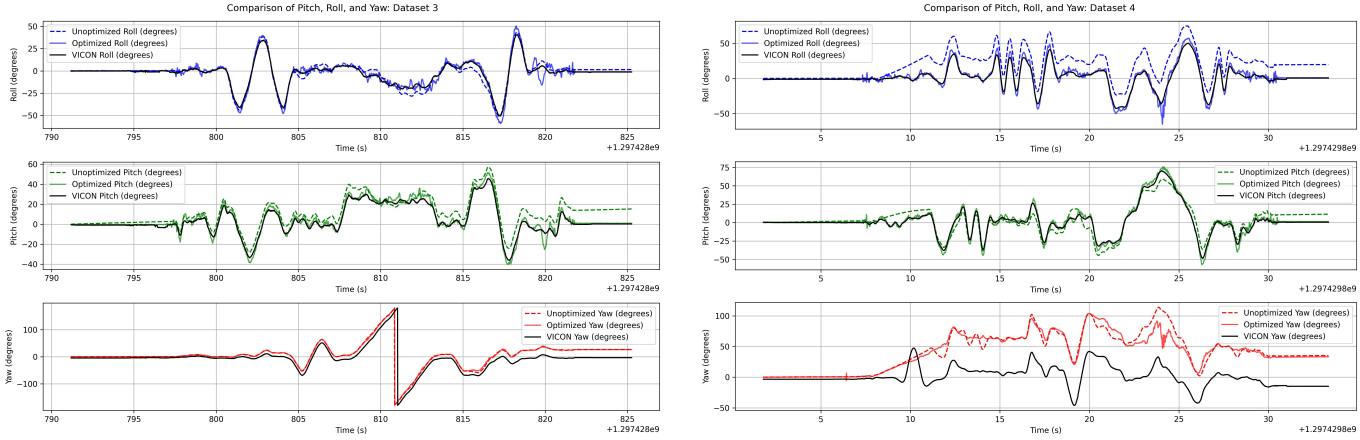

Fig. 3. Orientation Comparisons for Datasets 3 and 4

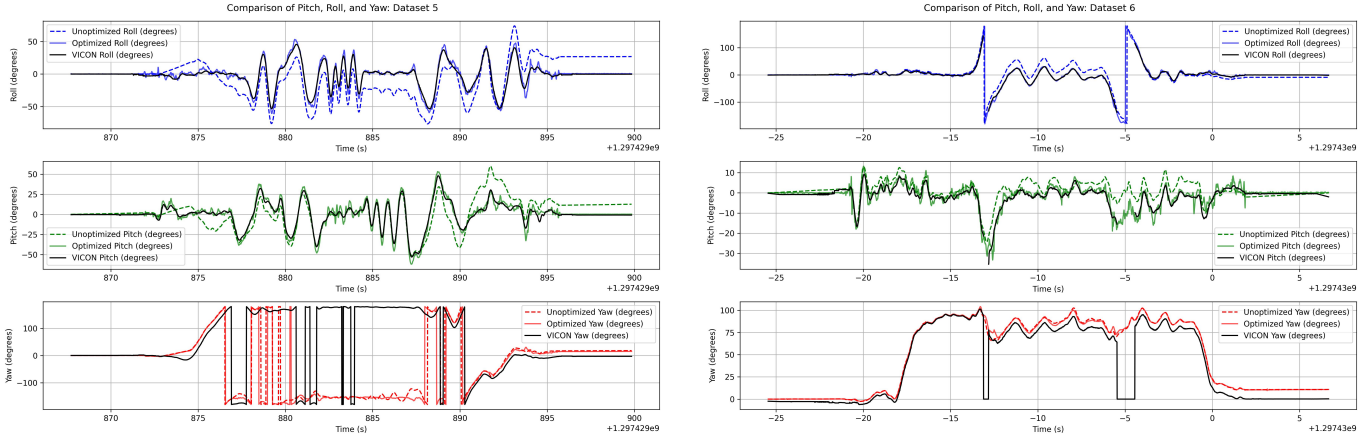

Fig. 4. Orientation Comparisons for Datasets 5 and 6

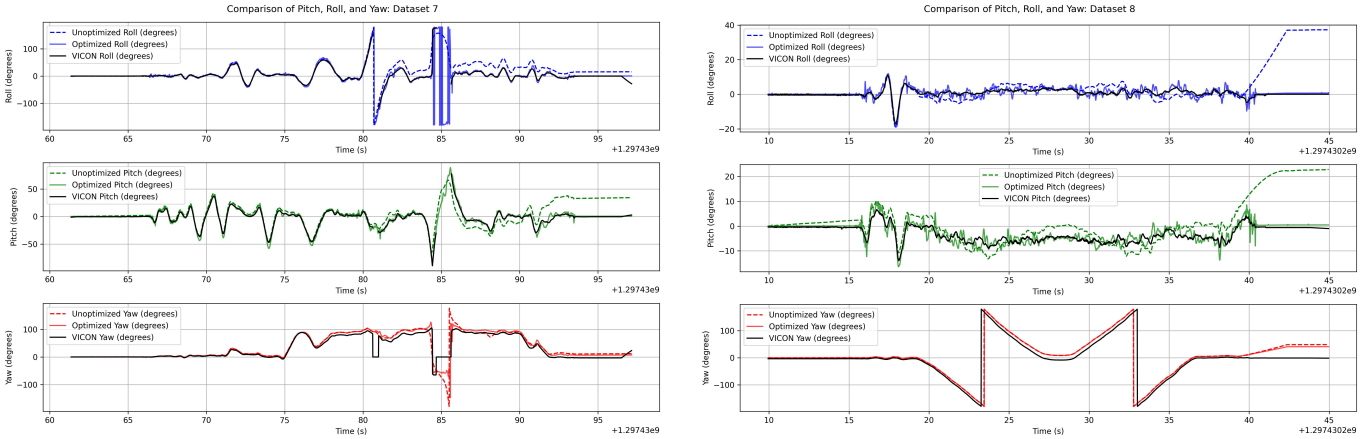

Fig. 5. Orientation Comparisons for Datasets 7 and 8

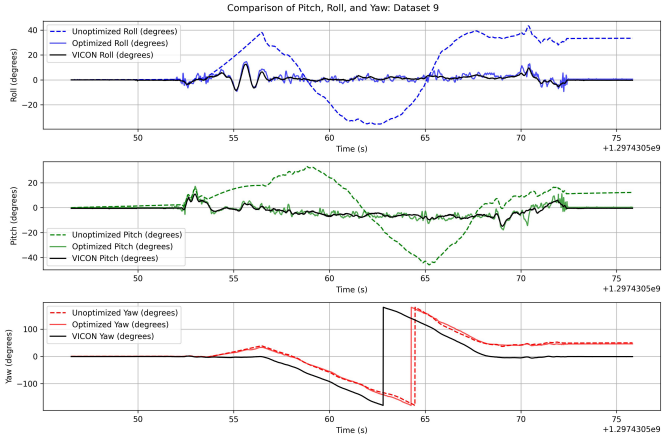

Fig. 6. Orientation Comparisons for Datasets 9

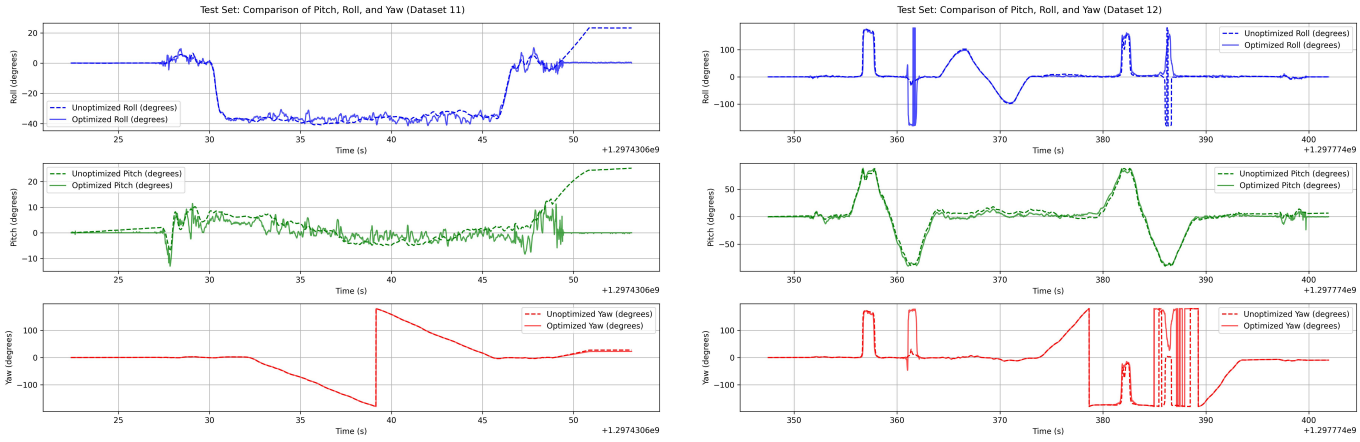

Fig. 7. Orientation Comparisons for Test Datasets 10 and 11

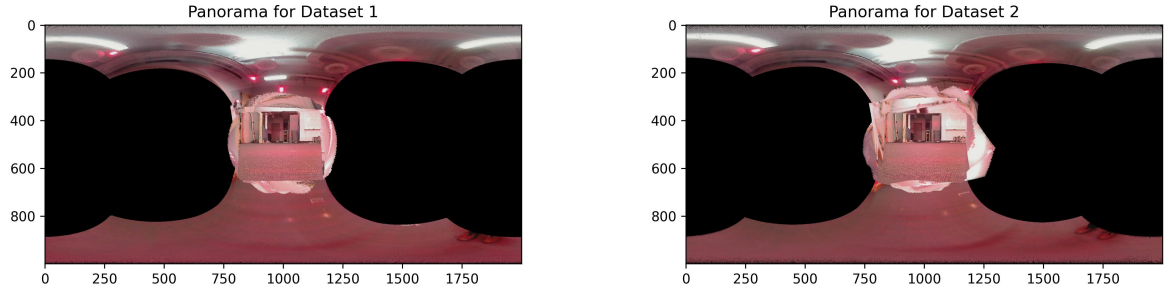

Fig. 8. Panorama for Datasets 1 and 2

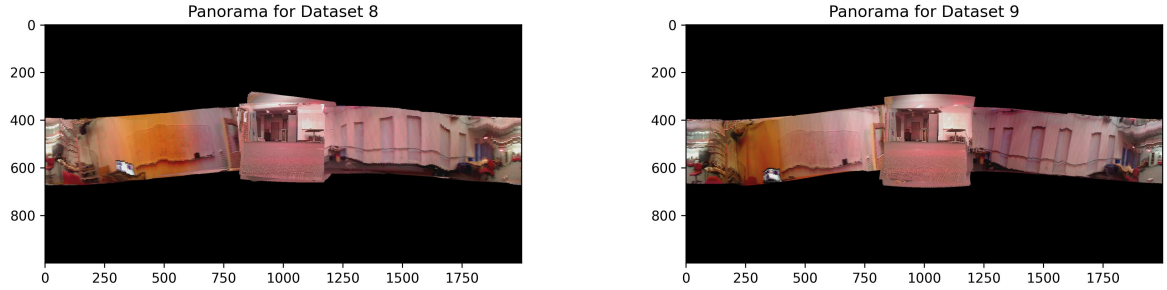

Fig. 9. Panorama for Datasets 8 and 9

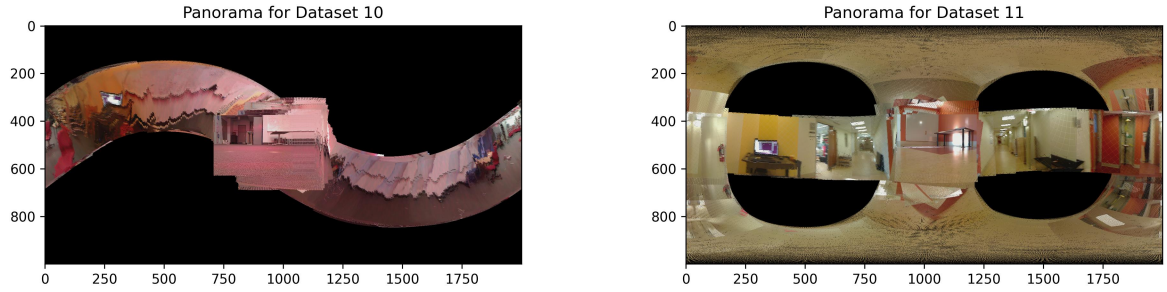

Fig. 10. Panorama for Test Datasets 10 and 11

to implement especially since I didn't understand how the timestamps worked in the data. Distinguising what the time index t meant and its difference from the time difference τ_t was difficult for me initially. For this reason, I decided to change the calibration logic to simply look at the first 500 timestamps. This became the heuristic approach by looking at the plots of all datasets and the first 500 timestamps of each data looked static enough.

In the later stages, when the gradient descent code was implemented, it looked like the results looked reasonable. One common issue from all plots, however, was that the yaw almost always looked zero. There were also very noisy parts of the estimation which can be interpreted as moments in which the orientation of the robot cannot be determined. Here is an example of this:

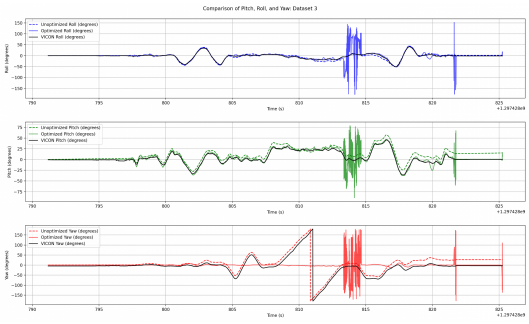

Fig. 11. Bad early example of optimized trajectories

Several debugging tips were taken in order to find the root cause of the problem. It turns out, the problem was with calibration. In looking at the calibrated acceleration values, the shape looked correct but the yaw was offset to zero. This discrepancy in calibration lead to the issues that can be seen figure 11.

B. Quaternions and Optimization

Implementing the quaternion operations and gradient descent functions were relatively simple. What wasn't simple, however, was writing it in a way that was efficient. During the first iterations of the code, a single iteration of the gradient descent algorithm was very slow and results could not even be seen. The solution to this problem was to re-implement the quaternion operations in a way that accepted stacked vectors and eliminate any for loops. This meant that any logic used when multiplying matrices or editing matrix elements had to be done in a way that had O(1) computation complexity. Jax and its implementation of Numpy helped a lot as many of their operations understood vectorization inherently.

In general, the results look good. It struggles somewhat when encountering massic spikes in rotations. For the most part, most of the roll and pitch looks good. For some rotations, it is a little noise when there are sudden changes in rotation but the general trajectory is traced really well.

C. Panorama

My original implementation of the panorama wasn't perfect. The images for datasets 8 and 9 seem reasonable as the relevant details can be seen and interpreted. The images for datasets 1 and 2, however, aren't great as the artifacts in the images aren't quite obvious. I believe the reason for reasonable datasets 8 and 9 panoramas stem from the fact that their rotations are rather simple. More complicated rotations fail with the current codebase. One obvious error seem to be the way certain rotations are handled within the projection. By looking at datasets 1 and 2 and their panoramas, the bottom side of the image seem to contain what looks like would belong to the top of the room. The solution to this was to take the transpose of the rotation matrix which heuristically makes sense due to the problems pointed out above.

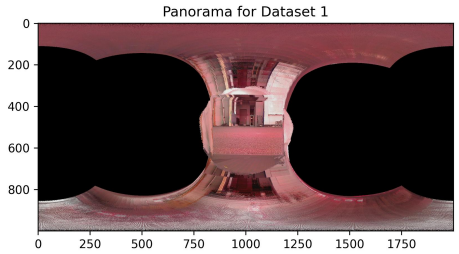

Fig. 12. Bad example of dataset 1 panorama

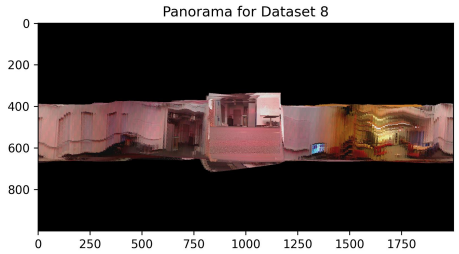

Fig. 13. Bad example of dataset 8 panorama

REFERENCES

[1] ECE 276A Course Materials
[2] Sensor Datasheets (IMU, Gyroscopes, Accelerometers)
[3] Transforms3D Library
[4] Jax Library