Home Assignment -Dan Weil

When solving the home assignment of 6D brick pose detection from RGBD images, I aimed to demonstrate my approach to tackling new vision challenges. I focused on creating a baseline solution that runs efficiently and designing a modular framework for ongoing algorithm improvements. Additionally, I outlined areas for improvement, along with resources estimation and time requirements. The final solution was implemented in C++ with Python prototyping and runs in approximately 1 second on my old CPU computer. Check out README.md for running instructions.

**Background:**

First, I conducted a thorough literature review to understand existing solutions for 6D object detection. Most recent notable state of the art projects includes NVIDIA's [BundleSDF](https://arxiv.org/abs/2303.14158) (2023) , label free model which also handles 3D reconstruction and tracking. It has ambiguous licenses, but it is still beneficial to understand the current direction of research in the field.

If we go back in time , to one of the first deep learning solutions for RGBD data is [PoseCNN](https://arxiv.org/abs/1711.00199) (2017). Although it was trained on specific objects and would require retraining for our purposes, it provided valuable insights. Inspired by PoseCNN, I devised the following pipeline:

1. **Object Segmentation (2D)**: Detect the pixels in the 2D image on which the brick lies.
2. **Pose Estimation**: Detect the rotation and translation of the segmented data using camera world coordinates.

**Algorithm building:**

**Object Segmentation (2D)**: Initially, I tried using classic vision filters like [Canny](https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html) and [Contour](https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html), but they didn't yield good results. I also experimented with 2D CNNs for this task, but wall bricks don't appear in well-known datasets like COCO or ImageNet. As a baseline, I used Meta’s open-source Segment Anything (actually, version 2 released this month), which takes a point and an RGB image as input and returns a mask. This model is a bit overkill due to its large capacity, especially the encoder. Therefore, I modified the original network to better suit our needs and to run faster. To eliminate the need for PyTorch, I created an ONNX export that can also resize the input. You can see the changes in the forked repository I [made](https://github.com/danweil24/seg_anything_1_onnx).

**Brick 2D Segmentation Algorithm Outline:**

1. **Detect point on the centered brick**:
   * I masked the wall region and took the most centered point with a small margin to avoid edges. I assumed minimum and maximum depth of the wall, which makes sense as we want the robot to be near the brick.
2. **Preprocess image and run inference**:
   * I imported the models to ONNX for each input size, creating a few models (1024, 704, 640). I used 640 as a trade-off for runtime but kept the rest as options.
3. **Post-processing and noise cleaning**:
   * When I reduced the network, I encountered some noise (adjustment bricks were added). To clean it, I used connected component filtering.

**Assumptions and Place for Improvement:**

To avoid the heuristic in finding the point and reduce runtime, I recommend simply training a [UNet](https://github.com/milesial/Pytorch-UNet) model for 100 epochs on wall data. We can use the Segment model for tagging. This task is estimated to take about two weeks and would incur the cost of GPU usage. Run time can be improved both in preprocessing and network execution, depending on the hardware we want it to run.

.

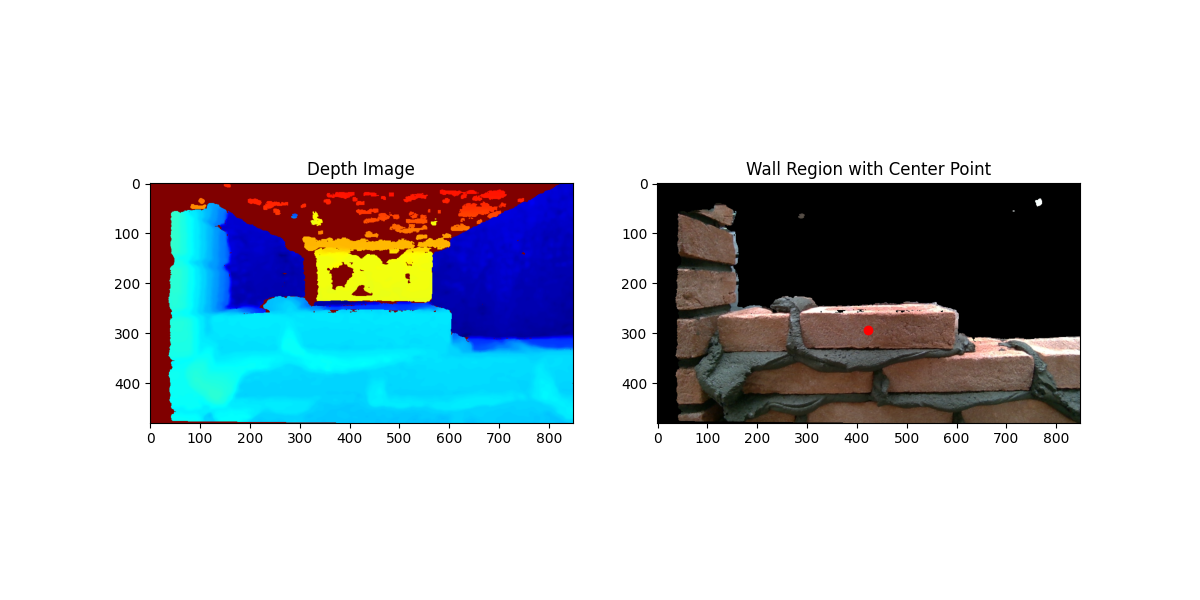


Fig1 : Detecting point on the center brick by heuristics

\



Fig2 : Network output, we can see the tradeoffs in runtime between models and performance, and filter results filtering

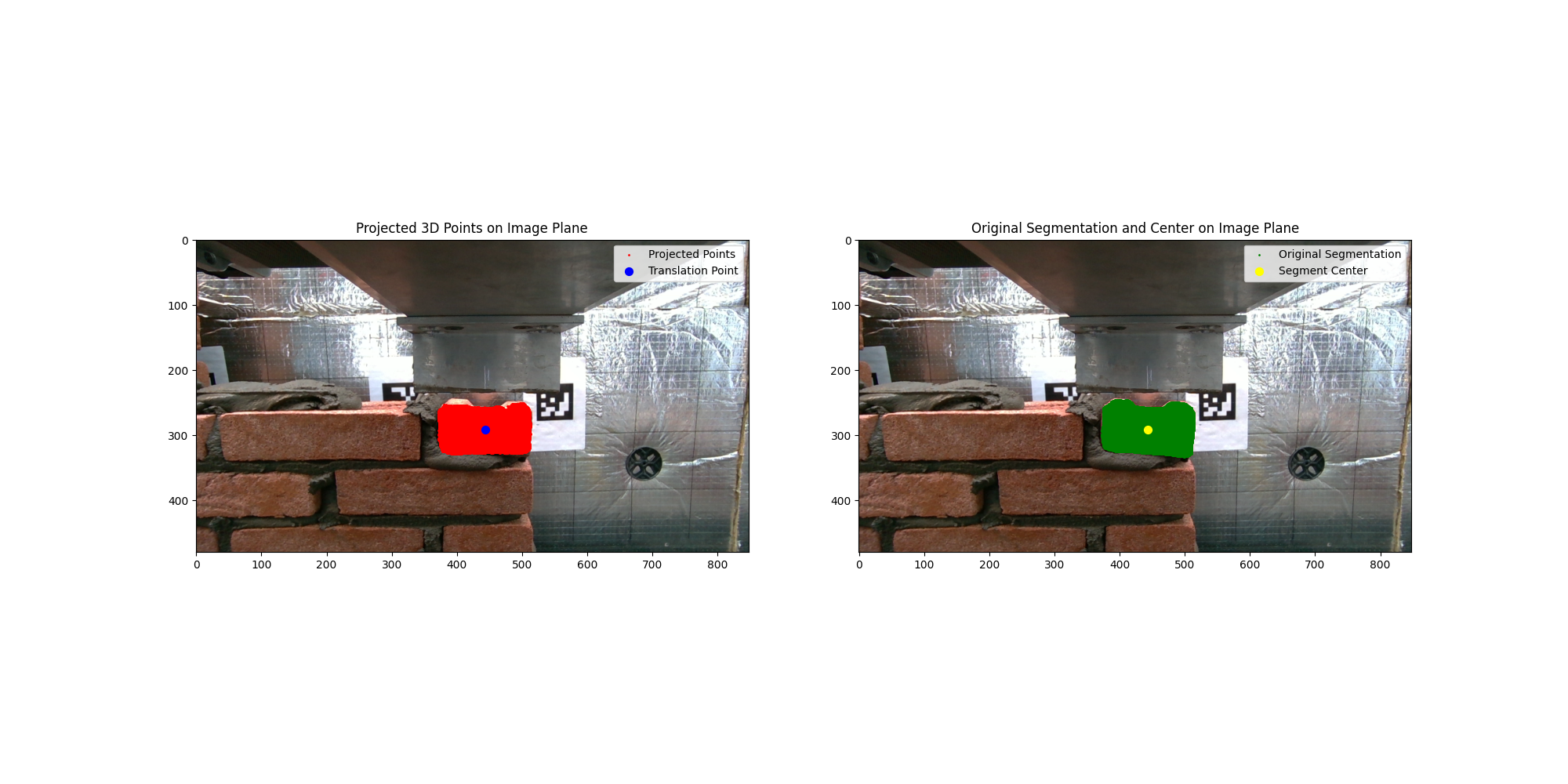
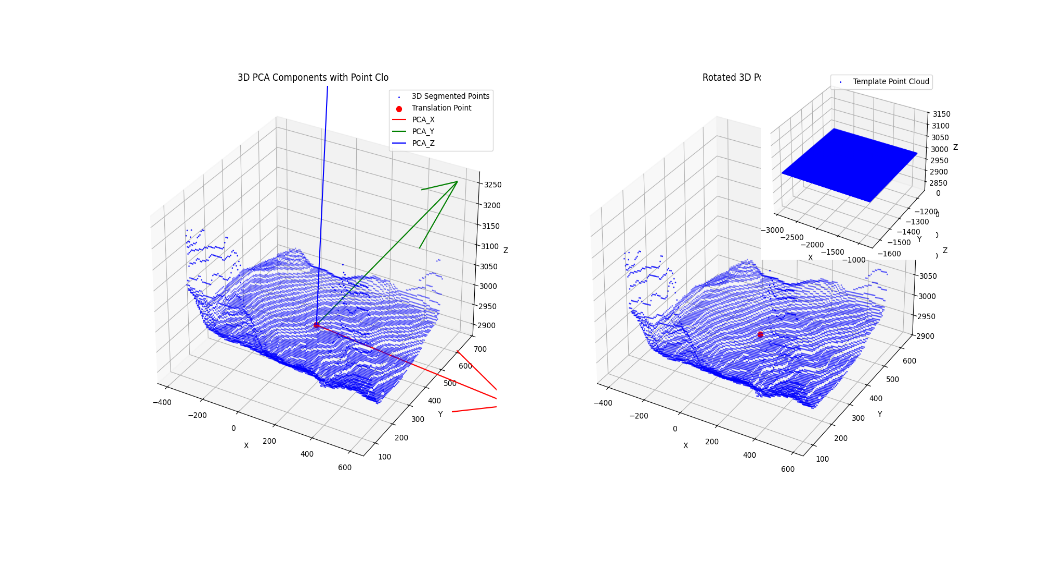
**Pose Estimation:**

After determining the 2D segmentation, I performed the following steps:

1. **Get 3D point cloud**:
   * Using the camera intrinsics and depth information, I generated the 3D point cloud of the object. To clean depth noise from the sensor and remove outlier points, I applied 1 standard deviation filtering from the center.
2. **Find PCA components and align to axis**:
   * I performed Principal Component Analysis (PCA) on the point cloud and aligned it to the principal axes.
3. **Assign to each axis its corresponding brick dimension**:
   * Using the known size of the brick with a tolerance coefficient, I assigned each principal axis to the corresponding brick dimension.
4. **Get the final rotation:**
   * I determined the final rotation by aligning the principal axes to the desired world coordinates. The translation is derived from the centered point cloud.

**Assumptions and Place for Improvement**:

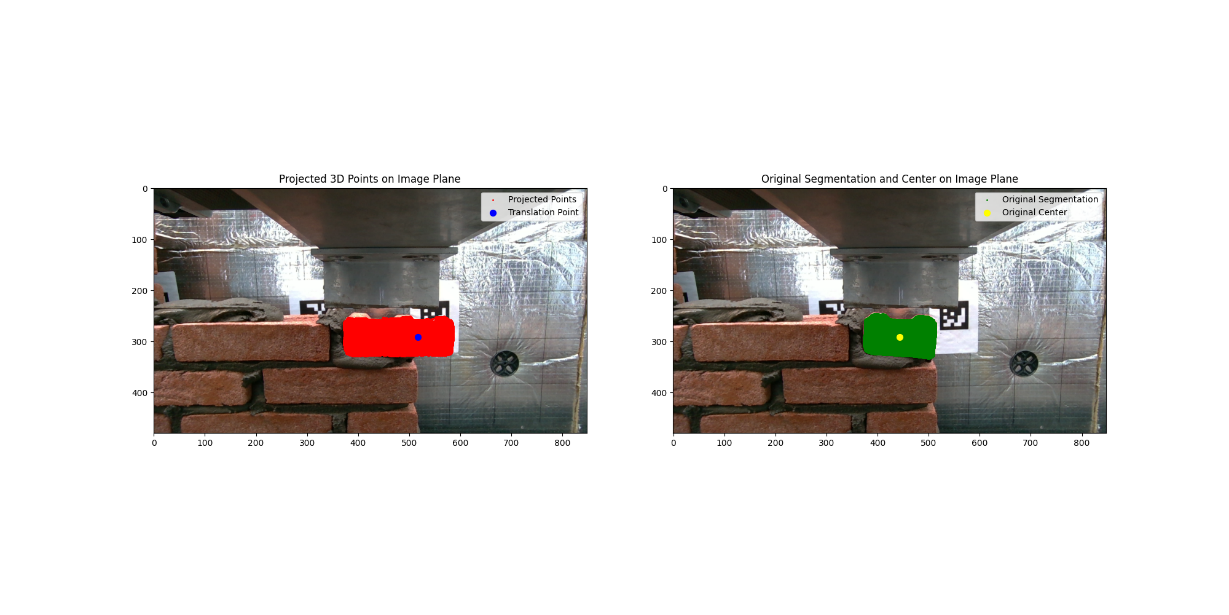
* Camera distortion : I didn’t hand the distortion for time matters and uncertainty of the sensor ,we can enhance accuracy with calibration.
* For Step 3: The algorithm works well when at least 80% of two sides of the brick are visible. It becomes harder if the brick is partially covered. In such cases, I still output the rotation of the visible portion. To solve this cold start situation we can use heuristics ,s.t the aspect ratios, but to be certain , I think it better to ask the robot to change POV.
* Assuming the brick size might be problematic for robustness, but I kept it as a parameter to allow for adjustments.



**Figure 3**: in the left plot ,point cloud with PCA before rotation to camera axis, near the point cloud is rotated and compared to the template, showing that the x-axis fits the short side.

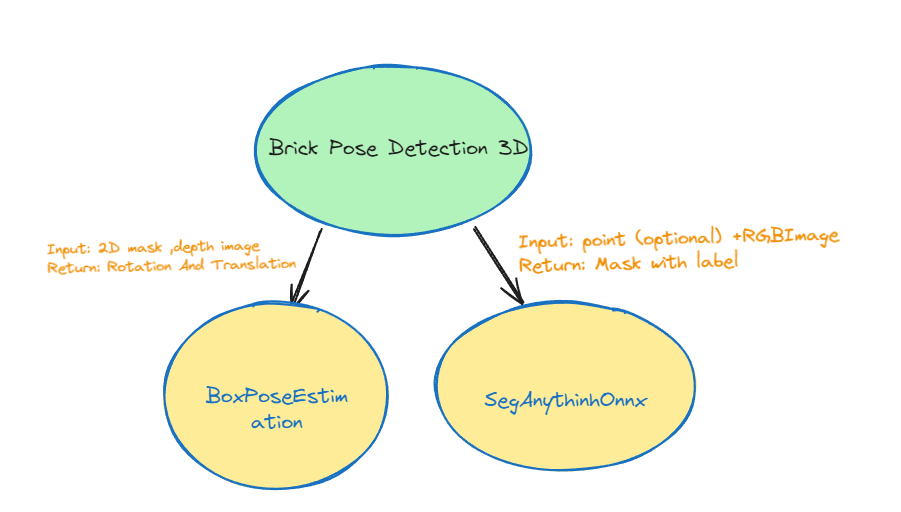
The right side visualizes step 2, rotating the point cloud to align with the image axes.

The lower image shows step 3 and final, using a debugging tool to test if the axes are misaligned by artificially adding points to the point cloud and rotating them according to the result.



**Code and Design Implementation:**

As a baseline and for plotting , I created a Python repository, which can be found (PROJECT\_DIR)\PythonVersion. For efficiency and to run on the target device, I created a C++ version using the Visual Studio compiler on Windows. I used known dependencies that have ARM implementations as well, including [OpenCV](https://opencv.org/), [ONNX Runtime](https://onnxruntime.ai/), [Eigen,](https://eigen.tuxfamily.org/index.php?title=Main_Page) and [Crow](https://crowcpp.org/master/) for HTTP. Check the (PROJECT\_DIR)\README for instructions on how to activate.



**Figure 4**: General design of the system: The main class BrickPoseDetection3D delegates to two independent modules, one for 2D detection and one for pose estimation. The segmentation model inherits from the abstract SegDetection class, allowing for robust changes to different models. Similarly, BoxPoseEstimation specifies that it’s for a box, while its parent class can be extended to other objects.

A screen shot of a computer

Description automatically generatedTo avoid hard-coded definitions such as the working unit or the order of the brick dimensions. For-instance In the task, it was stated that X represents length, Y represents depth, and Z represents height. However, it was more intuitive in the camera coordinates to keep X as length, Y as height, and Z as depth. This is a hyperparameter that can be easily changed. Here are some assumptions I made along the way in one structure.