Exit Finder algorithm:

input: vector of 3d points that orbslam got

output: vector of 2d points (symbolizing (x,z)) of exit points

steps of algorithm:

1) remove y coordinate from input vector (because the height from the ground of the points is not ver relevant)

2) of those points, round their coordinates to two digits after the decimal points and remove duplicates

3) do IPCL on points.

What's IPCL?

To know that we first need to know what's PCL:

PCL (Pseudo-Clustering) is an algorithm we invented that simply removes points from a vector. A point is removed iff the point doesn't have enough points close to it currently. We remove all the points that should be removed all at once at the end of the algorithm, thus, making previous removals not affect other points. This creates a vector of the original points but only ones that are dense in their region, hence the name.

Note that we can do the algorithm PCL multiple times, and That's IPCL (Iterative-PCL)!

4) do ransac 6 times, each time remove points close to the line found. Then, restore to the points from the stage before. Note: we find 6 lines due to the shape of the exit, it creates two more lines.

What's RANSAC?

RANSAC (Random sample consensus) is a known algorithm that determines a line that corresponds well with the point data by sampling 2 points a lot of times, passing a line through them and counting the amounts of points close to the line. After randomly sampling multiple times, we return the line with most points close to it.

5) of the 6 lines, find 4 that are the best.

How do we know which lines are the best? By checking which 4 has the biggest score.

How do we calculate a line's score? Get the points that are close to the line, do RPCL (Reverse-PCL: Remember PCL? Just get the compliment list of points to PCL!) on them and count the number of points.

6) From the 4 lines, we construct a quadrilateral which represents the 4 corners of the room.

7) Create a list of evenly distributed points on the circumference of the quadrilateral. remove points from the list that have too many close points from the list after step 3.

8) From the list before, count for each side of the quadrilateral how many points it still has on it. Now for each corner of the quadrilateral, sum the count of the 2 sides that intersect it. The corner with largest sum is the exit point and will be returned inside a vector.

Other attempts:

DBSCAN: After getting the room shape, remove from the point vector all points inside the room (or on its walls), after that do DBSCAN (an algorithm that finds point clusters), return center of clusters.

By distance: return the corner that is of furthest distance from center of room (due to the inward nature of the exit, the center should be further away from the real exit).

By angles: find the vertex in the room that makes the shape concave, and return it.

Different stage 5s: there were other attempts to score the lines in a different manner: distance of 2 furthest points, counting number of close points (but without RPCL) and more.

Scanning algorithm:

The Scanning algorithm is made of two faces, an “initial” scan, and a “wall” scan:

1) Initial scan: While not rotated a complete circle:
    a. go a bit back to get a wider view range
    b. go a bit to the right and look a bit to the right (cw), reverse by ccw and going left
    c. go a bit to the left and look a bit to the left (ccw), reverse by cw and going right
    d. go a bit to the front (where the current iteration started)
    e. look a bit to the right (cw)
2) Wall scan: for each wall:
    a. go to the wall by rotating to it and moving forward until hitting (done by checking if ORB-SLAM still sees key points) the wall, and then going back a bit
    b. circular scan (while not rotated a full circle: rotate a bit)

Other attempts:

A randomized version:

```
While not done
    a. go to a random location (by choosing a random movement command)
    b. circular _scan()
```

A retry mechanism (We found our current algorithm to be good enough to not require that):

> For every successfully command executed, append it to a commands vector
>
> If after executing a command the current location or orientation is unknown (soft fail), reverse it and re-execute it but slower
>
> After a certain threshold of soft fails, execute the reverse of the last successful command (stored in the vector)

<u>**Navigation algorithm:**</u>

Given starting point and target point, navigate to the target.

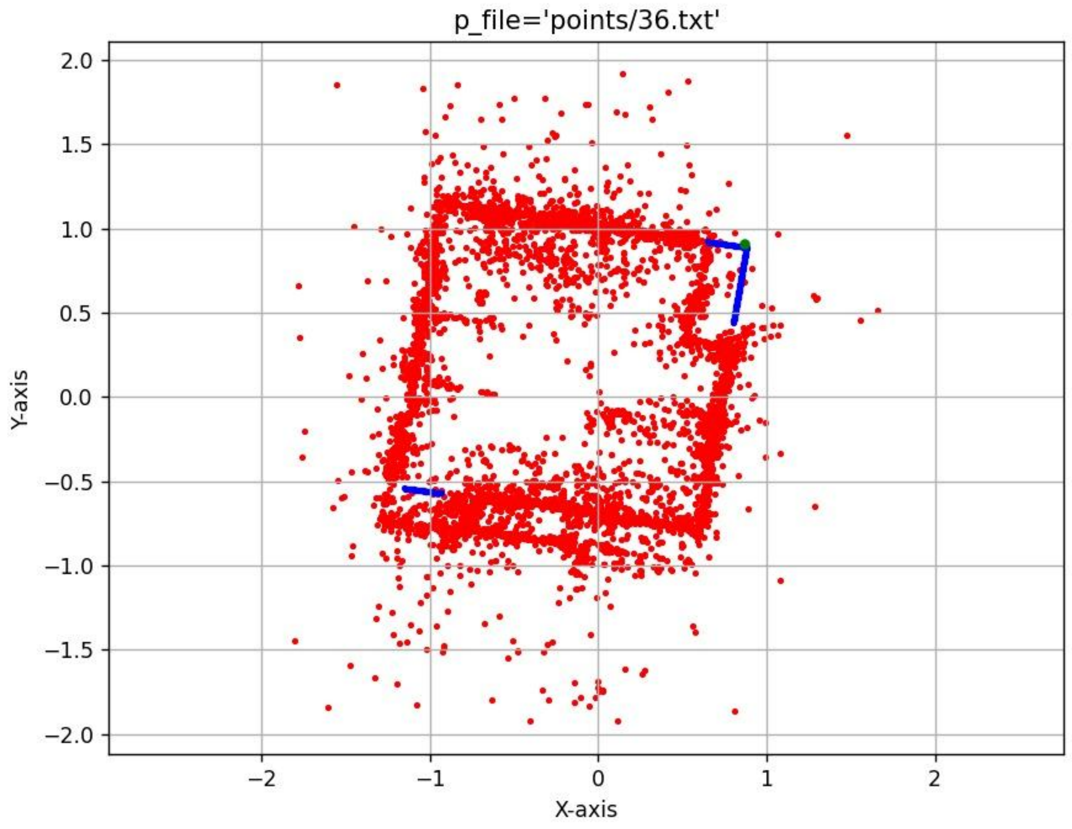
1. Transform the start (S) and target (T) points into 2d (since we ignore height)
2. Get current orientation and transform it to a current direction vector
3. The direction vector we want to move in is target vector – start vector ($\overrightarrow{ST}$)
4. Let P be a point that S sees (S + direction vector)
5. The angle diff is calculated using the formula: $\cos(\theta) = \frac{\overrightarrow{SP} \cdot \overrightarrow{ST}}{|\overrightarrow{SP}| \cdot |\overrightarrow{ST}|}$
6. If the cross product of the vectors ($\overrightarrow{SP} \times \overrightarrow{ST}$) is positive we rotate ccw, otherwise, rotate cw.
7. We keep going until we hit a wall like in the scanning algorithm but with a fail counter mechanism that helps us avoid momentary key-point misses and lets us stay closer to the wall.

<u>**How we got to this solution:**</u>

At first we just moved in each axis separately, this fails because “right” and “right after rotating θ degrees” are vastly different things. We then fought of the general idea of “look at the target, go there”, this worked at first but we noticed that ORB-SLAM is good at seeing other points but not so much at estimating the current location, so we decided to keep the currAngle and a currPos vector and operate on them, this insures we always have accurate position and orientation. This worked sometimes, but sometimes seemed to rotate in the wrong direction, so we added the cross-product check. This now worked wonderfully, except for when we wanted to go to the exit where we encountered early stopping, we implemented the “go to wall” method and that’s where we are now.

Visualization:

To help visualize what the exit finder algorithm thinks we added some visualization with python, it shows the points found by ORB-SLAM (red), the points on the lines near the exit (blue) and the exit point (green). For example:


p_file='points/36.txt'
Y-axis
X-axis