CS 489/689 Autonomous Racing: Lab 4 Scan Matching

Section 1001, Fall 2023

1 Lab Description

Students will learn to use (1) C++ ROS2 to complete (2) localization algorithms using (3) Quadratic Programming.

2 Learning Outcomes

Outcome 1 Learn to use C++ ROS2

Outcome 2 Learn about the Localization Algorithm, Scan Matching

Outcome 3 Learn to use Quadratic Programming

3 Rubric

Display of Naive Correspondence	20
Display of Fast Correspondence	20
Improvement Between Naive and Fast	60
Total	100

4 Submission Instructions

A completed project is demonstrated to the teaching assistant on the simulator. Implementations will be demonstrated in simulator only. The vehicle will be set along the track on the *right side* of the Levine Hall map (the default map) with the vehcile facing North. See Section 9 for an example visualization. Submissions of the `scanmatch_node` package to canvas will be accepted after a successful demonstration.

4.1 Demonstration Instructions

A satisfactory demonstration includes the following:

- Demonstration of a successful scan matching algorithm using the Marker plugin for the simultaneous visualization of:
  - The Naive Correspondence in one color.
  - The Fast Correspondence in another color.

The vehicle will drive one lap around the Levine Hall map, and the implemented correspondence algorithms will be compared along the way. The Fast Correspondence implementation is expected to quickly find the corresponding data sets, thus allowing faster computation of transform errors.

5 Scan Matching Skeleton Code

The skeleton package provided, `lab5_pkg`, includes three program files and their respective header files. Changes to `transform.cpp` and `correspond.cpp` are required. If changes to other files are required, they must be noted and presented during evaluation.

5.1 Lab 5 Package Contents

Within the package's `src` folder are the following source files:

1. **scanmatch_node.cpp** Contains the main function and the driver for the scan matching algorithm. When compiled and linked, it produces the `scanmatch_node`.
2. **correspond.cpp and correspond.h** The API description and implementation of correspondence search. The declarations are found within `correspond.h`. Definitions are found with within `correspond.cpp`.

   `getNaiveCorrespondence()`, the naive function implementation, has already been provided.

   `getCorrespondence()` must be implemented with the faster search. `getCorrespondence()` will require the following:

   (a) **Input**:
      1. old_points: vector of struct points containing the old points of the scan
      2. trans_points: vector of struct points containing the new points transformed in the previous scans frame
      3. jump_table: jump table created from the helper function `computeJump`
      4. c: vector of struct correspondences passed by reference

   (b) **Output**:
      1. c: vector of struct correspondences as a reference.

   (c) **Helper Functions**:
      1. computeJump: computes the jump table based on the previous table and current scan points
3. **transfrom.cpp and transform.h** The API description and implementation of transformation and transformation update. The declarations are found within `transform.h`. Definitions are found with within `transform.cpp`.

   `updateTransform()` will require the following:

   (a) **Input**:
      1. corresponds: vector of struct correspondence containing corresponding points
      2. curr_trans: contains the transform found by the previous optimization step as a reference. Must be updated with the new transform.

   (b) **Output**:
      1. curr_trans: The updated transform

   (c) **Helper Functions**:
      1. transformPoints(): applies a transform to the set of points. Input the required transform and the points to be transformed.
      2. get_cubic_root: finds the cubic roots given 4 input coefficients of a cubic polynomial.
      3. greatest_real_root: finds the greatest real root of a quartic polynomial using the 5 coefficients of the quartic polynomial as the input.
4. **visualization.cpp and visualization.h** The API description and implementation of visualization. A complete implementation will likely require updates to the visualization to support the display of two sets of markers.

5.2 Data Structures

Several data structures are provided by the skeleton. Feel free to use or avoid these data structures in favor of your own. The functions, however, must be implemented as defined.

Struct Point The struct the radial distance and angular distance of a point from the car frame. In this struct there are few functions implemented which can be used to derive other information from the points:

- `distToPoint(pointP)`: find the distance to another point
- `distToPoint2(pointP)`: find the square of the distance to another point
- `radialGap(pointP)`: find the radial gap to another point
- `getx()`: get the x coordinate of the point
- `gety()`: get the y coordinate of the point
- `wrapTheta()`: wrap theta around 2 pi during rotation
- `rotate(phi)`: rotate the point by angle phi
- `translate(x,y)`: translate a point by distance x and y
- `getVector()`: get the vector (in x and y)

Struct Correspondence This struct stores the correspondence values which you find through the fast search algorithm. It contains:

- P: the transformed points
- P_0: original points
- P_{j1}: first best point
- P_{j2}: second best point

  Within the struct also exist the following functions:
  - `getNormalNorm()`: get normal of the correspondence
  - `getPiGeo()`: get correspondence point as a geometry message
  - `getPiVec()`: get correspondence point as a vector message

Struct Transform The struct stores the transform which you calculate through optimization at every step. It contains the x translation, y translation and the theta rotation.

- `apply(pointP)`: apply the transform to a provided point
- `getMatrix()`: get the transformation matrix from the found transform

6 The PLICP Algorithm

This lab implements the approach from Andrea Censi's PLICP paper, specifically within `transform.cpp` and `correspond.cpp`. An understanding of the work and the method are required understand the approach and to succeed in the task at hand.

The PLICP (Point-to-Line Itterative Closest Point) algorithm is introduced as an ICP algorithm that utilizes a point-to-line metric. The resulting algorithm converges quadratically, where as vanilla ICP has a linear convergence.

To aid in understand the algorithm a summary of background information is repeated herein.

6.1 ICP

The vanilla ICP algorithm is a surface matching algorithm. It iteratively minimizes the distance between a set of roto-translated points to their projection on a surface. Thus, it is said vanilla ICP uses a point-to-point metric. This can lead to large initial errors, as the algorithm may initially be matching one point to another seemingly innocuous, yet unrelated point. Large errors mean a slow convergence, an expensive correspondence search and more often than not, points become designated as outliers that simply do not match within the current set of surface scan points.

6.2 The Point-to-Line Metric

Measures the distance between a point to its nearest line segement. The designated point may also be projected onto an extension of the line segment, overall allowing for a simpler derivation of distance to the surface.

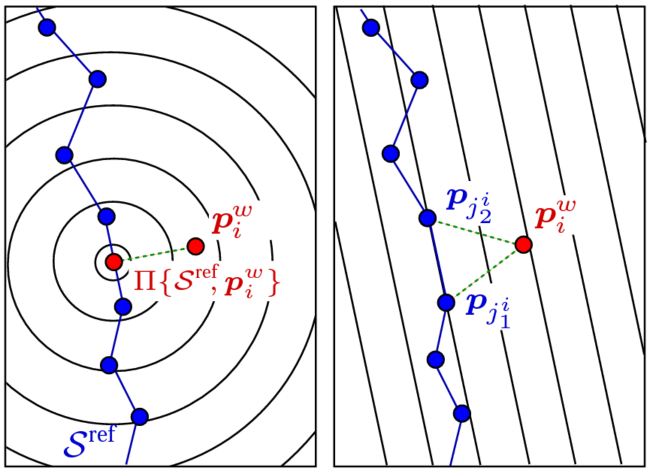


Figure 1: Image Visualization of a Point-to-Point Metric (Left) and Point-to-Line (Right)

6.3 PLICP

The algorithm takes 3 inputs:

- A reference scan.
- A current scan.
- An initial guess of the roto-transform, calculated by a previous iteration.

The algorithm outputs a transformation, which is the converged solution representing the roto-transform between the current and reference scan. This calculated transformation becomes the next iteration's initial guess, and will then be used, along with a formulated point-to-line error, to transform coordinates of the current scan onto the frame of the previous scan. These steps will be repeated until convergence with the surface point set.

There are two components of this approach, a transformation (implemented within `transform.cpp`), and correspondence (implemented within `correspond.cpp`). A complete implementation of the transform and two correspondence methods are required to complete the lab. The naive correspondence method has been included in the provided skeleton, the fast method has not.

6.4 Algorithm Implementation

The F1Tenth's Scan Matching Lab Video is a suggested guide for approaching a solution. In order to implement the scan matching algorithm, the following closed-form solution (1) must be derived from an optimization equation (3).

This optimization sums up all the distances from the transform point to the line segments, and tries to minimize the objetive function in order to acquire next iterations best transform guess.

$$\underset{x}{min}\, \mathbf{x^T M x + g^T x} \tag{1}$$

$$\underbrace{x\,(\sum_i M_i{}^T C_i M_i)\,x}_{\text{M}} + \underbrace{(\sum_i -2\pi_i{}^T C_i M_i)\,x}_{\text{g}} \qquad \text{s.t. } x^T W x = 1 \tag{2}$$

1. Equation (1) above matches the implementation layout within the skeleton code. It is derived from the following non-linear minimization problem that encompasses the point-to-line metric:

$$\underset{t,\theta}{min} \sum_i ||(R(\theta)p_i + t) - \pi_i||^2_{C_i} \tag{3}$$

where $p_i, \pi_i, t \in \mathbb{R}^2$; $R(\theta)$ is the 2x2 rotation matrix, and the norm $||a||^2_C$ is defined as $a^T C a$. π_i is the correspondence vector for the transform point.

To reach Equation (2), we will verify and apply the following steps:

(a) The optimization variable, which contains the position and orientation transforms to be optimized, is:

$$x = [x1, x2, x3, x4] \triangleq [t_x, t_y, cos\theta, sin\theta]$$

where the coordinates correspond to translation in x, y (transform) in 4D space. From this we can obtain a 3D solution if we apply a constraint.

(b) W is the weight matrix which is needed for applying the constraint $x_3^2 + x_4^2 = 1$, which is the equivalent expression to angle θ using cos and sin.

$$W = \begin{bmatrix} 0_{2x2} & 0_{2x2} \\ 0_{2x2} & I_{2x2} \end{bmatrix}$$

The constraint can then be written as $x^T W x = 1$, as was found in Equation (1).

(c) Matrix M_i:

$$M_i = \begin{bmatrix} 1 & 0 & p_{i0} & -p_{i1} \\ 0 & 1 & p_{i1} & p_{i0} \end{bmatrix}$$

(d) Points p_i in the scans are used to help define matrix M_i:

$$p_i = (p_{i0}, p_{i1})$$

(e) Matrix C_i:

$$C_i = w_i n_i n_i^T$$

where $n_i \in \mathbb{R}^2$ is the vector normal to the line.

2. Now, we solve the optimization problem as follows:

(a) Using Lagrange multipliers so the solution to this problem can be found and the solution takes the following form:

$$x = -(2M + 2\lambda W^{-T})g \tag{4}$$

where W is defined in the previous step and M and g are known to us. See Equation 2. We just need to find λ to solve this equation.

(b) We can find λ using:

i. Write the expression of the previous equation in the form:

$$2M + 2\lambda W = \begin{bmatrix} A & B \\ B^T & D + 2\lambda I \end{bmatrix} \tag{5}$$

you will need to derive expressions A, B, and D based off this equation.

ii. Then, calculate S in the following expression:

$$Q = (D - B^T A^- 1B + 2\lambda I) \triangleq (S + 2\lambda I) \tag{6}$$

iii. Calculate S^A:

$$S^A = det(S) \times S^{-1} \tag{7}$$

iv. Calculate $p(\lambda)$:

$$p(\lambda) = det(S + 2\lambda I) \tag{8}$$

v. Use the parameters found about to solve the following quartic equation to find λ. p is a function of λ. Use helper functions which have been provided in the code, you only need to write out the equation and find the coefficients of quartic equation that follows:

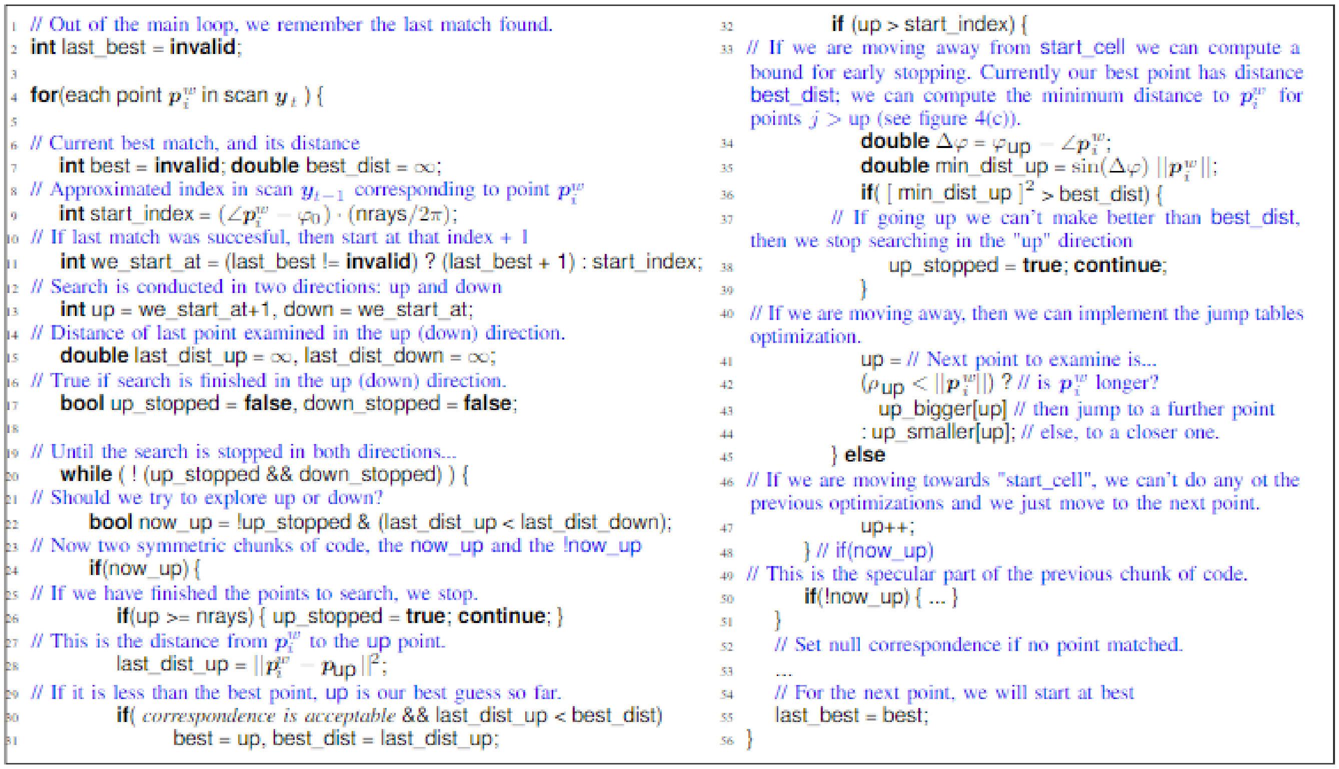
$$\begin{aligned} [p(\lambda)^2] = 4\lambda^2 g^T &\begin{bmatrix} A^{-1}BB^TA^{-T} & -A^{-1}B \\ -A^{-1}B & I \end{bmatrix} g \\ +4\lambda g^T &\begin{bmatrix} A^{-1}BS^AB^TA^{-T} & -A^{-1}BS^A \\ -A^{-1}BS^A & S^A \end{bmatrix} g \\ +g^T &\begin{bmatrix} A^{-1}BS^{A^T}S^AB^TA^{-T} & -A^{-1}BS^{A^T}S^A \\ -A^{-1}BS^{A^T}S^A & S^{A^T}S^A \end{bmatrix} g \end{aligned} \tag{9}$$

where I is an identity matrix.

vi. Now that you have λ, plug λ back into Equation 5 to solve for x.

vii. Repeat until convergence. This can be a fixed number of iterations or until the difference in the metric between iterations is below some threshold.

7 Fast Correspondence Search



```
// Out of the main loop, we remember the last match found.
int last_best = invalid;

for(each point $p_i^w$ in scan $y_t$ ) {

// Current best match, and its distance
    int best = invalid; double best_dist = $\infty$;
// Approximated index in scan $y_{t-1}$ corresponding to point $p_i^w$
    int start_index = $(\angle p_i^w - \varphi_0) \cdot (\text{nrays}/2\pi)$;
// If last match was succesful, then start at that index + 1
    int we_start_at = (last_best != invalid) ? (last_best + 1) : start_index;
// Search is conducted in two directions: up and down
    int up = we_start_at+1, down = we_start_at;
// Distance of last point examined in the up (down) direction.
    double last_dist_up = $\infty$, last_dist_down = $\infty$;
// True if search is finished in the up (down) direction.
    bool up_stopped = false, down_stopped = false;

// Until the search is stopped in both directions...
    while ( ! (up_stopped && down_stopped) ) {
// Should we try to explore up or down?
        bool now_up = !up_stopped & (last_dist_up < last_dist_down);
// Now two symmetric chunks of code, the now_up and the !now_up
        if(now_up) {
// If we have finished the points to search, we stop.
            if(up >= nrays) { up_stopped = true; continue; }
// This is the distance from $p_i^w$ to the up point.
            last_dist_up = $||p_i^w - p_{up}||^2$;
// If it is less than the best point, up is our best guess so far.
            if( correspondence is acceptable && last_dist_up < best_dist)
                best = up, best_dist = last_dist_up;
            if (up > start_index) {
// If we are moving away from start_cell we can compute a
   bound for early stopping. Currently our best point has distance
   best_dist; we can compute the minimum distance to $p_i^w$ for
   points $j$ > up (see figure 4(c)).
                double $\Delta\varphi = \varphi_{up} - \angle p_i^w$;
                double min_dist_up = $\sin(\Delta\varphi)\ ||p_i^w||$;
                if( [ min_dist_up ]$^2$ > best_dist) {
            // If going up we can't make better than best_dist,
   then we stop searching in the "up" direction
                    up_stopped = true; continue;
                }
// If we are moving away, then we can implement the jump tables
   optimization.
                up = // Next point to examine is...
                ($\rho_{up} < ||p_i^w||$) ? // is $p_i^w$ longer?
                  up_bigger[up] // then jump to a further point
                : up_smaller[up]; // else, to a closer one.
            } else
// If we are moving towards "start_cell", we can't do any ot the
   previous optimizations and we just move to the next point.
                up++;
        } // if(now_up)
// This is the specular part of the previous chunk of code.
        if(!now_up) { ... }
    }
    // Set null correspondence if no point matched.
    ...
    // For the next point, we will start at best
    last_best = best;
}
```

Figure 2: The Pseudo-C code for a smart algorithm provided in Andrea Censi's PLICP paper

After finishing coding the transforms algorithm, we will need to update our `correspond.cpp` from the provided naive implement to a fast correspondence search. The psuedo code provided in Censi's PLICP paper, shown in Figure 2 above, provides a good starting point for a fast implementation.

Here are a few pointers to take care of while coding the algorithm.

- Jump table is computed for one scan set, not each scan point. The scan table implementation is already present in the skeleton.
- Start the correspondence search from the previous best point correspondence found for each point.
- It is safe to make the assumption that the second best point is adjacent to the best point found. Decreasing or increasing the index (while taking care of the edge cases) should work. Studetns are free to find better implementations.
- Make use of early stopping criterion cleverly. It helps in making the search faster.
- For finding the distances between the points, the functions implemented in the Point struct will be useful. Be careful, the point to line metric is not the metric for finding the closest point. It is only the metric for optimization. For correspondence, only search for the closest point by euclidean distance.

7.1 Building and Utilizing the Package

Between all changes in these C++ files, it is required to rebuild the used packages. To do this, run the following commands:

```
root@5b24be3a66e0:/sim_ws# colcon build --packages-select lab5_pkg
root@5b24be3a66e0:/sim_ws# ros2 run lab5_pkg scanmatch_node
```

If the package is not being included, the workspace must be re-sourced. Confirm this by running the following:

```
# The following will build any package that is out of date
root@5b24be3a66e0:/sim_ws# colcon build --packages-select lab5_pkg
Starting >>> lab5_pkg
Finished <<< lab5_pkg [8.80s]

Summary: 1 package finished [9.41s]
root@5b24be3a66e0:/sim_ws#
```

Re-sourcing the Workspace

Then verify the package has been properly created and included. The package will not included until the workspace has been re-sourced.

```
root@29629eeaf74a:/sim_ws# ros2 pkg list | grep lab5_pkg
root@29629eeaf74a:/sim_ws#
# Not listed
root@29629eeaf74a:/sim_ws# source install/local_setup.bash
root@29629eeaf74a:/sim_ws# ros2 pkg list | grep lab5_pkg
lab5_pkg
# Now listed
```

Verifying lab5_pkg

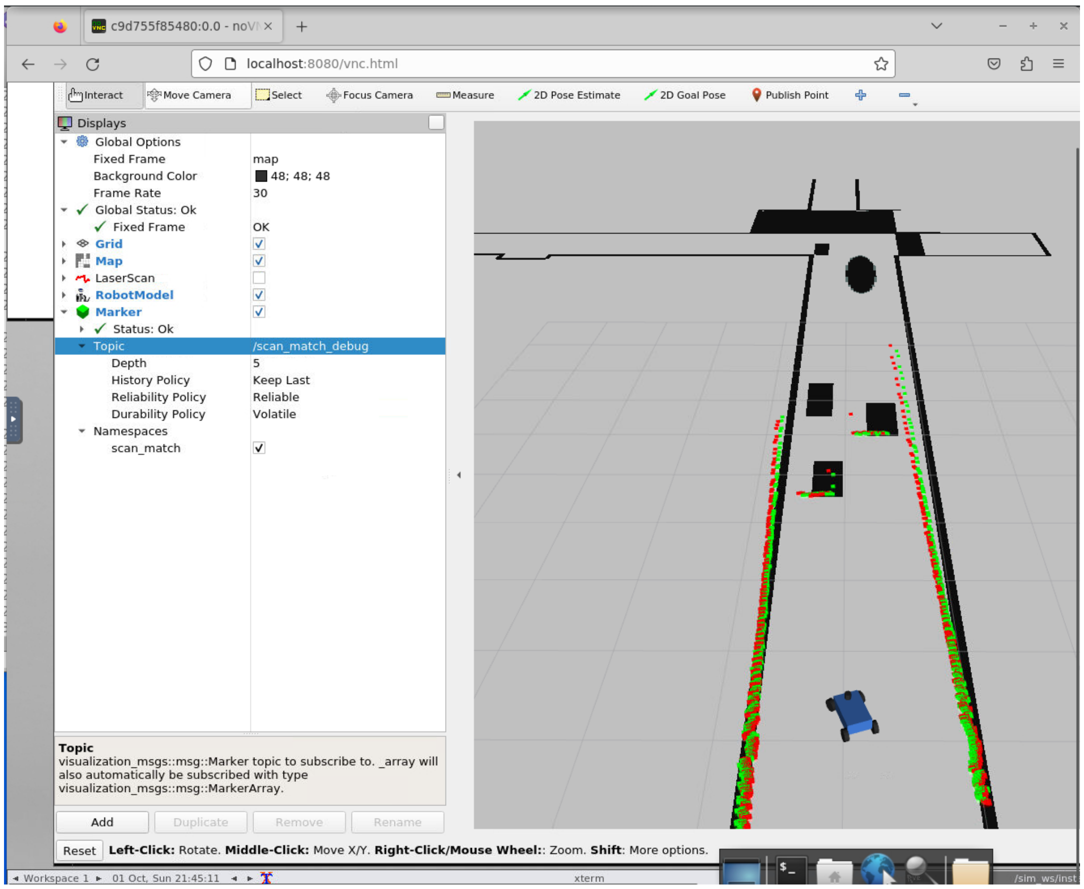
8 Eigen C++ Library

Eigen is a C++ template library for linear algebra. Included below are useful matrix functions included in the library:

- **Eigen::Transpose**: obtains the transpose of the matrix
- **Eigen::Block**: performs block operations- can be used to obtain a submatrix from an original matrix
- **Eigen::Identity**: obtains an identity matrix
- **Eigen::Inverse**: obtains the inverse
- **Eigen::Determinant**: obtains the determinant
- **Eigen::Trace**: the sum of the diagonal coefficients and can also be computed as efficiently using .diagonal().sum()
- **Eigen::Sum**: reduces a matrix to a single valued sum

9 Visualization on Simulator

To help visualize `transform.cpp` in the gym simulator, the `Marker` plugin is recommended. The Topic to be subscribed to is `/scan_match_debug`. Publishing to this topic will display colored markers on the global frame.



```
Displays
  Global Options
    Fixed Frame             map
    Background Color        48; 48; 48
    Frame Rate              30
  ✓ Global Status: Ok
    ✓ Fixed Frame           OK
  Grid                      ✓
  Map                       ✓
  LaserScan
  RobotModel                ✓
  Marker                    ✓
    ✓ Status: Ok
    Topic                   /scan_match_debug
      Depth                 5
      History Policy        Keep Last
      Reliability Policy    Reliable
      Durability Policy     Volatile
    Namespaces
      scan_match            ✓
```

Topic
visualization_msgs::msg::Marker topic to subscribe to. _array will also automatically be subscribed with type visualization_msgs::msg::MarkerArray.

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift**: More options.

Figure 3: The Transform Frames Projected by the Marker plugin

The density of markers is directly related to the number of iterations (`number_iter` within `transform.cpp`). Increase the number of iterations to increase the visibility of the matching. However, increasing the number of iterations, will slow convergence.

Lab 4 Resources

F1TENTH Lab 5: Scan Matching The F1TENTH group's description of Lab 5: Scan Matching. Instruction URL: https://github.com/f1tenth/f1tenth_labs_openrepo/tree/main/f1tenth_lab5

An ICP variant using point-to-line metric- Andrea Censi Andrea Censi's paper published in 2008 IEEE International Conference on Robotics and Automation on PLICP. Instruction URL: https://censi.science/pub/research/2008-icra-plicp.pdf

Eigen C++ Template Library Documentation Eigen documentation with helpful information for dealing with matricies: https://eigen.tuxfamily.org/index.php?title=Main_Page

F1Tenth's Scan Matching Lab Part 1 A breakdown of important concepts regarding Scan Matching's transformation implementation: https://www.youtube.com/watch?v=WmoOA1XXh_o