Lab 1

Erik Hammar

October 2022

1 Homogeneous Transform

In this lab the principal task was to create a class for homogeneous transform. The underlying logic is derived from the course book, where the homogeneous transformation matrix (1) is comprised of a three by three rotation matrix and a three by one translation vector.

$$T = \begin{bmatrix} R & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \tag{1}$$

In the class, called `HomTrans`, there are four attributes; the entire transform matrix, the rotation matrix, translation vector and the inverse of the transform matrix. For each transform feature a member function exists to perform the calculations.

1.1 Inverse

An advantage of the homogeneous transform matrix (1) is the inverse matrix and the lack of complex calculations. The much simpler and computationally efficient formula can be used (2). Since all vectors and matrices are implemented with the Eigen library, vector and matrix manipulation is convenient.

$$T^{-1} = \begin{bmatrix} R^T & -R^T\mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \tag{2}$$

```
class HomTrans {
    private:
    // data
    public:
        Matrix4d T;                     // Four dimensional homogeneous transformation matrix
        Matrix3d R;                     // Three dimensional rotation matrix
        Vector3d t;                     // Three dimensional translation vector
        Matrix4d invT;                  // Inverse matrix of T


        void createTMatrix(Vector3d anglesZYX, Vector3d translation);   // AnglesZYX should be
        void inverseT();
        Vector4d transformPoint(Vector4d p);
        HomTrans transformFrame(HomTrans frame);
        Matrix4d identityTransform();
        void eulerTransform(Vector3d angles, Vector3d translation);
        Vector3d getRPY();
        Vector4d toAngleAxis();
        void fromAngleAxis(Vector3d axis, double angle);
        Vector4d toQuaternion();
        void fromQuaternion(Vector4d q);
        Matrix3d getRotationX(double gamma);
        Matrix3d getRotationY(double beta);
        Matrix3d getRotationZ(double alpha);

        HomTrans();
        ~HomTrans();

};
```

Figure 1: Member function and attribute declarations of the homogeneous transform class `HomTrans`

```
void HomTrans::inverseT() {
    invT <<
    R.transpose(), -R.transpose()*t,
    0, 0, 0, 1;
};
```

Figure 2: Inverse implementation

1.2 Euler and RPY

Implementing the arithmetic behind each feature was straight forward from the course book, however the structure of the class and the data structures used for the function returns were difficult to get right. Since the class was used more rigorously later on some implementations were found to be ineffective or incompatible with other functions/tasks. For example, the member function `transformFrame` could have returned just a four dimensional matrix, or even be a void function, instead of another transform object.

This was the case for Euler transform and Euler angles since, when constructing the class, it was unclear how these functions were going to be used.

```
HomTrans HomTrans::eulerTransform(Vector3d angles) {
    HomTrans eT;
    Matrix3d Rz, Ry, Rx, eR;
    Rz = getRotationZ(angles(0));
    Ry = getRotationZ(angles(1));
    Rx = getRotationZ(angles(2));

    eR = Rz * Ry * Rx;

    Vector3d et(0, 0, 0);

    eT.T <<
    eR, et,
    0, 0, 0, 1;

    return eT;
};
```

(a) Euler Transform

```
Vector3d HomTrans::getRPY() {
    Vector3d rpy;

    rpy <<
    atan2(R(1,0), R(0,0)),
    atan2(-R(2,0), sqrt(pow(R(2,1),2) + pow(R(2,2),2))),
    atan2(R(2,1), R(2,2));

    return rpy;
};
```

(b) Roll-Pitch-Yaw angles

Figure 3: Code implementation

1.3 Angle-Axis and Quaternion

These calculation, in principle, were also straight forward. However, when using the class and converting the rotation of frames between different representations as well as between different libraries' data types, the order in which I place the elements in each vector was the discovered to be not so straight forward. In a lot of places the quaternion vector, for example, is returned with the order of the elements flipped.

```cpp
Vector4d HomTrans::toAngleAxis() {
    Vector4d angleAxis;
    double theta = acos((R(0,0) + R(1,1) + R(2,2) -1)/2);

    angleAxis <<
    theta,
    (R(2,1) - R(1,2))/(2*sin(theta)),
    (R(0,2) - R(2,0))/(2*sin(theta)),
    (R(1,0) - R(0,1))/(2*sin(theta));

    return angleAxis;
}
```

(a) From rotation to angle axis representation

```cpp
Vector4d HomTrans::toQuaternion() {
    Vector4d angleAxis;
    Vector4d q;

    angleAxis = this->toAngleAxis();

    q <<
    cos(angleAxis(0) / 2),
    sin(angleAxis(0) / 2) * angleAxis(1),
    sin(angleAxis(0) / 2) * angleAxis(2),
    sin(angleAxis(0) / 2) * angleAxis(3);

    return q;
}
```

(b) From rotation to quaternion representation

Figure 4: Code implementation

Figure 5: Unit test class `TestHomTrans`

2 Unit tests

To perform unit tests on the `HomTrans` class methods another calss was created called `TestHomTrans` 5. Each unit test compares the result of my implementation to the result of the corresponding result using Eigen transform class. The unit tests are implementedlike in figure 6.

3 Shortest Transform

For this task, I had some problems with values of different matrices and vectors. Before discussing the problems I will explain how the code works.

First, I create two `TransformStamped` objects for the two frames that I want to find the transform between and populate them with translation and rotation. Then I create four `HomTrans` objects, and populate two using the

Figure 6: Unit test for inverse homogeneous tranform

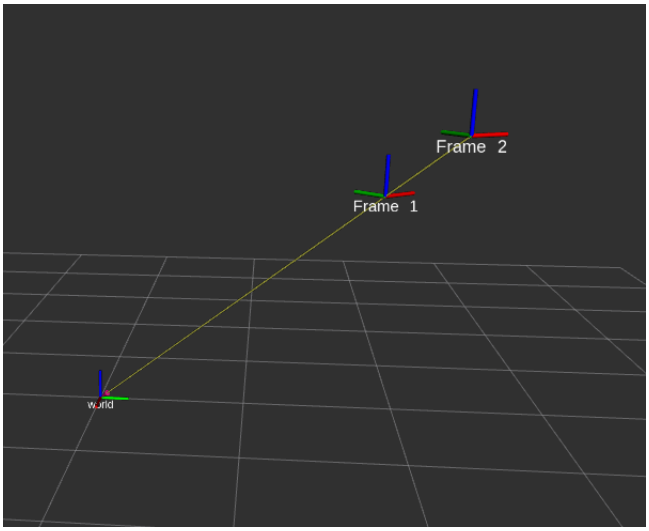

Figure 7: Frame 1 on its way to frame 2 in relation to frame 0

frame 1 and frame 2 transform stamped values. These now represent the homogeneous transform matrices from frame 0 to frame 1 as well as from frame 0 to frame 2. Now, to get the transform between frame 1 and 2 the inverse of frame transform 1 was used as in 3. (^iT_j denotes the homogeneous transform from frame i to frame j).

$$
\begin{aligned}
{}^1T_0 &= ({}^0T_1)^{-1} \\
{}^1T_2 &= {}^1\ T_0 * {}^0\ T_2
\end{aligned}
\tag{3}
$$

With the homogeneous transform matrix to get from frame 1 to frame 2 obtained the translation part and rotation part of the matrix were extracted and divided into equally sized parts. The parts were used to incrementally add to and update the frame 1 transform object and broadcast the new transform. The resulting effect is that frame 1 "walks" to the exact position and orientation as frame 2 (figure 7).

One part of the task was to randomly generate the two frames' values. The main issue was that when randomly generating the rotational part of the frames (which is done by generating RPY-values and creating a quaternion) the unit quaternion does not satisfy $\eta^2 + \epsilon_x^2 + \epsilon_y^2 + \epsilon_z^2 = 1$ and therefore not correct. This caused the conversion to homogeneous transform matrix to produce NaN-values. After trying to debug the problem for a very long time I

decided to initiate two fixed frames instead where I knew the quaternion was correct, just to be able to verify that the rest of the code was implemented correctly.

4 Running the code

The ROS package includes three source files (*hTransform.cpp, publisher˙node.cpp, test.cpp*) and three header files with the same names. To run the shortest transform program from the terminal, from the robotics workspace type: `rosrun lab1 publisher_node`.

5 Appendix

All files in *Robotics/src/lab1*:
hTransform.cpp
publisher˙node.cpp
test.cpp
hTransform.hpp
publisher$_n$*ode.hpp*
test.hpp