Lab 3

Erik Hammar

October 2022

1 Inverse Kinematic Solver

To complete the task an inverse kinematics solver is implemented by creating a class called `KDLSubscriber`. As seen in figure 1 there are a couple of KDL library objects. These are used to calculate different things later on. In the class constructor the class attributes are initiated. This part took a very long time to figure out since there wasn't a lot of documentation on KDL.

The joint array qc is set to a random pose. Then, in the function `solveIK()` two Eigen transform objects are set; one to the values based on the kinematic chain created by KDL tree and joint array qc, and the other to the values of the goal pose whit a kinematic chain based on the joint array $qg = (0, 0, 0)$. The next step is similar to the shortest transform task in lab 1, where the matrix of the current state is inverted and multiplied by the goal matrix to get the shortest transform from the current state to the goal state. From this transform a velocity vector is calculated by the function `getVelocity()`, this vector is used in the next step.

The next part is the algorithm which causes the current joint array qc to converge to the goal joint array qc. We use an equation similar to gradient decent (1).

$$q_c = q_c + \lambda J^{\#} v \tag{1}$$

There were some issues that arose during the implementation of the algorithm as well as calculating the pseudo inverse (2).

$$J^{\#} = J^T (J J^T)^{-1} \tag{2}$$

```
class KDLSubscriber {
    private:
    // data
    public:
    KDL::Tree my_tree;
    KDL::TreeFkSolverPos_recursive *fkSolver;
    ros::NodeHandle node;
    ros::Subscriber subscriber;
    ros::Publisher publisher;
    KDL::TreeJntToJacSolver *jacobiSolver;
    KDL::JntArray qc;
    std::vector<double> jntsPanda;
    std::vector<double> jnts3dof;
    sensor_msgs::JointState jointStateMessage;

    float randomGuess();
    void randomPose();
    Eigen::VectorXd getVelocity(KDL::JntArray qc, KDL::JntArray qg);
    void callBack(const sensor_msgs::JointState &jointState);
    void solveIK();
    void calculateEndFrame(const sensor_msgs::JointState &jointState);
    // Eigen::VectorXd calculateEndFrame(KDL::JntArray q);
    Eigen::Transform<double, 3, Eigen::Affine> calculateEndFrame(KDL::JntArray q);
    Eigen::MatrixXd calcualteJacobian(const sensor_msgs::JointState &jointState);
    Eigen::MatrixXd calcualteJacobian(KDL::JntArray q);

    KDLSubscriber();
    ~KDLSubscriber();
};
```

Figure 1: KDLSubscriber class

Since the jacobian J is (in most cases) an irregular matrix and it's impossible to invert a non-square matrix, we use the pseudo inverse. When testing the code I noticed NaN and Inf values in the six by six matrix resulting in inverting JJ^T. The problem was a non-invertible sparse matrix which results in division by zero. The solution to this was to add a small value to the diagonal of JJ^T.

Figure 2 shows how the velocity vector, pseudo inverse of J and qc are used in the algorithm. λ is set to 10^{-5}.

```
deltaVelocity = this->getVelocity(this->qc, qGoal);

while (deltaVelocity.norm() > 0.001) {

    cout << deltaVelocity.block<3,1>(0,0).norm() << endl;

    pseudoInverseJ = this->calcualteJacobian(this->qc);

    JJT = pseudoInverseJ * pseudoInverseJ.transpose();
    JJT.diagonal().array() += 0.01;
    pseudoInverseJ = pseudoInverseJ.transpose()*(JJT.inverse());

    deltaVelocity *= pow(10, 1);
    for (int i = 0; i < nrJoints; i++) {
        qCurrent(i) = this->qc(i);
    }

    qCurrent = qCurrent + LAMBDA * pseudoInverseJ * deltaVelocity;

    for (int i = 0; i < nrJoints; i++) {
        this->qc(i) = qCurrent(i);
        this->jointStateMessage.position[i] = this->qc(i);
    }

    this->jointStateMessage.header.stamp = ros::Time::now();
    this->publisher.publish(jointStateMessage);
    deltaVelocity = this->getVelocity(this->qc, qGoal);
}
cout << "Finished!" << endl;
```

Figure 2: Code implementation of the algorithm

2 Three DOF planar robot

As visible in figures 3 and 4 the randomly generated poses converge to the joint state $(0, 0, 0)$. The second image in figure 3 is the terminal output of the vector norm between current state and goal state and "Finished!" when the norm is smaller than a chosen tolerance.

3 Franka Panda robot

The code in the previous parts were implemented in as dynamic way as possible, meaning the vectors and matrices involving the robot configuration are allocated using the KDL tree function `getNrOfJoints()` which returns

Figure 3: Vizualisation three DOF arm converging

Figure 4: Converged

the number of joints in the tree. To access the robot description from the server, the name of the robot as well as which link to calculate are needed. These were also made dynamic by defining a string `ROBOT_DESC` to be either "three˙dof˙planar˙eef" or "panda˙link8". Changing the define in the header file makes the program read the other description and load those parameters.

Unfortunately, the algorithm gets the manipulator to move but not converge. I suspect it has something to do with the randomly generated starting pose and the goal pose, and whether the robot can reach those points.

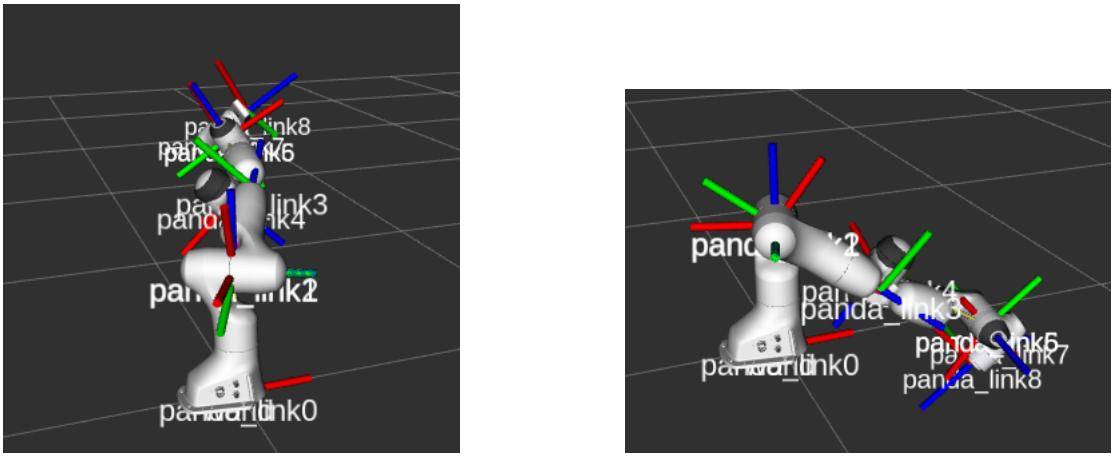

Figure 5: Seven degrees of freedom robot manipulator

4 Run the code

Choose which robot to run in *KDLSubscriber.hpp*. Run command in terminal: `roslaunch three_dof_planar 3dof_planar_fakesim.launch` for 3 DOF robot or `roslaunch three_dof_planar 3dof_planar_fakesim_panda.launch` for Franka panda robot.

5 Appendix

All files in *Robotics/src/lab3* and *Robotics/src/robotteknik*:
KDLSubscriber.cpp
lab3˙node.cpp
KDLSubscriber.hpp