Lab 2

Erik Hammar

October 2022

1 Joins and Chain

To create a kinematic chain where each joint position and pose can be calculated in relation to the previous one, I chose to implement two classes; one for Joint and one for Chain (figure 1. It is implemented so that each joint has an offset, which is the translation from the previous joint, and an angular value. With these values the state, the homogeneous transform using `HomTrans`, of the joint can be computed. With the joint offset embedded in each joint, the need for a link class was removed. A chain object consists of a vector of joints and each joint state attribute can be multiplied in sequence to get the transform matrix of the end-effector in relation to the world frame (frame 0).

```
class Joint {
    private:
    // data
    public:
    int jointID;
    // Joint* parentJoint;
    // Joint* childJoint;
    double angle;
    Vector3d offset;
    Vector3d rpy;
    HomTrans state;
    // void setParentJoint(Joint& parentJoint);
    // void setChildJoint(Joint& childJoint);
    void updateState(Vector3d rpy, Vector3d offset);

    Joint(Vector3d offset = Vector3d(0, 0, 0), Vector3d rpy = Vector3d(0, 0, 0), int ID = 0);
    ~Joint();

};
```

(a) Joint class

```
class Joint;

class Chain {
    private:
    // data
    public:
    std::vector<Joint*> jointList;
    void addJoint(Vector3d offset, Vector3d rpy);
    void printJoints();
    Matrix4d getTransform();
    void updateJointAngle(int jointID, double angle);


    Chain();
    ~Chain();

};
```

(b) Chain class

Figure 1: Header files of classes

The member function `getTransform()` of the chain class iterates through the joint list/joint vector and accesses each state and accumulatively multiplies them. To make sure the transformations are correct, they were printed out in the console during development (see figure 3).

```
Matrix4d Chain::getTransform() {
    HomTrans updatedFrame;
    Vector3d rpy, currentTranslation;
    Matrix4d totalTransform;
    totalTransform <<
        1, 0, 0, 0,
        0, 1, 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1;

    for (Joint* i : this->jointList) {
        rpy = i->state.getRPY();
        cout << "RPY: " << endl << rpy << endl;
        rpy = Vector3d(rpy(2), rpy(1), rpy(0) + i->angle);
        cout << "Updated RPY: " << endl << rpy << endl;

        currentTranslation = i->state.T.block<3,1>(0,3);
        cout << "Translation: " << endl << currentTranslation << endl;

        updatedFrame.eulerTransform(rpy, currentTranslation);
        cout << "Applied frame: " << endl << updatedFrame.T << endl << endl;

        totalTransform = totalTransform * updatedFrame.T;
        cout << "Total transform: " << i->jointID << endl << totalTransform << endl;

    }
    return totalTransform;
}
```

Figure 2: Calculates the end-effector transform in relation to the world frame

2 State controller

Since the urdf-parser was not implemented, the link lengths and joint configuration was read from the robot description xacro file and hard-coded into the implementation. This is done in the function `init()` in the StateController class. The update function of the StateController class then gets the current joint positions from the joint state handle and updates the joint angles in

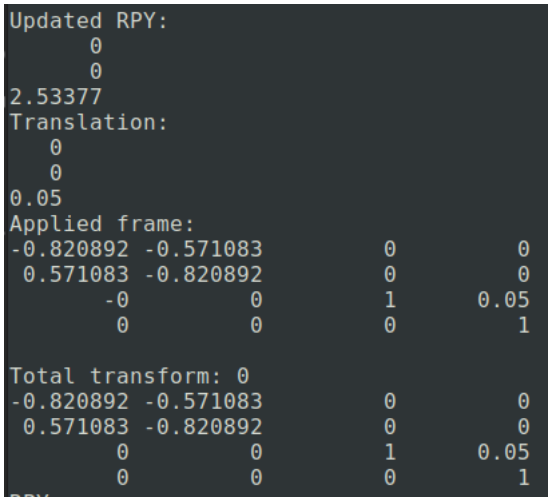

Figure 3: Values of the joint states when calculating end-effector transform

the kinematic chain (figure 4 and 5).

Using a transform broadcaster, the last joint of the kinematic chain corresponding to the end-effector, is broadcasted and displayed in RVIZ. As seen in (figure 6) the yellow line connects the world frame and the single broadcasted frame which follows the end-effector of the three degrees of freedonm planar robot.

3 Run the code

Since the classes are used in the state controller package, running one terminal with the command `roslaunch three_dof_planar 3dof_planar_gazebo.launch` and in another terminal running `rosrun rqt_joint_trajectory_controller rqt_joint_trajectory_controller`.

4 Appendix

All files in *Robotics/src/lab2* and *Robotics/src/robotteknik*:
chain.cpp
joint.cpp
state˙controller.cpp
chain.hpp

```cpp
bool StateController::init(hardware_interface::JointStateInterface* hw,
           ros::NodeHandle& root_nh, ros::NodeHandle& controller_nh) {
   ROS_INFO("State Controller: here read robot description from parameter server, initialize publishers, read parameters, load joi
   // std::string my_joint;
   // if(!root_nh.getParam("joint", my_joint)) {
   //    ROS_ERROR("Could not find joint name");
   //    return false;
   // }
   for (auto i : hw->getNames()) {
      this->j.push_back(hw->getHandle(i));
   }

   this->my_chain.addJoint(Vector3d(0, 0, 0.05), Vector3d(0, 0, (double)M_PI_4));
   this->my_chain.addJoint(Vector3d(0.5, 0, 0), Vector3d(0, 0, (double)-M_PI_4/2));
   this->my_chain.addJoint(Vector3d(0.3, 0, 0), Vector3d(0, 0, (double)-M_PI_4/2));
   this->my_chain.addJoint(Vector3d(0.2, 0, 0), Vector3d(0, 0, 0));

   std::cout << "End effector frame transform: " << std::endl << this->my_chain.getTransform() << std::endl << std::endl;

   this->updateBroadcast();

   return true;
}
```

Figure 4: Initializes the kinematic chain with $\mathbf{q}_0$

joint.hpp
state˙controller.hpp

```cpp
void StateController::update(const ros::Time& time, const ros::Duration& period) {
   ROS_INFO("State Controller: here read joint handles, compute forward kinematic model and publish tool tip pose to TF");
   double angle1 = this->j[0].getPosition();
   double angle2 = this->j[1].getPosition();
   double angle3 = this->j[2].getPosition();

   this->my_chain.updateJointAngle(0, angle1);
   this->my_chain.updateJointAngle(1, angle2);
   this->my_chain.updateJointAngle(2, angle3);

   std::cout << "update function" << std::endl;

   // std::cout << "End effector frame transform: " << std::endl << this->my_chain.getTransform() << std::endl << std::endl;
   this->updateBroadcast();

}
```

Figure 5: Updates joint angles in kinematic chain object when joint states of robot change

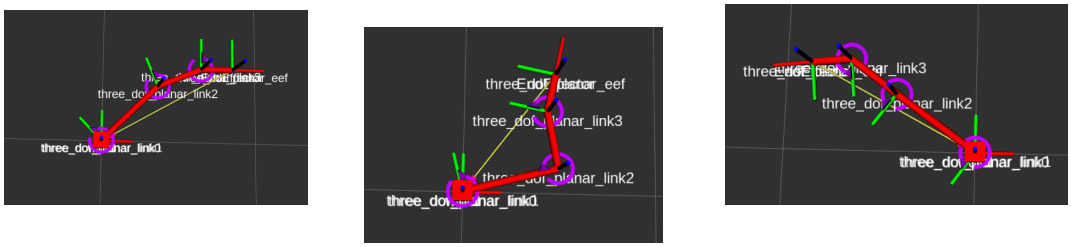

Figure 6: Vizualisation of robot arm with the EEF