Manipulator Modeling & Control

1 Content of this lab

The goal of this lab is to program in C++ the three fundamentals models for robot manipulators:

- Direct Geometric Model: what is the pose of the end-effector for a given joint configuration?
- Inverse Geometric Model: what joint values correspond to a given end-effector pose?
- Kinematic Model: what is the velocity screw of the end-effector when the joints move?

These models will be applied on three different robots, and used to perform basic point-to-point control.

As it is not a lab on C++, most of the code is already written and you only have to fill the required functions:

- `Robot::init_wMe()` to initialize the fixed transform between the wrist and end-effector
- `Robot::fMw(q)` for the direct geometric model wrist-to-fixed frame
- `Robot::inverseGeometry(M)` for...well, the inverse geometry
- `Robot::fJw` for the kinematic model wrist-to-fixed frame

1.1 Installing and loading the lab in Qt Creator

This project uses the ROS[1] framework which imposes some particular steps to configure the environment. An actual ROS course will be held in the second semester, for now just configure as follows:

1. Open a terminal and download the lab package: `rosdl ecn_manip`.
2. Run Qt Creator from the top screen icon
3. Load the `ros/src/ecn_manip/CMakeLists.txt` file through `File...open project`
4. QtCreator asks for a compilation folder: give `ros/build/ecn_manip`
5. The files should be displayed and ready to compile and run
6. Compilation is done by clicking the bottom-left hammer
7. Run your program with the green triangle. It can be stopped by clicking on the red square

[1] Robot Operating System, http://www.ros.org

1.2 Expected work

The **only** files to be modified are:

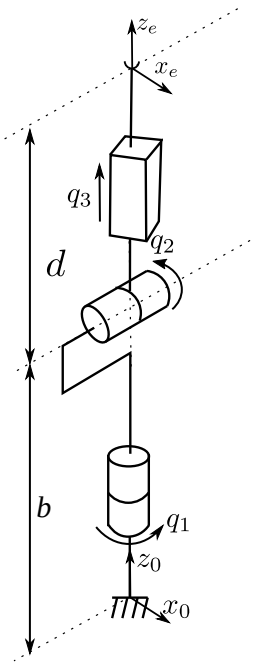
- `control.cpp`: main file where the control is done depending on the current robot mode
- `robot_turret.cpp`: model of the RRP turret robot
- `robot_kr16.cpp`: model of the industrial Kuka KR16 robot
- `robot_ur10.cpp`: model of the Universal Robot UR-10

We will use the ViSP[2] library to manipulate mathematical vectors and matrices, including frame transforms, rotations, etc. The main classes are detailed in Appendix A.
At the end of the lab, you should upload them through the portal on my website

2 The robots

Three robots are proposed with increasing complexity. You should start with the Turret, then the Kuka KR16 then the UR-10.
For each of these robots, the table of Modified Denavit-Hartenberg parameters should be defined. As we saw during the lectures, once you have the table then the Direct Geometric and Direct Kinematic Models can be almost automatically derived. A tool is provided in this package to generate C++ code for the Geometric and Kinematic models, see Appendix B for details on this tool.
The fixed frame and the end-effector frame are imposed by the schematics. All intermediary frames have to be placed according to MDH convention, with a final fixed transform between the wrist and end-effector frames.

2.1 Turret RRP robot



This is a very simple robot to begin the modeling, as shown in the figure.
The fixed frame $\mathcal{F}_0$ and end-effector frame $\mathcal{F}_e$ are imposed, as well as the joint direction.
The constant values for this model are: $b = 0.5$ m and $d = 0.1$ m.

All computation may be done first without using the Denavit-Hartenberg parameters as the robot is quite simple.

[2]Visual Servoing Platform, http://visp.inria.fr

2.2 Kuka KR16 anthropomorphic robot

This robot is a classical 6R architecture with a spherical wrist, as shown in the next figure:

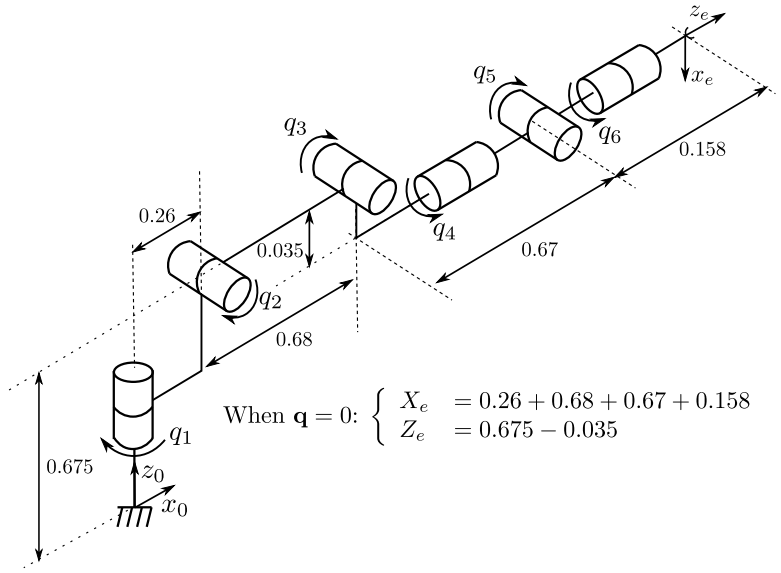


Figure 1: Model of the Kuka KR16 with imposed fixed and end-effector frames

The wrist frame is typically at the common point between joints 4, 5 and 6. The constant transform ${}^{w}\mathbf{M}_{e}$ between the wrist and the end-effector should thus include the 0.158 distance and potential rotations.

2.3 Running the simulation

Once everything is installed and compiled, the base simulation can be run from a terminal (only 1 line depending on the robot you work on):

```
roslaunch ecn_manip turret.launch
roslaunch ecn_manip kr16.launch
roslaunch ecn_manip ur10.launch
```

3 Building the models

The main code

In the file `control.cpp`, the switch-case block corresponds to the 6 exercises to be done for each robot. Note that if they work for the Turret robot, then they should work for all robots as long as the model is correct.
The current exercise is changed by using the buttons in the GUI.

Exercise 1: Checking the Direct Geometric Model

In this exercise, the robot follows the manually given joint positions. The task is to build and print the direct geometric model, depending on the current value of $\mathbf{q}$. This is to be done in two functions:

- `Robot::init_wMe()` where you have to write the constant matrix ${}^{w}\mathbf{M}_{e}$ from the end-effector to the wrist frame
- `Robot::fMw(q)` where you have to write and return the transform $\mathbf{M}$ between the fixed and wrist frame

Once you have computed the `M` matrix, the code calls `robot->checkPose(M);` in order to print both your matrix and the true one. No need to tell that they should be equal.
What can be told is that it is useless to continue until they are.

Exercise 2: Checking the Direct Kinematic Model (Jacobian)

If you select `Manual Operational Velocity` in the GUI then the sliders allow you to give a desired operational velocity (ie twist) for the end-effector.
The robot Jacobian has to be defined in the function `Robot::fJw(q)` that should return the Jacobian of the wrist in the fixed frame.

In the main code, the commanded velocity has then to be mapped to the joint velocity through the robot Jacobian: `vCommand = robot->fJe(q).pseudoInverse()*v;`
You can check that the end-effector is able to follow straight 3D lines in x, y or z direction. Similarly, it should be able to rotate around x, y or z. This means the Jacobian computation is correct.

Exercise 3: Checking the Inverse Geometric Model

If you select `Direct point-to-point` in the GUI then the robot will receive two operational setpoints, at a given sampling time that can be changed with the `Switching time` slider in the GUI. The current setpoint is in the matrix `Md` and its corresponding joint position is in the vector `qf`.
Nothing has to be done in the main code, but of course the `Robot::inverseGeometry(M)` function has to be filled in order to compute the actual inverse geometry solution.

Exercise 4: Interpolated point-to-point control

Exercise 5: Inverse Geometric Model and trajectory tracking

Exercise 6: Operational control through Jacobian

A Using ViSP

ViSP is a library for Visual Servoing (wait for the M2 Robotics to know more!) and also includes all necessary tools to play with mathematical vectors, matrices and 3D transforms. The full documentation is here: http://visp-doc.inria.fr/doxygen/visp-3.1.0/classes.html.
The main classes to be used are:

- `vpMatrix`: a classical matrix of given dimensions, can be multiplied with other matrices and vectors if the dimensions are ok. The pseudo-inverse of matrix `M` is available with `M.pseudoInverse()`
- `vpColVector`: a column vector of given dimension
- `vpHomogeneousMatrix`: the classical 4×4 3D transform matrix

Most of the model computation is about initializing a matrix or vector to correct dimensions (which is already done) and fill its elements with formulae.
To write 1 at element (i,j) of matrix M, just write `M[i][j] = 1;`
Similarly, to write 1 at element (i) of vector q, just write `q[i] = 1;`

B Code generation from Modified Denavit-Hartenberg parameters

The core of robot modeling is to build the MDH table. From this, the DGM and KM can be derived automatically. The Inverse Geometry still has to be done by hand (even if some advanced tools allow code-generation for the IG resolution).
As it is quite tedious to have to compute all matrices when a solution of MDH is investigated, a tool is provided to generate the C++ code. An example can be found in the file `dh_example.yml`:

```
notation: [alpha, a, theta, r]
joint:
1: [0, 0, q1, 0]
2: [-pi/2, 0, 0, q2]
3: [pi/2, 0, q3, r3]
4: [-pi/2, a4, q4+pi/2, 0]
```

The command to be run is: `rosrun ecn_manip dh_code.py <name of the file>`.
In the case of the above MDH table, it leads to these outputs:

```
// Generated pose code
const double c1 = cos(q[0]);
const double c1p3 = cos(q[0]+q[2]);
const double c4 = cos(q[3]);
const double s1 = sin(q[0]);
const double s1p3 = sin(q[0]+q[2]);
const double s4 = sin(q[3]);
M[0][0] = -s4*c1p3;
M[0][1] = -c4*c1p3;
M[0][2] = -s1p3;
M[0][3] = a4*c1p3 - q[1]*s1;
M[1][0] = -s4*s1p3;
M[1][1] = -s1p3*c4;
M[1][2] = c1p3;
M[1][3] = a4*s1p3 + q[1]*c1;
M[2][0] = -c4;
M[2][1] = s4;
M[2][2] = 0;
M[2][3] = r3;
M[3][0] = 0;
M[3][1] = 0;
M[3][2] = 0;
M[3][3] = 1.;
// End of pose code
```

```
// Generated Jacobian code
const double c1 = cos(q[0]);
const double c1p3 = cos(q[0]+q[2]);
const double s1 = sin(q[0]);
const double s1p3 = sin(q[0]+q[2]);
J[0][0] = -a4*s1p3 - q[1]*c1;
J[0][1] = -s1;
J[0][2] = -a4*s1p3;
//J[0][3] = 0;
J[1][0] = a4*c1p3 - q[1]*s1;
J[1][1] = c1;
J[1][2] = a4*c1p3;
//J[1][3] = 0;
//J[2][0] = 0;
//J[2][1] = 0;
//J[2][2] = 0;
//J[2][3] = 0;
//J[3][0] = 0;
//J[3][1] = 0;
//J[3][2] = 0;
J[3][3] = -s1p3;
//J[4][0] = 0;
//J[4][1] = 0;
//J[4][2] = 0;
J[4][3] = c1p3;
J[5][0] = 1.;
//J[5][1] = 0;
J[5][2] = 1.;
//J[5][3] = 0;
// End of Jacobian code
```

This is exactly what you would get by doing the manual computation from the initial table, which I hope is quite appreciated.