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:

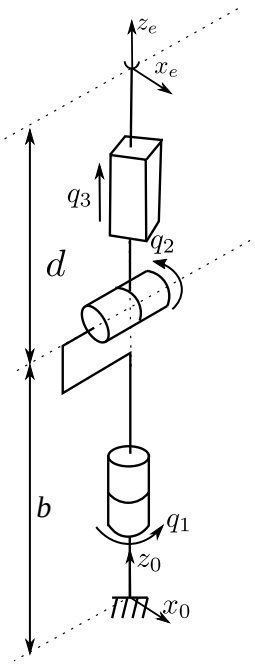
- `control.cpp`: main file where the control is done depending on the current robot mode
- `robot_turret.cpp`: model of the RRP turret robot
- `robot_kr16.cpp`: model of the industrial Kuka KR16 robot
- `robot_ur10.cpp`: model of the Universal Robot UR-10

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:

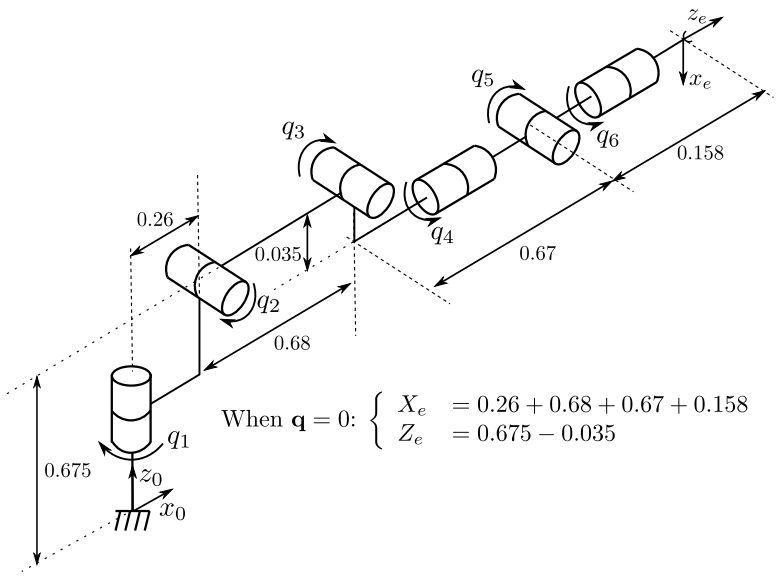


Figure 1: Model of the Kuka KR16 with imposed fixed and end-effector frames

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 include the 0.158 distance and potential rotations.

2.3 Universal Robot UR-10 anthropomorphic robot

This robot is a less classical 6R architecture, without spherical wrist. A tool is also attached with an arbitrary end-effector offset.
The Inverse Geometry is given for this robot. The constant transform ${}^{w}\mathbf{M}_{e}$ between the wrist and the end-effector should include the 0.1922 distance and potential rotations.

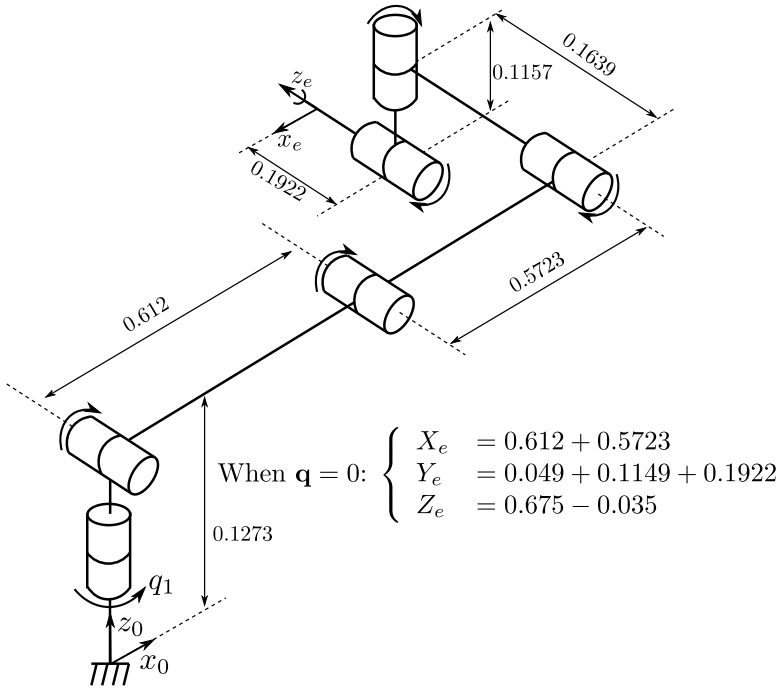


Figure 2: Model of the UR-10 with imposed fixed and end-effector frames

2.4 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
```

This will display two windows:

- RViz is the 3D simulator to show the current configuration of the robot. It will also display the base and end-effector frames, and the one that you compute. You can thus easily check if the computation is fine.
- The other window is composed of 4 panels:
  - Buttons at the top left to change the current exercise
  - Sliders at the bottom left to manually control the robot (exercises 1 and 2)
  - Sliders at the top to change the switching time of the position control (exercises 3+)
  - A plotting region with two tabs, one for the operational space and one for the joint space. The joint space is displayed in a normalized fashion, depending on the joint limits (upper and lower dotted lines)

3 Building the models

The main code

In the file `control.cpp`, the `if - else if` blocks correspond 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.
This velocity is ${}^{e}\mathbf{v}_{e}$ and is defined in the end-effector frame.
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 then to be mapped to the joint velocity in two steps:

1. Map to the fixed frame: ${}^{f}\mathbf{v}_{e}^{*} = \begin{bmatrix} {}^{f}\mathbf{R}_{e} & 0 \\ 0 & {}^{f}\mathbf{R}_{e} \end{bmatrix} {}^{e}\mathbf{v}_{e}^{*}$
   Call `M.getRotationMatrix()` to get this rotation.
   The `ecn::putAt` function may be of good use.
2. Map to the joint space: $\dot{\mathbf{q}} = \mathbf{J}^{+}.{}^{f}\mathbf{v}_{e}^{*}$
   Call `robot->fJe(q).pseudoInverse()` to get the Jacobian pseudo-inverse

You can check that the end-effector is able to follow straight 3D lines in x, y or z direction in its own frame. 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. The robot will try to have all joints reach their desired value as soon as possible, which will lead to a discontinuous motion.

In practice, this exercise is to check that the Inverse Geometric Model is correct: the robot should reach the desired pose.

When computing the inverse geometry, try to find a solution as near as possible to the passed `q0`. It should also be within the joint limits, that are available in `q_min` and `q_max`.

Many useful functions are available to solve classical trigonometric equations. They are detailed in Appendix C.

A given potential solution should be stored using `addCandidate({q1, q2, ..., qn});`
At the end of the Inverse Geometry function, use `return bestCandidate(q0);` that will pick the candidate that is the nearest from the current joint position.

Exercise 4: Interpolated point-to-point control

In this exercise the previous pose setpoint `M0` and its corresponding joint position `q0` should be used to interpolate the joint positions.

Here the goal is to compute the current setpoint $\mathbf{q}_c = \mathbf{q}_0 + p(t)(\mathbf{q}_f - \mathbf{q}_0)$. The interpolation function $p(t)$ was defined during the lectures and should take into account the maximum joint velocity and acceleration that are stored in the `vMax` and `aMax` vectors. Basically this amounts to computing the minimum time `tf` needed to reach $\mathbf{q}_f$ from $\mathbf{q}_0$. This computation should be done in the `if(robot->newRef())` and then used to compute the current `qCommand`.

Exercise 5: Operational control through Inverse Geometric Model

Exercises 3 and 4 lead to the correct pose but not in a straight 3D line. Indeed the robot is only controlled in the joint space with no constraints between the two extreme poses.
In this exercise, the goal is to perform the interpolation not in the joint space but in the Cartesian space. In practice, we will compute many poses between `M0` and `Md` and command the robot to reach sequentially all these poses. As they will be pretty close one from each other, the resulting motion will be close to a straight line.
The `robot->intermediaryPose(M0, Md, alpha)` function returns a pose between `M0` and `Md`, where `alpha` is between 0 and 1 and should be computed from t, t_0 and $t_f = 1$ s.
Then, the inverse geometry of this pose should be sent to the robot as joint position setpoint.

Exercise 6: Operational control through Jacobian

In this exercise the goal is still to follow a straight line between the two points, but this time by sending a joint velocity command.

As seen in class, this command should make the end-effector approach its desired pose. The steps are:

1. Compute the pose error in the desired frame: ${}^{e*}\mathbf{M}_e = {}^{f}\mathbf{M}_{e*}^{-1}.{}^{f}\mathbf{M}_e$
   - in the code, ${}^{f}\mathbf{M}_{e*}$ corresponds to `Md` and ${}^{f}\mathbf{M}_e$ to `M`
   - In practice we use the $(\mathbf{t}, \theta\mathbf{u})$ representation with `p.buildFrom()`
2. Compute the desired linear and angular velocities: $v = -\lambda {}^{f}\mathbf{M}_{e*}\mathbf{t}, \quad \omega = -\lambda {}^{f}\mathbf{M}_e(\theta\mathbf{u})$
3. Map these velocities to the joint space: $\dot{\mathbf{q}} = \mathbf{J}^{+} \begin{bmatrix} v \\ \omega \end{bmatrix}$

λ is a gain that can be changed from the GUI. Increasing it will lead to a faster convergence. It can be obtained through `robot->lambda()`.

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;`

3D transforms can be easily changed between homogeneous matrix, rotation matrix, translation vector or $\theta\mathbf{u}$ vector. Another type called a pose vector consists in the full $(\mathbf{t}, \theta\mathbf{u})$ vector of dimension 6:

```
vpHomogeneousMatrix M;
vpPoseVector p;
vpRotationMatrix R;
vpTranslationVector t;
vpColVector tu(3);

M.getRotationMatrix()*t;        // extract the rotation part from M and multiply with t
p.buildFrom(M);                 // build (t, theta-u) from M
t = M.getTranslationVector();   // extract the translation part

// copy the theta-u part of p into tu
for(int i = 0; i < 3; ++i)
        tu[i] = p[i+3];

// compute inverse transform
M = M.inverse();
```

Two utility functions are provided to put a matrix or vector into a bigger one:

- `void ecn::putAt(M, Ms, r, c)`: Writes matrix `Ms` inside matrix `M`, starting from given row and column.
- `void ecn::putAt(V, Vs, r)`: Writes column vector `Vs` inside vector `V`, starting from given row. These two functions do nothing but print an error if the submatrix or vector does not fit in the main one.

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.

C Trigonometric solvers

The code contains solvers for all types of trigonometric equations (except type 1 that is obvious). They take as argument the coefficients of the equations and also the joint limits in order to prune impossible solutions. They return:

- A `std::vector<double>` for equations of 1 unknown (types 2 & 3).
  We can go through the valid solutions with:

```
// assume q1 should ensure type 2: X.sin(q1) + Y.cos(q1) = Z

for(double q1: solveType2(X, Y, Z, q_min[0], q_max[0]))
{
        // here q1 is a valid solution, may be used to deduce other joint values
}
```

- A `std::vector<JointSolution>` for equations of 2 unknowns (types 4 to 8).
  In this case, the valid solutions can be explored with:

```
// assume q1 and q2 should ensure type 6 equation:
//              W.sin(q2) = X.cos(q1) + Y.sin(q1) - Z1
//              W.cos(q2) = X.sin(q1) - Y.cos(q1) - Z1

for(auto qs: solveType6(X, Y, Z1, Z2, W, q_min[0], q_max[0], q_min[1], q_max[1]))
{
        // here a (q1,q2) solution is available in qs.qi and qs.qj
}
```

Additional functions can check and even update (by $\pm 2\pi$) an angular joint value to that is fits within the joint limits:

```
// assume q1 was computed by hand, check joint limits
if(inAngleLimits(q1, q_min[0], q_max[0]))
{
        // here q1 is ok
}

// we have the sinus s1 and cosinus c1 of q1
if(inAngleLimits(q1, s1, c1, q_min[0], q_max[0]))
{
        // here q1 is ok and s1 and c1 absolute values are <= 1
}
```

Below are listed the syntax for all trigonometric solvers:

Type	Equation	Syntax	Returns (vector of)
1	$Xq_i = Y$	does not exist	q_i
2	$Xs_i + Yc_i = Z$	`solveType2(x, y, z, q_min, q_max)`	q_i
3	$X_1 s_i + Y_1 c_i = Z_1$ $X_2 s_i + Y_2 c_i = Z_2$	`solveType3(x1, y1, z1, x2, y2, z2, q_min, q_max)`	q_i
4	$X_1 q_j s_i = Y_1$ $X_2 q_j c_i = Y_2$	`solveType4(x1, y1, x2, y2, qi_min, qi_max, qj_min, qj_max)`	(q_i, q_j)
5	$X_1 s_i = Y_1 + Z_1 q_j$ $X_2 c_i = Y_2 + Z_2 q_j$	`solveType5(x1, y1, z1, x2, y2, z2, qi_min, qi_max, qj_min, qj_max)`	(q_i, q_j)
6	$Ws_j = Xc_i + Ys_i + Z_1$ $Wc_j = Xs_i - Yc_i + Z_2$	`solveType6(x, y, z1, z2, w, qi_min, qi_max, qj_min, qj_max)`	(q_i, q_j)
7	$W_1 c_j + W_2 s_j = Xc_i + Ys_i + Z_1$ $W_1 s_j - W_2 c_j = Xs_i - Yc_i + Z_2$	`solveType7(x, y, z1, z2, w1, w2, qi_min, qi_max, qj_min, qj_max)`	(q_i, q_j)
8	$Xc_i + Yc_{ij} = Z_1$ $Xs_i + Ys_{ij} = Z_2$	`solveType8(x, y, z1, z2, qi_min, qi_max, qj_min, qj_max)`	(q_i, q_j)