This assignment is concerned with the mathematical description and definition of inverse kinematics of serial link robotic arms.

The following exercises are solved with MATLAB and utilize the Robotics System Toolbox from Mathworks. Some tasks involve the use of ROS on a Linux (Ubuntu) system which is also available in the Retina Pool. Note, Matlab Linux uses different keyboard shortcuts per default (copy, paste, etc.). You might change it at *MATLAB - Preferences - Keyboard - Shortcuts - Active Settings: Windows Default Set*.

Before the assignment, read this document entirely and complete the quiz on Forward Kinematics in the Moodle workspace of the course. In case you are unable to answer the questions in the quiz correctly go back to the lecture slides and please carefully study the chapters 2.12 on the inverse kinematic problem in the book by Siciliano et al [1]. Further recommended reading is the chapter on kinematics in the Springer Handbook of Robotics [2].

Inverse Kinematics

The inverse kinematics problem is the opposite of the forward kinematics problem: Given the desired end effector pose determine the joint configuration to achieve that pose. Inverse kinematics either rely on a closed-form solution or a numerical solution. Analytical solutions provide a set of equations that fully describe the connection between the end effector position and the joint angles. For standard serial manipulators, such as robot arms with a spherical wrist closed form solutions of the inverse kinematics exist.

Numerical solutions are universal as they rely on numerical algorithms, and provide solutions even if no closed-form solution is available. For a given pose there might be multiple solutions, e.g. elbow-up and elbow-down posture, or no solution at all, e.g. if the end effector pose is outside the manipulators workspace.

For a numerical solution the inverse kinematics problem is formulated as an optimization problem. In fact the objective is to minimize the error between the target transform $\mathbf{H}_t$ for the end effector and the forward kinematics of a joint configuration $\mathbf{H}_e(\mathbf{q})$. For that purpose we decompose the pose error into a position and a rotation part. The position error is simply the distance of the origin of both transforms

$$\mathbf{e_p}(\mathbf{q}) = [e_x \;\; e_y \;\; e_z]^T = [p_{xt} - p_{xe} \;\; p_{yt} - p_{ye} \;\; p_{zt} - p_{ze}]^T \tag{1}$$

with
$[p_{xt} \;\; p_{yt} \;\; p_{zt}]^T = [\mathbf{H}_t(1,4) \; \mathbf{H}_t(2,4) \; \mathbf{H}_t(3,4)]^T$
and
$[p_{xe} \;\; p_{ye} \;\; p_{ze}]^T = [\mathbf{H}_e(\mathbf{q})(1,4) \; \mathbf{H}_e(\mathbf{q})(2,4) \; \mathbf{H}_e(\mathbf{q})(3,4)]^T$

For the rotation part the relative orientation matrix $R_d = R_t R_e(\mathbf{q})'$ is converted to an axis angle representation $[\alpha \;\; w_x \;\; w_y \;\; w_z]^T$. The orientation error is given by

$$\mathbf{e_w}(\mathbf{q}) = [e_{wx} \;\; e_{wy} \;\; e_{wz}]^T = [\alpha w_x \;\; \alpha w_y \;\; \alpha w_z]^T \tag{2}$$

The overall error is a six-dimensional vector $\mathbf{e}(\mathbf{q}) = [e_x \; e_y \; e_z \; e_{wx} \; e_{wy} \; e_{wz}]^T$. The Robotics system toolbox provides a helper function to calculate the error vector $\mathbf{e}$ between two transforms

```
robotics.manip.internal.IKHelpers.poseError(tformt, tformq)
```

The objective is to minimize the error norm in the least squares sense.

$$\min_{\mathbf{q}} \frac{1}{2}||\mathbf{e}(\mathbf{q})|| = \min_{\mathbf{q}} \frac{1}{2}(e_x^2 + e_y^2 + e_z^2 + e_{wx}^2 + e_{wy}^2 + e_{wz}^2) \tag{3}$$

A more general error is obtained by weighting the individual errors

$$\min_{\mathbf{q}} \frac{1}{2}\mathbf{e}_W(\mathbf{q}) = \min_{\mathbf{q}} \frac{1}{2}\mathbf{e}'\mathbf{W}\mathbf{e} \tag{4}$$

in which $\mathbf{W}$ is a positive definite matrix, often diagonal. In fact if a feasible solution $\mathbf{q}^*$ of the inverse kinematics problem exists, namely the target pose is within the robot workspace then the pose error becomes zero

$$\mathbf{e}(\mathbf{q}^*) = \mathbf{0} \tag{5}$$

The problem (4) constitutes a non-linear least squares problem for which efficient optimization algorithms exist. The Jacobian is the matrix of first order derivatives of a vector valued function. In our case we are interested in partial derivatives of the error vector w.r.t. to joint angles $J_{ij} = \frac{\partial e_i}{\partial q_j}$ that form the Jacobian $\mathbf{J}$. The current solution $\mathbf{q}$ is improved with a Levenberg-Marquardt (damped least squares) step $\mathbf{q}' = \mathbf{q} + \Delta\mathbf{q}$ with $\Delta\mathbf{q}$ obtained from the algebraic solution of

$$(\mathbf{J}^T\mathbf{J} + \lambda\mathbf{I})\Delta\mathbf{q} = \mathbf{J}^T\mathbf{e} \tag{6}$$

The `InverseKinematics` class creates an inverse kinematics (IK) solver to calculate joint configurations for a desired end effector pose based on a specified rigid body tree model. This code generates an inverse kinematics solver object for the `robotics.RigidBodyTree` object and determines the inverse kinematics solution as a configuration object for the target pose `targetpose`.

```
ik = robotics.InverseKinematics('RigidBodyTree',robot);
weights = ones(6,1);
initialpose = robot.homeConfiguration;
randconf=robot.randomConfiguration;
targetpose = robot.getTransform(randconf,'ee_link');
[targetsol, solnInfo] = ik('ee_link',targetpose,weights,initialpose);
robot.show(targetsol);
```

In this case, `targetpose` originates from a random configuration `randconf` and forward kinematics.

The Robotics System Toolbox provides a non documented helper function `robotics.manip.internal.IKHelpers.poseError` to calculate the pose error according to to equations (1) and (2) between two transforms. The following code checks the resulting configuration `targetsol` by transforming it into the taskspace with forward kinematics and comparing it to `targetpose`:

```
% pose error in task space
poseerror=robotics.manip.internal.IKHelpers.poseError(targetpose,...
             robot.getTransform(targetsoln,'ee_link'));
% joint space error
jointerror=JointConf2JointVec(randconf)-JointConf2JointVec(targetsol);
```

If the original configuration `randconf` is known, the check can also be performed in the joint space between `randconf` and `targetsol`. However, `randconf` is usually unknown in real applications.

The helper function

```
function [ q ] = JointConf2JointVec( configuration )
```

provided in Moodle extracts the joint vector from the fields `JointPosition` in the configuration structure array.

Now, please complete tasks 1 to 9 in the template.

Nonlinear Least Squares Problems

You might skip this section if you are already familiar with local optimization and in particular non-linear least squares problems from the courses 'Scientific Programming in Matlab' or 'Datenbasierte Modellierung und Optimierung'.

In least-squares problems, the objective function f has a particular structure that is exploited by the optimization algorithm

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j(x)^2 \tag{7}$$

in which $\mathbf{r}(x) = (r_1(x), \ldots, r_m(x))^T$ denotes a vector of residuals. Non-linear least squares problems are particular relevant for regression in which a parametrized model is identified from observed data. In this case $r_j(x)$ measures the disagreement between the prediction of the model and the observed output over a set of training data $\{(x_i, y_i)\}$. By minimizing Eq. (7), the parameters of the model are selected such that it matches the data in a least squares sense. Numerical optimization schemes such as Levenberg-Marquardt exploit the particular structure of f and its derivatives.

The gradient of f is given by

$$\nabla f(x) = \sum_{j=1}^{m} r_j(x)\nabla r_j(x) = J(x)^T r(x) \tag{8}$$

in which the Jacobian $J(x) = [\frac{\partial r_j(x)}{\partial x_i}]_{i=1,2,\ldots,n}^{j=1,2,\ldots,m}$ denotes the matrix of first order partial derivatives.

In least-squares problems the Hessian $\nabla^2 f(x)$ matrix can be approximated from the knowledge of the Jacobian.

$$\begin{aligned} \nabla^2 f(x) &= \sum_{j=1}^{m} \nabla r_j(x)\nabla r_j(x)^T + \sum_{j=1}^{m} r_j(x)\nabla^2 r_j(x)^T && (9)\\ &= J(x)^T J(x) + \sum_{j=1}^{m} r_j(x)\nabla^2 r_j(x)^T && (10) \end{aligned}$$

The first term $J(x)^T J(x)$ often dominates the second term in particular if the residuals $r_j(x)$ become small. Algorithms such as Levenberg-Marquardt exploit this property of the Hessian for non-linear least squares problems.

The Levenberg-Marquardt algorithm solves non-linear least squares problems that arise in least squares regression. The Levenberg-Marquardt method can be interpreted as a trust region approach. Assuming a spherical trust region S_k of radius Δ_k each Levenberg-Marquardt step solves the quadratic problem

$$\min_p \frac{1}{2}||J_k p + r_k||^2 \text{ with } p \in S_k \tag{11}$$

This is equivalent of choosing a model function:

$$m_k(p) = \frac{1}{2}||r_k||^2 + p^T J_k^T r_k + \frac{1}{2} p^T J_k^T J_k p \tag{12}$$

The optimal Gauss-Newton step of Eq. (12) is given by

$$J_k^T J_k p_k^{GN} = -J_k^T r_k \tag{13}$$

In case $p^{GN} \in S_k$ ($||p_k^{GN}|| < \Delta$) it coincides with the solution of Eq. (12). Otherwise, there is a $\lambda > 0$ such that the solution $p = p^{LM}$ of Eq. (12) satisfies $||p|| = \Delta$ and

$$(J_k^T J_k + \lambda I)p = -J_k^T r_k. \tag{14}$$

In practice λ is determined by a Cholesky decomposition of $(J_k^T J_k + \lambda I)$.

`lsqnonlin` solves non-linear least-squares curve fitting problems of the form in Eq. (7). You can provide optional lower and upper bounds `lb` and `ub` on the components of x.

Rather than to compute the sum of squares $f(x)$, `lsqnonlin` requires the user-defined function to compute the vector-valued function of residuals $\mathbf{r}(x) = (r_1(x), \ldots, r_m(x))^T$.
`x = lsqnonlin(fun,x0)`
starts at the point `x0` and finds a minimum of the sum of squares of the functions described in `fun`. The function `fun` should return a vector of values and not the sum of squares of the values, in case of the Rosenbrock function $r_1(x_1, x_2) = 10(x_2 - x_1^2),\ r_2(x_1, x_2) = 1 - x_1$. `lsqnonlin` operates with `'Algorithm' 'levenberg-marquardt'` or `'trust-region-reflective'` (default). Notice that `'levenberg-marquardt'` does not handle constraints.

Numerical Solution of Inverse Kinematics

This assignment deals with the numerical solution of inverse kinematics with the help of the Matlab optimization toolbox. The solution of the inverse kinematics problem is a non-linear least squares problem of equation (7), in particular the cost function w.r.t. to joint vector $\mathbf{q}$ becomes

$$E(\mathbf{q}) = \frac{1}{2}(e_x^2 + e_y^2 + e_z^2 + e_{wx}^2 + e_{wy}^2 + e_{wz}^2) \tag{15}$$

namely to minimize the norm of the pose error in target space $\mathbf{e}$.

Now, please complete tasks 10 to 14 in the template.

Redundant Manipulator

A redundant manipulator has more than six degrees of freedom which means that it has additional joint parameters that allow the configuration of the robot to change while it holds its end effector in a fixed position and orientation. A typical redundant manipulator has seven joints, for example three at the shoulder, one elbow joint and three at the wrist. This manipulator can move its elbow around a circle while it maintains a specific position and orientation of its end effector.

In case of a redundant manipulator with more degrees of freedom than task space dimensions there is an infinite number of solutions of the inverse kinematics problem. The inverse kinematics problem is of particular interest in the case of a redundant manipulator since it admits infinite solutions. The above inverse kinematics algorithms can be extended to redundant manipulators by adopting a task space augmentation technique. Formally, an additional objective or constraint is imposed to be satisfied along with the end effector task. Typical objectives include obstacle avoidance, limited joint range, manipulability measures and singularity avoidance.

Let us consider limited joint range, that means we would like the joint configuration to be as close as possible to the home configuration $\mathbf{q} = \mathbf{0}$. Assume that $\mathbf{q}$ is a solution of Eq. (5). If you add a small vector $\mathbf{q_n}$ that lies in the null space of the Jacobian $\mathbf{J}$ then $\mathbf{q}' = \mathbf{q} + \mathbf{q_n}$ also satisfies Eq. (5). The small joint displacement $\mathbf{q_n}$ causes no motion of the end effector as it lies in the null space of the Jacobian. We can utilize this property to optimize the secondary objective such as to find a joint vector $\mathbf{q}$ that satisfies (3) and has minimal norm $||\mathbf{q}||$.

Explore the redundancy of the 7-DOF Sawyer robot to optimize the joint range of the inverse kinematics solution namely to find a solution of minimal norm of the first three joint variables $||\mathbf{q}(1:3)||$. The projection of the joint vector $\mathbf{q}$ into the null space of the Jacobian $\mathbf{J}$ is given by

$$\mathbf{q}_{null} = (\mathbf{I} - \mathbf{J}^{\dagger}\mathbf{J})\mathbf{q} \tag{16}$$

in which $\mathbf{J}^{\dagger}$ denotes the pseudo inverse of a non-square matrix.

Now, please complete tasks 15 to 18 in the template.

References

[1] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[2] Kenneth Waldron and James P. Schmiedeler. Kinematics. In Bruno Siciliano and Oussama Khatib, editors, *Springer Handbook of Robotics*, pages 9–33. Springer, 2008.