An Inverse Kinematics Problem Solver Based on Screw Theory for Manipulator Arms

Bartek Łukawski, Ignacio Montesino Valle, Juan G. Victores, Alberto Jardón and Carlos Balaguer

Robotics Lab, Department of Systems Engineering and Automation, Universidad Carlos III de Madrid

{blukawsk, imontesi, jcgvicto, ajardon, balaguer}@ing.uc3m.es

Abstract

Several methodologies exist for solving the inverse kinematics of a manipulator arm. Basing on screw theory, it is possible to efficiently obtain complete and exact solutions. An open-source C++ implementation of an automated problem solver of this kind is introduced, and a comparative with selected known algorithms is established using the TEO humanoid robot platform by Universidad Carlos III de Madrid. The Orocos Kinematics and Dynamics Library is used for geometry and motion-related operations.

Keywords: robotics, screw theory, inverse kinematics, manipulator arm, Orocos KDL.

1. SCREW THEORY

The kinematic transformation from task space to joint configuration in serial-chain manipulators is a common problem in robotics with a twofold approach. Closed-form, i.e. algebraic or geometric methods are desirable due to efficiently and accurately providing all available solutions. As a drawback, those are mostly ad hoc techniques that can be applied to certain robot types, only. Numeric methods, on the other hand, trade speed, accuracy and completeness for generalization to any kinematic structure with an arbitrary number of degrees of freedom, therefore including redundant robots [1].

It is of interest in the field of robotics to further explore closed-form methods that allow certain degree of generalization while still retaining the aforementioned advantages. Numeric solutions may not be suitable for real-time applications due to the uncertainty of the algorithm convergence. In addition, singularities can be hard to avoid or mitigate for certain methods. On the ground of the mathematical theory of screws, founded upon the theorem of the displacement of a rigid body [2], a strictly geometric methodology can be developed to effectively and efficiently solve inverse kinematics for a range of common robot architectures [3].

Several mathematical principles lie at the mathematical basis of screw theory. An arbitrary rotation of a vector in space R_ω, given an axis ω and angle of rotation θ, can be defined by the Euler-Rodrigues formula (Equations 1, 2).

$$R_\omega(\theta) = e^{\hat{\omega}\theta} = I_3 + \hat{\omega}\, sin\theta + \hat{\omega}^2\,(1 - cos\theta) \quad (1)$$

$$\hat{\omega} = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \quad (2)$$

By further applying the Chasles' theorem, any rigid body motion can be described by a translation along a line and a rotation about that same line. The joint effect of said translation and rotation motion is referred to as "screw", defined by the coordinates of an axis and the angle of rotation performed about it.

A screw is therefore the mathematical tool that describes a pose transformation in the context of screw theory. The differentiation of a screw results in a "twist" ξ, i.e. an infinitesimal screw, which is encoded by a translation vector ν and the direction of the screw (rotation) axis ω as $\xi_{6x1} = [\nu\ \omega]^T$.

A connection can be established between homogeneous and exponential transformations in $SE(3)$ via the Lie algebra (M.S. Lie, 1842–1899). The exponential formula, also called *matrix exponential*, is a link between the usual representation of robot frames of reference, commonly encoded through the Denavit-Hartenberg algorithm [4], and motion described by screws; see Equations 3, 4.

$$H(\theta_i) = e^{\hat{\xi}_i\theta_i} \quad (3)$$

$$e^{\hat{\xi}\theta} = \begin{bmatrix} e^{\hat{\omega}\theta} & (I - e^{\hat{\omega}\theta})(\vec{\omega} \times \vec{\nu}) + \vec{\omega}\vec{\omega}^T\vec{\nu}\theta \\ 0 & 1 \end{bmatrix} \quad (4)$$

To represent a kinematic chain, a sequence of homogeneous transformations between the inertial and tool frames H_{ST} is referred to in the screw theory realm as the *product of exponentials* [5] (PoE). Said product expresses the relative motion of a rigid body and must be complemented with an additional $H_{ST}(0)$ transformation from base to tip; see Equation 5.

$$H_{ST}(\theta) = \prod_i e^{\hat{\xi}_i\theta_i}\, H_{ST}(0) \quad (5)$$

https://doi.org/10.17979/spudc.9788497498418.0864

2. CANONICAL PROBLEMS

The product of exponentials formula defines a sequence of steps to be performed in order to obtain the forward kinematics of the chain given a joint configuration as input (θ_i). By roughly following the reverse order and applying any necessary operations on both sides of the formula (such as left- or right-side vector multiplications and substractions), the inverse kinematics of the chain are solved as well. Each step is treated as a separate problem (subproblem) and approached in a specific way, also accounting for any simplifications characteristic for that case.

Individually, the subproblems aim to solve a simple, canonical case using a purely geometric approach. By combining them and applying (using any convenient algebraic operations) over the original problem statement given by the product of exponentials, the full set of joint-space solutions is obtained.

Originally, three canonical problems were defined by Paden and Kahan [6], [7], known as the Paden-Kahan (PK) subproblems.

- **PK1:** Rotation about a single axis. Single solution. Equation 6.
- **PK2:** Rotation about two subsequent crossing axes. Dual solution. Equation 7.
- **PK3:** Rotation about a single axis to a given distance δ. Dual solution. Equation 8.

$$e^{\hat{\xi}\,\theta}\,\vec{p} = \vec{k} \tag{6}$$

$$e^{\hat{\xi_1}\,\theta_1}\,e^{\hat{\xi_2}\,\theta_2}\,\vec{p} = \vec{k} \tag{7}$$

$$\left\| e^{\hat{\xi}\,\theta}\,\vec{p} - \vec{k} \right\| = \delta \tag{8}$$

The available set of subproblems is not limited to the above mentioned. Additional cases can be studied and their resolution proposed, if possible. For instance, an analogous collection has been introduced by Pardos-Gotor [3] (PG) targeting prismatic joints, and one rotation case not included in PK.

- **PG1:** Translation along a single axis. Single solution. Equation 6.
- **PG2:** Translation along two subsequent crossing axes. Single solution. Equation 7.
- **PG3:** Translation along a single axis to a given distance. Dual solution. Equation 8.
- **PG4:** Rotation about two subsequent parallel axes. Dual solution. Equation 7.

Cases PK1, PK3 and PG1 are subject to simplification. In Equation 6, the screw rotation motion does not affect any point p that lies on axis ω. In Equation 8, the same reasoning applies for any k lying on ω. Regarding PG1, any equidistant pair of points p, k can be picked. The goal of these operations is to drop one or more terms of the product of exponentials, thus simplyfing subsequent steps. The failure to find a convenient case of simplification can render the inverse kinematics problem insoluble through the application of screw theory techniques.

3. OROCOS KDL

Various C++ libraries based on Eigen exist to represent Lie groups and related mathematical artifacts [8], [9], [10]. MATLAB$^{\text{TM}}$ toolboxes can be found that fulfill the same goal [3], [11]. Graphical interfaces provide a means to properly visualize and understand the geometrical implications of screws [12]. However, a C++ API for solving inverse kinematics problems in robotics regardless of the mathematical backend was lacking at the time of conceiving this paper.

To overcome that, the problem solver described in this work relies on the well-established Kinematics and Dynamics Library (KDL), part of the Open Robot Control Software (Orocos) project [13]. This C++/Python library is widely used among the Robot Operating System (ROS) community and beyond.

Three components are building blocks of this library:

- A collection of geometric primitives for describing vectors, rotation matrices, homogeneous transformation matrices, twists, etc. and common operations between them.
- A collection of kinematic families to describe robot joints, segments, chains and tree structures. Here, a family of solvers (forward and inverse kinematics and dynamics) is provided to target various domains (position, velocity, acceleration).
- A collection of motion-related classes: paths, trajectories and velocity profiles.

For the purpose of the problem solver introduced here, geometric primitives are extensively used to avoid re-implementing those tools. In fact, its public API exposes methods to deal with KDL frames and vectors, and also conversion utilities for that matter.

4. IMPLEMENTATION

A C++ implementation of a closed-form, screw theory-based algorithm following the previous premises is described here. It is conceived as an open-source library; sources are available on GitHub [14].

All seven subproblems introduced earlier have been implemented and extensively tested on a set of popular, non-redundant manipulators listed and solved in [3], in addition to the TEO platform.

The core of the library is represented by the collection of C++ classes shown in Figure 1.

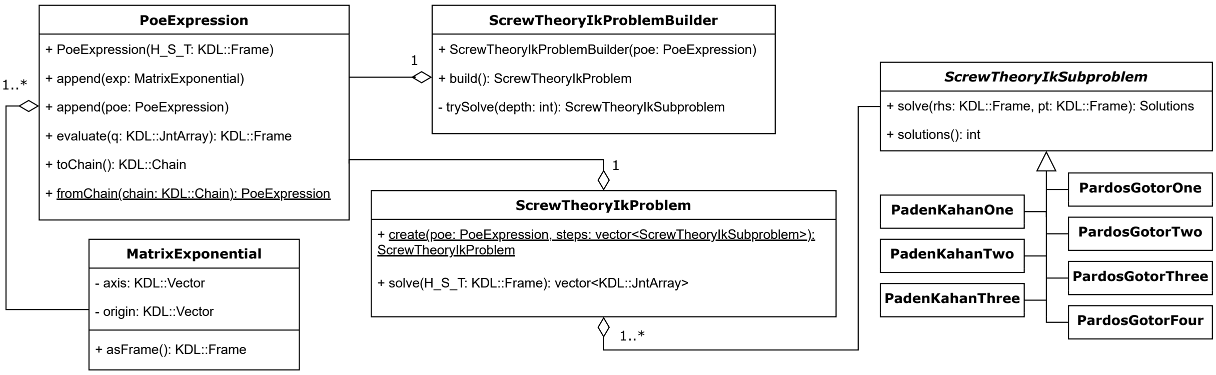

Figure 1: UML class diagram.

In more detail:

- `MatrixExponential` represents the exponential transformation of a rigid body in space with a vector describing an axis and the angle to rotate about it. A helper method `asFrame` is provided to evaluate the exponential matrix for an input θ and return a `KDL:Frame` (representing a homogeneous tranformation).
- `PoeExpression` uses a sequence of `MatrixExponentials` to describe a kinematic chain represented by the product of exponentials of the screw axes. The `evaluate` method effectively performs forward kinematics. Conversion methods `fromChain` and `toChain` are provided to expose this class as an `KDL::Chain`.
- `ScrewTheoryIkProblem` represents the already configured IK solver instance that knows what steps need to be performed by the `solve` method. Each step is a subclass of `ScrewTheoryIkSubproblem`.
- `ScrewTheoryIkSubproblem` is an interface to specific subproblem implementations. Here, `solve` provides as many solutions as expected by the concrete subclass.
- `ScrewTheoryIkProblemBuilder` is a single-method class responsible for the creation of a `ScrewTheoryIkProblem` instance. It is used to configure the solver and to `build` the subproblems that will be used to solve the main inverse kinematics problem.

Aditional classes are provided to manage and select the optimal solution according to predefined criteria, using a guess joint configuration (usually the current configuration of the robot) as input. Currently implemented selectors include:

- Least overall angular displacement criterion: the solution that entails the shortest motion across all joints is selected.
- Humanoid gait: specifically designed for the TEO platform, it prevents unnatural leg motion (such as a knee joint bent backwards). As a secondary criterion, the previous method is used on passing candidate solutions.

Besides purely geometrical operations, this library features a sequential problem solver that aims to assemble a pipeline of operations on exponential matrices using previously selected compatible subproblems. In this process, all appliable and necessary simplifications are performed in a brute-force manner, using a set of random points and several checks on screw axes until a valid subproblem is found. Since this configuration process is done by `ScrewTheoryIkProblemBuilder`, the actual problem solver represented by `ScrewTheoryIkProblem` is aware of the correct sequence (if possible) of steps since instantiation, thus aiming to reduce computation efforts in $solve$.

The pipeline begins with the representation of the product of exponentials as in Equation 5. Equation 9 represents an intermediate step having all unknown terms arranged to the left side, and those known and the invariants to the right side. Progressively, unknown terms will be moved to the right side either pre- or post-multiplying.

$$\begin{aligned} \prod_{i=j}^{j+k} e^{\hat{\xi}_i \theta_i} = \left(\prod_{i=1}^{j-1} e^{\hat{\xi}_i \theta_i} \right)^{-1} H_{ST}(\theta)\, H_{ST}^{-1}(0) \dots \\ \dots \cdot \left(\prod_{i=j+k+1}^{N} e^{\hat{\xi}_i \theta_i} \right)^{-1} \end{aligned} \tag{9}$$

N is the number of joints, j the index of the first unknown term, and $k+1$ the number of unknown terms.

As a limitation to this algorithm, and inherently to the screw theory methodology, it is not always possible to obtain inverse kinematics. The implemented collection of subproblems could be expanded to cover additional cases, yet some mechanisms (e.g. redundant manipulators) will still remain unreachable to this method. In order to solve TEO limbs, a workaround was introduced: if the input product of exponentials is deemed unsolvable, it is reversed (the last exponential term is now the first and so on) and tested again.

5. SOLUTION SEARCH ALGORITHM

The basics of `ScrewTheoryIkProblemBuilder` are described here. The goal is to find a sequence of subproblems that allow to solve inverse kinematics given a product of exponentials.

Snippet 1 corresponds to the main entry point of the builder class. Here, solution search (Snippet 2) is performed first on the input product of exponentials, and then on the reversed product of exponentials.

Snippet 1 Main builder entry point

```
function BUILD($poe$)
    $known\_poe\_terms \leftarrow 0$
    $steps \leftarrow search\_solutions()$

    if $known\_poe\_terms$ is $poe.size()$ then
        return $create(poe, steps)$
    end if

    $poe.reverse\_self()$
    $known\_poe\_terms \leftarrow 0$
    $steps \leftarrow search\_solutions()$

    if $known\_poe\_terms$ is $poe.size()$ then
        return $create(poe, steps)$
    end if

    return $null$
end function
```

Snippet 2 dives into the main solution search routine. A collection of test $points$ is prepared (see Snippet 3) in order to feed them to $simplify$ (see Section 2) and try_solve (Snippet 4). For pseudocode readability, these points are not explicitly represented as inputs to said functions, and further terms $test\_pointsX$ refer to points picked from said collection.

The implemented subproblems require a varying number of points to operate with. Since the algorithm performs brute force on searching a valid solution, it first traverses the entire point collection using a single point at once, then it enters the next $stage$ to operate with two points at once. No subproblem uses more than two points, hence here MAX_POINTS is hardcoded to 2 and might be expanded in the future if more subproblems are incorporated.

In the process of solution search, a simplification is attempted if possible, then all subproblems are tested for suitability. If the point (or points) do not lead to a valid solution, the simplified product of exponentials is reset to its original state and the next set of points is tested instead.

Snippet 3 builds the collection of points to operate with. On iterating over the terms of the product of exponentials, the origin point of each axis is selected as well as the point of intersection between axes of two different terms. To help resolve certain subproblems, a random point on each axis is added, and also the robot base (origin) and TCP.

Snippet 2 Main solution search routine

```
function SEARCH_SOLUTIONS
    $points \leftarrow search\_points(poe)$
    $init\_test\_points(points[0])$
    $steps \leftarrow \emptyset$
    $n\_points \leftarrow 1$

    while ($n\_points <= MAX\_POINTS$
    and $unknown\_poe\_terms > 0$) do
        $refresh\_simplification\_state()$
        $simplify(poe, depth)$
        $subproblem \leftarrow try\_solve(poe, depth)$

        if $subproblem$ is not $null$ then
            $steps.insert(subproblem)$
            $n\_points \leftarrow 1$
            continue
        end if

        for $stage = 1 : MAX\_POINTS$ do
            if all points tested then
                $reset\_test\_points()$

                if $stage$ is $n\_points$ then
                    $n\_points \leftarrow n\_points + 1$
                    break
                end if
            else
                $update\_test\_points()$
                break
            end if
        end for
    end while

    return $steps$
end function
```

Snippet 4 seeks a valid subproblem for its inclusion in the solver's pipeline. The characteristics of each one are taken into account, including the number of required test points, i.e. n_points (either one or two in the current implementation), and the number of unknown terms each subproblem is able to solve at once, i.e. $unknown\_poe\_terms$ (also either one or two). Additional checks are carried out when needed, such as querying the motion type (rotation or translation), testing whether screw axes are parallel, etc.

6. EXPERIMENTS

The performance of the proposed screw theory implementation has been compared with two numeric inverse kinematics solvers found in Orocos KDL. The kinematic chain tested against corresponds to the 6-DOF left arm of TEO, a full-size 28-DOF humanoid robot from Universidad Carlos III de Madrid (Figure 2). Out of all eight possible solutions, the one that entails the least joint displacement relative to the initial configuration is chosen.

Snippet 3 Point search and validation

```
function SEARCH_POINTS($poe$)
    $points \leftarrow \emptyset$
    $points.insert(\{0, 0, 0\})$

    for each $e1 \in poe$ do
        $points.insert(e1.get\_origin())$

        for each $e2 \in poe$ do
            if ($parallel\_axes(e1, e2)$ or
            $colinear\_axes(e1, e2)$) then
                continue
            end if

            $p \leftarrow intersection(e1, e2)$

            if $p$ exists then
                $points.insert(p)$
            end if
        end for

        $points.insert(random\_p\_on\_axis(e1))$
    end for

    $points.insert(get\_TCP(poe))$
    return $points$
end function
```

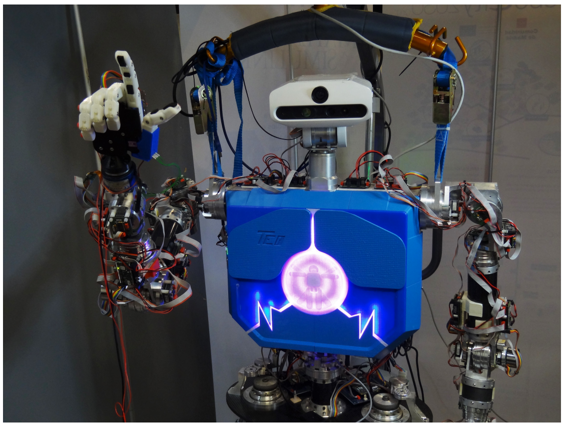

Figure 2: TEO humanoid robot

Having the screw theory solver (ST) as baseline, the selected algorithms are Levenberg-Marquardt (LMA, using damped SVD) and Newton-Raphson (NR, computing the pseudo-inverse Jacobian via truncated SVD). An epsilon (*eps*) parameter quantifies the precision of operations involving floating point numbers.

The results for three different target poses are presented in Table 1. The initial "guess" joint configuration corresponds to the left elbow rotated by 90 degrees pointing forward. The target cartesian poses (to be passed as input to the IK solvers) are derived via forward kinematics from the following approximate joint configurations, in degrees: 1. random pose far from the initial guess: $(-45, 45, 45, -75, 45, -90)$; 2. close to several joint

Snippet 4 Find a valid subproblem

```
function TRY_SOLVE($poe, n\_points$)
    $unknown\_poe\_terms \leftarrow analyze(poe)$
    $e1 \leftarrow get\_last\_term(poe)$
    $e2 \leftarrow get\_next\_to\_last\_term(poe)$

    if $unknown\_poe\_terms$ is 1 then

        if $n\_points$ is 1 then

            if ($e1.type$ is $rotation$ and
            not $lies\_on\_axis(e1, p1)$) then
                return PK1($e1, p1$)
            end if

            if $e1.type$ is $translation$ then
                return PG1($e1, p1$)
            end if

        end if

        if $n\_points$ is 2 then

            if ($e1.type$ is $rotation$ and
            not $lies\_on\_axis(e1, p1)$ and
            not $lies\_on\_axis(e1, p2)$) then
                return PK3($e1, p1, p2$)
            end if

            if $e1.type$ is $translation$ then
                return PG3($e1, p1, p2$)
            end if

        end if

    else if ($unknown\_poe\_terms$ is 2 and
    $n\_points$ is 1) then

        if ($e1.type$ is $rotation$ and
        $e2.type$ is $rotation$ and
        not $parallel\_axes(e1, e2)$ and
        $intersecting\_axes(e1, e2)$) then
            return PK2($e1, p1$)
        end if

        if ($e1.type$ is $translation$ and
        $e2.type$ is $translation$ and
        not $parallel\_axes(e1, e2)$) then
            return PG2($e1, e2, p1$)
        end if

        if ($e1.type$ is $rotation$ and
        $e2.type$ is $rotation$ and
        $parallel\_axes(e1, e2)$ and
        not $colinear\_axes(e1, e2)$) then
            return PG4($e1, e2, p1$)
        end if

    end if

    return $null$
end function
```

limits: $(-90, 20, -45, 90, -90, 45)$; 3. close to the initial guess: $(-30, 0, 0, -60, 0, 0)$. The mean elapsed time after 10^5 solve iterations on an Intel® Core™ i7-10700F CPU is obtained (compiled with GCC 9.3.0).

TABLE 1: Performance of IK algorithms.

Algorithm	Mean elapsed time (μs)		
	Pose 1	Pose 2	Pose 3
ST (baseline)	5.16	5.13	5.13
LMA ($eps : 10^{-5}$)	66.46	212.85	53.05
LMA ($eps : 10^{-3}$)	55.22	155.96	44.10
NR ($eps : 10^{-5}$)	74.29	–	31.46
NR ($eps : 10^{-3}$)	66.02	–	23.70

Results for the ST solver are consistent. Numeric solvers converge faster the closer the target pose is to the initial guess. In extreme situations, close to joint limits, the NR solver is unable to converge.

Accuracy has been not deemed critical for the selected set of parameters (the worst scenario given $eps = 10^{-3}$ rendered deviations well under one millimeter). A degradation has been observed when using $eps = 10^{-2}$, with no significant improvement in elapsed times.

7. CONCLUSIONS

In this paper, an inverse kinematics solver benefiting from screw theory mathematical fundamentals as a fast, efficient and effective closed-form method has been introduced. It has been proven on a real robot platform that it performs better than widely used numeric solvers by at least one order of magnitude.

In addition, it is easy to use as it only requires the description of a kinematic chain expressed in terms of a product of exponentials (or a Denavit-Hartenberg standard representation of homogeneous transformation matrices) to produce an internal, correctly ordered pipeline of steps iterating over each canonical subproblem. Other solvers would assume that said pipeline was previously determined and solved by a human so that only an input frame must be fed into the algorithm. Since automatic simplification and subproblem selection is performed on initialization, no penalty is imposed on runtime.

These features allow the presented solver to merge the best of two worlds: easy-to-generalize numeric solvers (although known limitations exist) and fast, efficient and accurate closed-form solvers.

Acknowledgement

The research leading to these results has received funding from: European project "Human Centric Algebraic Machine Learning" (ALMA), H2020-EIC-FETPROACT-2019; ROBOASSET, "Sistemas robóticos inteligentes de diagnóstico y rehabilitación de terapias de miembro superior", PID2020-113508RB-I00 funded by AGENCIA ESTATAL DE INVESTIGACION (AEI); RoboCity2030-DIH-CM, Madrid Robotics Digital Innovation Hub, S2018/NMT-4331, funded by "Programas de Actividades I+D en la Comunidad de Madrid" and cofunded by the European Social Funds (FSE) of the EU; the R&D&I project PLEC2021-007819 funded by MCIN/AEI/10.13039/501100011033 and by the European Union NextGenerationEU/PRTR; and cofunded by Structural Funds of the EU.

References

[1] Siciliano, B., Khatib, O., (2008) Handbook of robotics, Springer.

[2] Ball, R.S., (1900) The theory of screws, Cambridge University Press.

[3] Pardos-Gotor, J.M., (2018) Screw theory for robotics: a practical approach for modern robot kinematics, Amazon Fulfillment.

[4] Denavit, J., Hartenberg, R.S., (1955) A kinematic notation for lower-pair mechanisms based on matrices, Journal of Applied Mechanics.

[5] Brockett, R.W., (1984) Robotic manipulators and the product of exponentials formula, Lecture Notes in Control and Information Sciences, Mathematics.

[6] Paden, B., (1986) Kinematics and control robot manipulators. PhD thesis, Dept. of Electrical Engineering and Computer Sciences, Univ. of California, Berkeley.

[7] Kahan, W., (1983) Lectures on computational aspects of geometry, Dept. of Electrical Engineering and Computer Sciences, Univ. of California, Berkeley.

[8] Solà, Joan et al., (2021) A micro Lie theory for state estimation in robotics, arXiv:1812.01537.

[9] GitHub: Sophus, C++ implementation of Lie groups using Eigen, https://github.com/strasdat/Sophus. Accessed: 2022-07-18.

[10] GitHub: smooth, Lie theory for robotics, https://github.com/pettni/smooth. Accessed: 2022-07-18.

[11] Corke, Peter, Robotics Toolbox for MATLAB™, https://petercorke.com/toolboxes/robotics-toolbox/. Accessed: 2022-07-18.

[12] Sagar, K., Ramadoss, V. et al., (2020) STORM: Screw theory toolbox for robot manipulator and mechanisms, Int. Conf. on Intelligent Robots and Systems, Las Vegas, NV, USA.

[13] KDL Wiki, The Orocos Project, https://orocos.org/wiki/orocos/kdl-wiki.html. Accessed: 2022-04-07.

[14] GitHub: kinematics-dynamics, https://github.com/roboticslab-uc3m/kinematics-dynamics/. Accessed: 2022-04-07, latest commit hash: bcfc3c7.