Automated Symbolic-Numeric Methods for Efficient Kinematics Computation of Loops in Mechanical Systems

Stefan Klein, Andrés Kecskeméthy
Institute of Mechanics and Mechanisms, Graz University of Technology,
Kopernikusgasse 24, A-8010 Graz, Austria.
email: {klein,kecs}@mechanik.tu-graz.ac.at.

This paper describes an object-oriented implementation and extension of the algorithm described in [6] for finding and generating closed-form solutions in single-loop mechanisms featuring such closed-form solutions. In contrast to the previously mentioned approach, the present method relies purely on the result of geometric projections, without performing symbolical simplifications. This makes it possible to compute the closed-form solutions in a rather short time without the need of employing algebraic packages such as Mathematica or Maple. By this, the approach is suitable for integration in CAD systems, where a compromise between fast solution generation and fast solution computation is required. Moreover, the approach allows one to cycle through all solutions of a mechanism featuring explicitly solvable kinematics.

1 Introduction

Multibody systems consist of mechanical parts (bodies) interconnected by joints, such as revolute, prismatic, spherical, etc. [9] When the interconnection structure is open, i.e., the system has tree structure, the relative motions at the joints are independent and one can readily compute the absolute motion of each body starting from the ground and traversing the branches joint-by-joint and body-by-body.

When the system displays closed loops, however, the relative motions at the joints must fulfill closure conditions for each loop and hence are not independent [6]. In order to formulate these closure conditions, the loop is regarded as a sequence of n homogeneous matrices $A_i = \begin{bmatrix} \mathbf{R}_i & \underline{r}_i \\ 0 & 1 \end{bmatrix}$, $i = 1, \ldots, n$, describing the translation $\underline{r}_i$ and rotation $\mathbf{R}_i$ between coordinate frames $\mathcal{K}_{i-1}$ and $\mathcal{K}_i$ either within bodies or as input-output frames at the joints. The closure of the loop implicates that the first and last coordinate frames $\mathcal{K}_0$ and $\mathcal{K}_n$ must coincide, which can be stated as

$$A_1 A_2 \ldots A_n = I_4. \tag{1}$$

Here, matrices A_i contain six dependent joint variables $\beta_1, \ldots, \beta_6$ in the case of a spatial mechanism and three dependent joint variables $\beta_1, \ldots, \beta_3$ in the case of a planar or spherical mechanism. Six of the twelve scalar equations contained in Eq. (1) are dependent because of the orthogonality condition ${R_i}^T R_i = I_3$ of rotation matrices, and for planar or spherical mechanisms three of the remaining six equations are always fulfilled identically. As the closure conditions are coupled nonlinear equations, their resolution is non-trivial. Hence, in current commercial code for kinematics solution of mechanical systems, iterative methods such as Newton's method are employed [2], [1]. However, iterative solutions have two major drawbacks: (1) they can become inefficient when the system is complicated, and (2) because there are multiple solutions for the closure conditions, the convergence of the iterative method to a particular solution is fortuitous, and the solution can “jump” from one branch to another during simulation.

2 Loops rendering closed-form solutions

In contrast to iterative methods, kinematisists have worked for several decades with closed-form solutions, which exist in particular cases in which the architecture of the mechanical system and its geometric parameters fulfill certain conditions. Such systems form a negligible subset of “special cases” within the set of all possible mechanical systems, but for technical applications, they are the rule and not the exception. The problem with such closed-form solutions is that they require a vast experience and dexterity from the analyst, who has to find the “right” way of formulating the closure conditions and work out their solutions with pencil and paper.

In this paper, we propose a set of C++ [10] classes that can be employed for detecting special loop architectures and generating the corresponding closed-form solutions. The developed classes generalize the Mathematica [11] program SYMKIN for fully symbolical generation of closed-form solutions [8]. This method is briefly reviewed below for better reference, leaving out details that can be recovered from [6] or [8].

The basic idea behind this present method is to (1) reorder the multiplication sequence of matrix multiplications in Eq. (1), including carrying some of the matrices to the right-hand side, and (2) to apply geometric projections selected in such a way that the resulting scalar equations display an echelon form in terms of the unknowns from the outset. This is in contrast to other approaches to this goal, in which the closure conditions are regarded as a set of algebraic equations that are then manipulated with generic algebraic procedures such as Gröbner Bases. The projections are taken between two of the geometric elements point and plane, which are represented by the origin or a coordinate plane of a frame, respectively. Let $\underline{e}_i, i = 1,2,3$, denote unit vectors in direction of the coordinate axes, $\underline{r}$ the difference vector of two radius vectors each describing the position of the origin of a frame, and $\mathbf{R}$ the rotation matrix between two frames. With the homogeneous matrix $A = \left[\begin{smallmatrix} \mathbf{R} & \underline{r} \\ 0 & 1 \end{smallmatrix}\right]$, the regarded projections are:

$$\begin{array}{lll} \text{the squared distance between two origins:} & g_{PP}(A) & = ||\underline{r}||^2, \\ \text{the squared distance between an origin and a plane:} & g_{EP}(A; \underline{e}_i) & = \underline{e}_i \cdot \underline{r}, \\ \text{the cosine of the angle between two planes:} & g_{EE}(A; \underline{e}_i, \underline{e}_j) & = \underline{e}_i^T \mathbf{R}\, \underline{e}_j. \end{array} \tag{2}$$

The selection of proper projections is accomplished with the aid of an *invariance property matrix (IPM)* [4], which holds the isotropy groups of the connecting joints and bodies. By choosing the two “longest” subchains, each leaving a geometric element invariant, and generating a projection π between those two elements, an equation can be produced in which many of the unknowns are eliminated automatically. Here, “length” is measured as the number of dependent variables contained in the subchains. Let $\hat{\mathrm{A}}_A$ and $\hat{\mathrm{A}}_B$ denote the resulting homogeneous transformation matrices between the tip and the base of these subchains. Then, Eq. (1) can be partitioned as

$$\hat{\mathrm{A}}_B A_{II} \hat{\mathrm{A}}_A = {A_I}^{-1}. \tag{3}$$

Carrying out projection π on both sides of Eq. (3) yields

$$\pi(A_{II}) = \pi({A_I}^{-1}). \tag{4}$$

Hence, all individual transformations contained in $\hat{\mathrm{A}}_A$ and $\hat{\mathrm{A}}_B$ are eliminated from the outset, including the dependent variables contained therein. If in this process all but one of the unknowns β_i are eliminated, Eq. (4) can be employed to solve for this first unknown, and then the rest can be resolved in a sequential manner, which leads to equations displaying an echelon form:

$$\begin{aligned} f_1(\beta_{i_1}) &= 0 \\ f_2(\beta_{i_2};\beta_{i_1}) &= 0 \\ f_3(\beta_{i_3};\beta_{i_2};\beta_{i_1}) &= 0 \\ f_4(\beta_{i_4};\beta_{i_3};\beta_{i_2};\beta_{i_1}) &= 0 \\ f_5(\beta_{i_5};\beta_{i_4};\beta_{i_3};\beta_{i_2};\beta_{i_1}) &= 0 \\ f_6(\beta_{i_6};\beta_{i_5};\beta_{i_4};\beta_{i_3};\beta_{i_2};\beta_{i_1}) &= 0 \quad . \end{aligned} \tag{5}$$

The resulting equations each have the structure $A\,\cos\beta + B\,\sin\beta + C = 0$ for rotational unknowns and $A\,s^2 + B\,s + C = 0$ for translational unknowns [4]. They can be produced "symbolically" by recording the operations needed to compute the coefficients A, B and C, as explained below.

3 Online solution generation and computation

For simulation purposes, the symbolical formulas generated by the Mathematica [11] program SYMKIN need to be converted into a programming language code such as FORTRAN or C++ through an additional pass. Thus, the formula generation can take place off-line and it is not time-critical itself. In contrast to this, in online simulations the solution formulas need to be generated *and* executed immediately. Hence, a new approach needs to be derived that amalgamates these two stages in an efficient manner. To this purpose, numerical multibody code may be extended with features for symbolical formula generation procedures as the one described in Section **2**. In this paper, we chose the object-oriented multibody simulation package M◻BILE [5] as the platform for multibody simulation. The basic processing scheme is to generate the explicitly solvable formulas once per simulation run, and then evaluate these continuously for motion rendering. This processing scheme was implemented in a class `MoLoop` which shall be referred below.

The code `MoLoop` is split into two parts. The first part processes the topology of a loop and generates all the information necessary to solve it; this part is executed only once at the initialization of `MoLoop`. The second part makes use of this information and actually solves the loop; this part is implemented in the virtual member function `doMotion` containing motion-transmission code for all objects in M◻BILE. Both parts are described in more detail in the following subsections.

3.1 Topological processing

At initialization, `MoLoop` requires the following input:

(1) a list containing the joints and rigid links forming a closed loop, given as a MOBILE-object `MoMapChain`,

(2) a list of the dependent joint variables to be resolved, given as a MOBILE-object `MoVari-ableList`.

For list (1) revolute (R) and prismatic (P) joints are supported. The following steps are executed at initialization for the topological processing of the loop:

1. After processing the list (1), an invariance property matrix is set up, in which the invariance properties of each object in the loop are stored column-wise. The regarded invariance properties state whether the transformation matrix of the referred object leaves an origin ("$\underline{o}$") or one or more coordinate plane(s) ("Π_x", "Π_y", "Π_z") invariant. Tab. 1 summarizes the invariance properties of prismatic (P) and revolute (R) joints, where a "1" denotes invariant and a "0" denotes not invariant. The resulting invariance property matrix (IPM) is stored in an object of class `MoIPM`.

	P			R		
	x	y	z	x	y	z
Π_x	0	1	1	1	0	0
Π_y	1	0	1	0	1	0
Π_z	1	1	0	0	0	1
$\underline{o}$	0	0	0	1	1	1

Table 1: Lookup-table for invariance properties

`MoIPM` contains member functions for

– finding the two longest subchains $\hat{\mathrm{A}}_A$ and $\hat{\mathrm{A}}_B$ leaving a geometric element invariant and delivering the chain partitioning as stated in Eq. (3),

– delivering the two invariant geometric elements regarding to $\hat{\mathrm{A}}_A$ and $\hat{\mathrm{A}}_B$, and

– decomposing the matrix A_{II} of Eq. (4) into three sub-transformations A_ℓ, A_r and A_E, where A_E holds the actual unknown and A_ℓ, A_r render the left and right subchain embracing A_E. With this decomposition, Eq. (4) can be written as

$$\pi(A_\ell A_E A_r) = \pi({A_I}^{-1}). \qquad (6)$$

2. If the IPM contains rows totally filled with ones, these rows are removed, because this situation corresponds to a planar or spherical mechanism, where three of the six spatial closure conditions are always fulfilled identically.

3. In a class `MoESF`, all information required to compute the coefficients A, B, C of the explicit solution formulas is stored. This comprises

- the joint corresponding to A_E in Eq. (6),
- the matrices A_ℓ, A_r and ${A_I}^{-1}$ of Eq. (6), stored as lists of homogeneous matrices which refer to loop objects and multiply to A_ℓ, A_r and ${A_I}^{-1}$, respectively, and
- the two invariant geometric elements associated with the subchains $\hat{\mathrm{A}}_A$ and $\hat{\mathrm{A}}_B$.

This step is repeated, eventually in slight variations, until an explicit solution formula is generated for each dependent joint variable.

4. Finally, every joint and rigid link is associated with a homogeneous matrix $A_i = \begin{bmatrix} \mathbf{R}_i & \underline{r}_i \\ 0 & 1 \end{bmatrix}$ to represent the loop objects. This is realized by another class `MoSpatialTransformation` providing the translation $\underline{r}_i$ and rotation $\mathbf{R}_i$ of every joint type, using the actual value of the joint variable.

3.2 Motion transmission

The motion transmission function `doMotion` of `MoLoop` computes the actual pose of all the bodies within the corresponding loop object. This function is invoked once at the start of a motion simulation and every time the user changes a parameter or degree of freedom of the mechanism. It uses the list of `MoESF`s generated during the topological processing of the loop. The function `doMotion` performs the following steps for every `MoESF` in the list:

1. The actual coefficients A, B, C of the explicitly solvable constraint equation are computed in two steps:
   (1a) Evaluation of the actual matrices A_ℓ, A_r and ${A_I}^{-1}$ of Eq. (6),
   (1b) Application of the geometric projection stored in the corresponding ESF to both sides of Eq. (3). The resulting Eq. (4) delivers the desired coefficients for this `MoESF`.
2. With the actual coefficients, the resulting equation is solved for the unknown joint variable, applying an explicit solution formula [8]. In general, there are two different solutions, and both of them are computed and stored in this step. At some stages of the algorithm it is possible to generate two equations instead of one to solve for the unknown variable. In these cases the solution is unique.
3. After determining the unknown joint variable, the kinematics of the corresponding joint are computed. At this point, the solution branch for this `MoESF` is selected dependent on the configuration which is chosen for the whole loop.

After processing the list of `MoESF`s and thus computing all dependent variables of the loop, the relative motions of all joints are known and the absolute motion can be computed by traversing the loop joint-by-joint and body-by-body.

4 Examples

The first investigated example is a simple planar loop corresponding to the wheel suspension of a trailer. It is modeled as a planar mechanism consisting of four rigid bodies interconnected by three revolute joints (R) with parallel axes and one prismatic joint (P) with its joint axis being normal to those of the revolute joints. The kinematical structure is single-loop with one degree of freedom, for which the prismatic joint is chosen. The other three joint variables are unknown. The example shall be employed to describe the basic structure of the closed-form solutions generated in this setting. Fig. 1 illustrates the corresponding processing steps and the ensuing output.

First, a set of arrays for the objects (rigid links, joints, vectors, coordinate frames, etc.) needed to model the wheel suspension are created and initialized. For joints, the joint variable and the joint axis are passed as arguments. The type of joint variable, translational or rotational, classifies the joint to be either prismatic or revolute.

In this example, the first joint is the revolute joint `J[0]`; this joint connects the base coordinate frame `K[0]` as input and `K[1]` as output. The next object is the rigid link `L[0]` that connects the output frame of the previous revolute joint with the next coordinate frame `K[2]`, etc., until one reaches the prismatic joint `J[3]` that closes the loop by supplying as its output frame the base frame of the complete loop, i.e., `K[0]`. All revolute joint axes are aligned with the global x axis; the prismatic joint has its axis in direction of the global z axis.

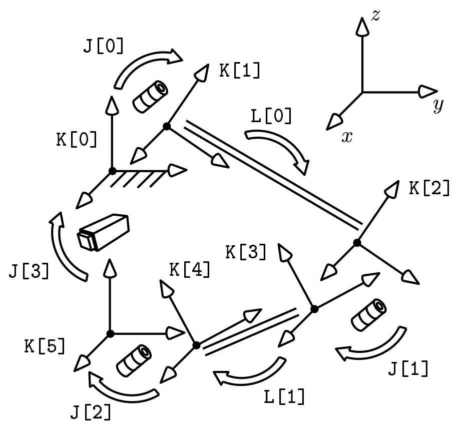
All loop objects are concatenated in a chain "`loopChain`" and passed to an instance "`loop`" of type `MoLoop`, together with a list "`allDepVars`" containing the three dependent joint variables `ang[0]`, `ang[1]` and `ang[2]`. The first frame `K[0]` of `loopChain` is regarded to be fixed to the ground.

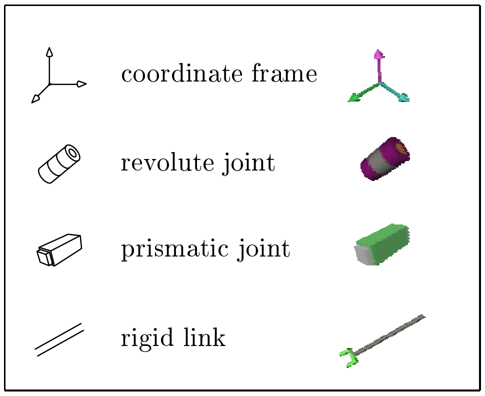
The independent variable of the loop is chosen as the linear translation `lin` contained in the prismatic joint `J[3]`. After setting a value for the position of this variable (suffix ".q"), the motion transmission function `doMotion` of `MoLoop` is invoked to determine the position of all bodies in the loop. Here, the user has to select the desired configuration out of two possible solutions. The resulting simulation windows for both solution branches are depicted in Fig. 1.

The resulting computational effort is compared with corresponding formulas generated by SYMKIN in Tab. 2. It is seen that the fully symbolic package SYMKIN produces much more efficient

Op.	SYMKIN	`MoLoop`	
$+,-$	69	236	(342 %)
$*$	78	307	(394 %)
$/$	8	0	
$\sqrt{\ }$	1	1	(100 %)
sin	3	1	(33 %)
cos	3	1	(33 %)
arctan	4	4	(100 %)

Table 2: Operation count for the wheel suspension: SYMKIN "vs." `MoLoop`





```
_void main() {

static MoFrame K[6];
static MoRigidLink L[2];
static MoElementaryJoint J[4];
static MoAngularVariable ang[3];
static MoLinearVariable lin;
static MoVector v[2];

static MoReal l1=2*0.73;
static MoReal l2y=2*0.31;
static MoReal l2z=2*0.06;
static MoReal l3y=2*0.425;
static MoReal l3z=2*0.065;

v[0]=MoVector(0,l1-l2y,l2z);
v[1]=MoVector(0,-l3y,l3z);

J[0].init(K[0],K[1],ang[0],xAxis);
L[0].init(K[1],K[2],v[0]);
J[1].init(K[2],K[3],ang[1],xAxis);
L[1].init(K[3],K[4],v[1]);
J[2].init(K[4],K[5],ang[2],xAxis);
J[3].init(K[5],K[0],lin,zAxis);

static MoMapChain loopChain;
loopChain << J[0] << L[0] << J[1]
          << L[1] << J[2] << J[3];
static MoVariableList allDepVars;
allDepVars << ang[0] << ang[1] << ang[2];
static MoLoop loop(loopChain, allDepVars);

lin.q=0.2;
loop.selectSolution(1);
loop.doMotion(DO_POSITION);
}
```

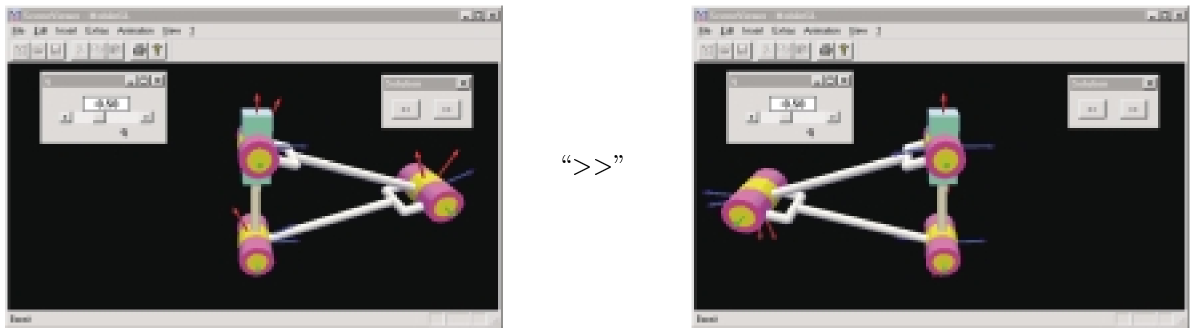


Figure 1: Model, source code and two configurations of the wheel suspension

code than the present method. Here, it must be noted that in SYMKIN the operation count involves also the global kinematics, while `MoLoop` only regards the determination of the dependent variables, so that the performance boost of SYMKIN is even more significant. On the other hand, `MoLoop` has the advantage that it produces very rapidly the explicit solution formulas sought for, and hence is suitable for online applications. Moreover, the present code is still more efficient than current numerical code. Tab. 3 displays the computation time needed for 300,000 loop solution computations, measured in seconds, for the present method and the “half-analytic” solution described in [5], as well as a purely iterative approach using the numerical routine “hybrj” of the

NAG library with error tolerance 10^{-8}.

Method	sec.
`MoLoop`	13
half-analytic	20.5
iterative	47

Table 3: Performance comparisons for the wheel suspension

As a more involved example, the kinematics of an elbow manipulator were computed by `MoLoop`. The manipulator consists of six revolute joints involving six dependent rotational joint variables and three rigid links. Its end-effector can be controlled by the independent joint variables of three prismatic and three revolute joints providing the six degrees of freedom of the end-effector as a combination of translational motion and BRYANT-angles. The eight possible configurations of the manipulator belonging to one end-effector state are reproduced in Fig. 2.

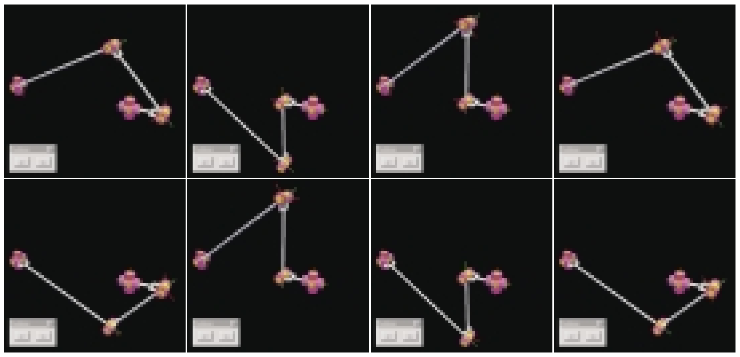


Figure 2: The eight configurations of the elbow manipulator

Conclusions

The present semi-symbolical method makes it possible to obtain closed-form solutions for single-loop mechanisms online, which is an advantage as compared to fully symbolical approaches, where solution generation may take minutes or hours. At the same time, the established solutions, while involving more computational effort than the fully symbolic approach, render much more efficient code at solution time than iterative methods. This makes the method suitable for motion animation in CAD systems, while at the same time allowing the user to switch between configurations.

Acknowledgments

Funding of the present work by the European Community as a Marie-Curie Fellowship under Contract Number ERBFMBICT972704 is gratefully acknowledged.

References

[1] J. E. Dennis and R. B. Schnabel. *Numerical Methods For Unconstrained Optimization And Nonlinear Equations.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1983.

[2] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization.* Academic Press, London New York Toronto Sidney San Francisco, 1981.

[3] A. Kecskeméthy. *Objektorientierte Modellierung der Dynamik von Mehrkörpersystemen mit Hilfe von Übertragungselementen.* Fortschrittberichte VDI, Reihe 20 Nr. 88. VDI-Verlag, Düsseldorf, 1993.

[4] A. Kecskeméthy. Kinematics of robots and mechanisms. In C. Melchiorri and A. Tornambè, editors, *International Summer School on Modelling and Control of Mechanisms and Robots*, pages 39 – 79. World Scientific Publishing Co. Pte. Ltd., Singapure, 1996.

[5] A. Kecskeméthy. *MOBILE User's Guide Version 1.3.* Institute for Mechanics and Mechanisms, TU Graz, 1999.

[6] A. Kecskeméthy and M. Hiller. Automatic closed-form kinematics-solutions for recursive single-loop chains. In *Flexible Mechanisms, Dynamics, and Analysis, Proc. of the 22nd Biennal ASME-Mechanisms Conference, Scottsdale (USA)*, pages 387–393, 1992.

[7] G. A. Kramer. *Solving Geometric Constraint Systems: A Case Study in Kinematics.* The-Massachusetts-Institute-Of-Technology-Press, Cambridge, Massachusetts, 1992.

[8] T. D. Krupp. *Symbolische Gleichungen für Mehrkörpersysteme mit kinematischen Schleifen.* Berichte aus der Softwaretechnik. Shaker Verlag, Aachen, 1999.

[9] W.O. Schiehlen. *Technische Dynamik.* Teubner, Stuttgart, 1986.

[10] Bjarne Stroustrup. *The C++ Programming Language, second edition.* Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Reading, MA, 1991.

[11] Stephen Wolfram. *The Mathematica Book.* Wolfram Media/Cambridge University Press, third edition, 1996.