Code Generation of Algebraic Quantities for Robot Controllers

Marco Frigerio, Jonas Buchli and Darwin G. Caldwell

Department of Advanced Robotics, Istituto Italiano di Tecnologia (IIT), via Morego, 30, 16163 Genova
{marco.frigerio, jonas.buchli, darwin.caldwell} @iit.it

***Abstract*— Controllers for articulated robots such as an arm or a humanoid commonly need to continuously calculate complex algebraic quantities, such as the joint space inertia matrix or Jacobians. An effective and fast implementation of the calculation of these quantities is crucial to achieve complex, yet robust controllers and thus enable sophisticated behaviors in robots. Although the nature of these algebraic quantities is very well known in robotics, they do not lend themselves easily to manual implementation, because of ambiguities and the complexity in their development and use. We propose an approach that addresses this issue by relying on automatic code generation, thus relieving the user from hand crafted development. Our approach also addresses efficiency and speed, in order to satisfy the strict requirements of real time robot controllers, yet it is easy to use. We show the effectiveness of our method by means of some preliminary comparisons.**

I. Introduction

This paper describes a software tool to support the development of simulators as well as fast model–based controllers for articulated robots, such as operational space controllers and impedance controllers [1]. All these applications make extensive use of algebraic expressions such as coordinate transforms, Jacobians, range and null space projectors, among the others [2].
Despite an established theoretical understanding of these objects in robotics, sound and efficient implementations are far from being trivial, yet they are critical for successful simulations and control and therefore for the safety of the robot itself. Writing software for these components is error prone and time consuming, also because of the lack of established, general software *models* which address these concepts, and thus the lack of *reusable* implementations.
Moreover these algebraic quantities typically suffer from ambiguities in their definition or in their usage (cf. Section III-C), so that developing them by hand is even more tricky. On the other hand, attributes that might resolve these ambiguities are usually not explicit in software packages, limiting their flexibility and reusability. The development becomes even more challenging when real time constraints and limited hardware resources demand fast and efficient code.

In short, our method addresses these issues by automating the implementation of these algebraic expressions, with the aim of improving the development process and thus the quality and the robustness of the software itself. It exhibits several desirable features which we can briefly summarize here, that will be further developed in the rest of the paper:

- Ease of use: the user is required to deal only with high level information – as for instance *which* Jacobians are of interest for a given robot – leaving implementation details to the computer. Ideally this approach should be used whenever possible, especially when dealing with complex systems like robots.
- Robustness: an automated code generation process is repeatable and cannot introduce occasional mistakes that would occur with manual development. Of course the process itself needs to be validated, but once established it allows very fast development and rapid prototyping.
- Flexibility: simple yet general models driving our implementation limit restrictions or anyway enable extensibility. For example it is possible to generate the transforms between any pair of possible frames attached to a kinematic tree, following the user's needs.
- Efficiency: despite the points above, the generated implementations can be optimized to address the speed and efficiency constraints of real time systems, making our approach suitable for use in real robot controllers.

Our method is based on Domain Specific Languages (DSLs), special purpose languages built on top of domain models, with a custom syntax [3]. The user can adopt such a language to provide high level input information and then have the computer perform an automatic transformation, for instance into executable code.
This work is tightly related to our previous one about a DSL for the specification of kinematic trees [4], which focuses on rigid body dynamics algorithms to show the benefits of model-based code generation in robotics. Our contribution with the present work goes in the same direction: another simple DSL independent of the first one, which provides the infrastructure to generate implementation of any coordinate transformation. Used in conjunction with the kinematics DSL, it allows to enrich the generation process tailored for a given robot, and specifically to obtain arbitrary transforms and geometric Jacobians. Thanks to off-line optimization, the code can also be very efficient and thus appropriate for computation intensive algorithms and hard real-time requirements (e.g. control loops in embedded machines).
As an additional contribution, we also added to the code generator of the kinematic DSL new algorithms, the composite-rigid-body algorithm to compute the joint space inertia matrix M, and the $L^T L$ factorization for the same matrix, both as detailed by Featherstone [5].

Altogether these components provide a robust support to start the development of model–based controllers such as operational space controllers.

The DSLs allow in principle to generate code in any language, but in this work we focus on C++ (we use the linear algebra library Eigen [6]). We use the Xtext workbench for Eclipse [7] and the Xtend2 language [8] to develop the languages; these tools provide the generic infrastructure to actually transform models into code (i.e. text files), which is not the focus of this paper. Parsers and generators developed with these technologies can be deployed as standalone Java packages. In addition, we rely on the Maxima symbolic computation engine [9], for doing off-line optimization and therefore generate efficient code (see Section III-E). Our purpose is to provide reusable tools constructed with open source technologies; indeed, a user who would like to use our system would only need a Java runtime environment and Maxima (which can be built for many systems).

The rest of the paper is organized as follows: Section II discusses some related work, while Section III explains the rationale and the assumptions behind our software, whose main purpose is to generate code. Section IV describes some of the achieved results and finally Section V discusses about future development of our work.

II. Related work

Since the introduction of the operational space concept by Khatib [10] its use has been extensively discussed in the robotics control literature (e.g. [11], [12], [13], [14], [15]). However, almost all these works address the theoretical aspect of the controller, rather than explicitly focusing on the software required to implement such approaches. Maybe the most notable exception to this is the Stanford Whole-Body Control (`S-wbc`) open source project [16], initially released in 2009 as a result of the efforts of the Stanford Robotics Lab to bring the operational space formulation into a software for the community.

The `S-wbc` comprises a library for joint-space kinematics and dynamics, on top of which additional components provide the abstractions required by operational space control. Specific effort has been spent on the configurability of the framework, so for instance classes can expose their own parameters via a reflection mechanism, and these can be changed at runtime.

Even though our approach shares with the `S-wbc` some design principles and goals such as flexibility and robustness, we aim at software components that are even more general and reusable, since they have a narrower domain, and not at a full-featured framework. We are also interested in efficiency and speed of execution, for scenarios with hard real time constraints. We aim at having our software be used as a building block of other applications, and the `S-wbc` itself might be an example of these.

Generation of code and equations is a feature available in some commercial programs. SD/FAST [17] is a feature–rich simulator of mechanical systems that produces C or Fortran implementation of the corresponding equations of motions. Robotran [18] targets multi-body systems and can output symbolic equations for kinematics and dynamics for MATLAB and SIMULINK. Similar considerations as for the `S-wbc` apply here for the comparison with our work.

We focus on a specific functionality, basing the implementation on explicit models – therefore improving documentation, usability and extensibility – and adopting only open source technologies.

III. Models and DSLs

A. Workflow

The diagram in Figure 1 gives an overview of the logical workflow we use to turn the high level information about the robot kinematics into working implementations of transforms and Jacobians. The starting point is the robot kinematics,

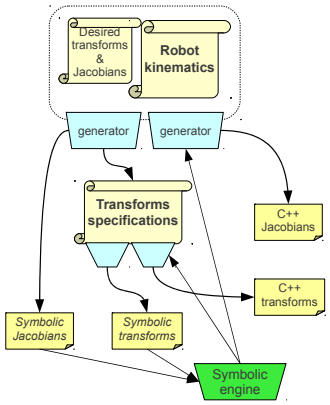

Fig. 1. A conceptual representation of the workflow for the transformation of models into executable code. The scroll-like blocks are instance documents of some DSL, the trapezoidal ones perform computations and the sheet-like ones are resulting code blocks. The curved arrows represent code generation, while the straight ones represent input/output data flows.

briefly described in Section III-B, together with a user specified list of desired transforms and Jacobians.

Instead of directly implementing the logic to generate the transforms from the robot kinematics, we designed a simple and general DSL for the abstract specification of any transform, which in turn has its own generator (see Section III-C). As shown in the workflow, the kinematic information is transformed into such an abstract, language independent description of the matrices.

This language is completely general and not constrained to robotics, and can be used as standalone tool; in our workflow we happen to *generate* a document compliant with the language, but this is just a special use case.

Having a dedicated language enforces the separation of concerns by making the model underlying the coordinates transforms explicit. It allows to have simpler and more reusable components; in this way it was possible to implement the generation of different kind of transforms (for homogeneous vectors and spatial vectors [5]) sharing the same, common abstract description.

One of the end products of the generation is the code for the symbolic computation engine, which can be used in turn as an input for subsequent steps of the workflow. Such code

can be interpreted and the result exploited to generate more efficient code for some other language. See Section III-E.

B. Kinematic trees

Information about the kinematic structure of a robot is the basis to generate the building blocks for its controllers. The robot model is encoded in a document of another DSL, which is detailed in our previous work [4].
Such a robot model specifies how links are connected via prismatic or revolute joints, and where the joints are located. Reference frames are attached to links and joints according to a convention (see also chapter 4 of [5]), but any number of additional, custom frames can be associated to any link of the structure. All these frames have a name and they are used to uniquely identify transforms and Jacobians, as detailed in the next sections.

C. The coordinate transforms DSL

A coordinate transformation matrix maps a coordinate vector to another coordinate vector, representing the same physical entity (e.g. a point) in a different reference frame:

$$p_B = {}_BX_A \cdot p_A$$

where A and B are two frames, p is the coordinate vector; more on the notation in the following paragraphs.
The key point behind a model for this kind of objects is to explicitly expose the information that is usually implicit and therefore lead to ambiguities and potential mistakes. The most important is the *direction* of the encoded transform: as an example, assume that frame B is rotated by an angle α about the z axis, with respect to A; this input *information* cannot be misunderstood (assuming right handed frame and right hand rule for the angle sign), but the expression for the rotation matrix $R_z(\alpha)$ does not tell whether it transforms vectors from B to A or vice versa, thus is ambiguous. For instance, the code in the MATLAB robotics toolbox [19] uses one convention while the 6D transforms introduced by Featherstone in [5] use the other one.

For the sake of simplicity we will focus only on the direction property, assuming to deal always with right handed coordinate frames and left–multiplication (i.e. the matrix always left–multiply a column vector of coordinates), these being other possible ambiguities in the usage of the transforms. We believe that this assumption is not a big limitation since different choices are quite uncommon; moreover, by additional attributes in the model we could possibly deal also with such uncommon cases.

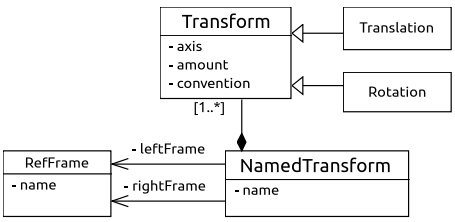

Fig. 2. The model for the coordinate transformations at the basis of the DSL

Figure 2 shows an UML class diagram representing the model we designed to provide a foundation for the DSL and its code generator. It comprises the following elements:

- `Transform`: this represents an atomic transformation matrix, that is either a rotation or a translation about or along one of the six Cartesian axes, by a certain amount (angular or linear displacement). The `convention` property is a two-valued field that tells whether the matrix is supposed to (pre)multiply a vector expressed in the rotated frame or in the base one. Together with the first attributes, this uniquely determine the matrix. We can think of these objects as generic, building blocks for composite transforms.
- `RefFrame`: a simple named object which identifies uniquely a certain reference frame (e.g. on a robot).
- `NamedTransform`: a transform between two specific frames, with a unique name, constituted by an arbitrary number of simple transforms. The two properties `leftFrame` and `rightFrame` explicitly identify the role of the matrix: it takes vectors expressed in `rightFrame` (the vector is on the right of the matrix, as in ${}_BX_A \cdot p_A$) and gives back vectors of `leftFrame`. This is usually only represented in the algebraic notation but not in code; as seen before, we will use the formalism ${}_{leftFrame}X_{rightFrame}$. Note that this information disambiguates also the issue of right or left multiplication.

```
Model Leg
Frames {
    hip, upperLeg, lowerLeg, Foot
}
TransformedFramePos = right

     {hip}_X_{upperLeg} = Rz(-PI/2.0) Rz(q_HFE)
{upperLeg}_X_{lowerLeg} = Tx(0.35) Rx(PI) Rz(q_KFE)
{lowerLeg}_X_{upperLeg} = Rz(-q_KFE) Tx(-0.35) Rx(-PI)
     {hip}_X_{Foot}     = Rz(-PI/2.0) Rz(q_HFE) Tx(0.35)
                          Rx(PI) Rz(q_KFE) Tx(0.33) [XFoot]
```

Fig. 3. A sample document of the coordinate transforms DSL, for an hypothetical simple robotic leg. `q_HFE` and `q_KFE` could be labels for Flexion-Extension joints, respectively in the hip and the knee.

Figure 3 shows a simple example of a document of our DSL, which could be the specification of the transforms of some imaginary three-link robotic leg. It contains a global name, a declaration of the frames referenced afterwards and then a list of the user–defined transformations. Tokens in the form `Rx()`, `Ty()`, ..., are keywords of the language and represent the atomic transformations; for simplicity, the `convention` attribute of each of them is specified only once globally (with the `TransformedFramePos` keyword). The transformations that are meaningful for the user must be specified with a syntax that clearly identifies the left and right frame attributes, but a custom, optional name can be specified at the end of the line in square brackets. The language supports numerical constants as well as symbolic identifiers, including the keyword `PI` that stands for π; this is useful to model the dependency of the transform from some variable, which in our case will typically be the status of the joints of the robot.

D. Geometric Jacobians

Jacobians map joint space velocities to Cartesian velocities of some reference frame of interest located in the robot, according to the well known formula

$$\dot{\mathbf{x}} = J(\mathbf{q})\dot{\mathbf{q}}$$

As for a coordinate transform, also the Jacobian is fully identified by two reference frames, which we call the "target frame" and the "origin frame". They are respectively the frame of the body whose velocity (or force) is of interest and the frame in which the velocity (or force) has to be expressed. In the robotics literature it is common to find expressions like "end-effector Jacobian" or "constraint Jacobian", where these expressions are really just shortcuts to identify the frame of interest, without mentioning the origin one since it is assumed to be known from the context (an often coincides with the robot base frame). However, to have software deal automatically with these objects in a general way, this information cannot be left implicit.
The addition of Jacobians in the domain model is quite straightforward, and it is shown in Fig. 4. The drawing also

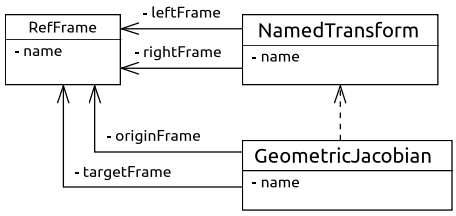

Fig. 4. A view of the domain model including the class for geometric Jacobians.

shows (by means of a dashed arrow) the dependency with the `NamedTransform`, since any geometric Jacobian can be fully computed on the basis of direct kinematics [2].

We adopted this model to represent Jacobians in our software and to generate code that implements them. However we added this logic in the same package of the robot kinematics, without affecting the coordinate transforms DSL, as can be seen from the workflow in Figure 1.

E. The role of the symbolic computation engine

In our workflow we use a symbolic computation engine (Maxima), purely to achieve higher efficiency in the code generated for other languages, in our case C++. Without symbolic calculus, one could still generate executable implementations, simply by translating the atomic transforms into matrices in the target language, and the composite transforms into products of such matrices; this is exactly what we do to generate code for Maxima itself. The process is even more convenient with a small library in the target language already implementing the six objects `Rx()`, ..., `Tz()`. With this approach, once a code generator is developed, one still gets relieved from developing manually the various transforms of the robots.

On the other hand, when efficiency is of concern, as in our case, the symbolic engine can be exploited to develop the multiplications between the atomic transforms and get the resulting matrix, possibly simplified to a compact form. Therefore the engine must be able to perform linear algebra and possibly some simplifications of trigonometric functions. Then, another generator can take advantage of a priori knowledge about the structure of the resulting matrix: which elements are constant, which ones are equal to some other, which and how many unique trigonometric functions the matrix depends on, so that the generated code does not compute sines and cosines (typically the most expensive functions) multiple times.

IV. Results

In this section we will present some of the results we achieved so far, to demonstrate the feasibility and the convenience of the approach together with the performance of the generated code.

We ran some regular tests to check the correctness of the generated implementations. We mainly performed some comparisons with the SL software package [20], which has been in development for more than fifteen years and it is used in several research labs for simulations and hard real time model–based control of real robots.
As an additional validation, we partially rewrote a controller for our quadruped robot HyQ [21] using our approach. This program – which is detailed in another paper from our research group [22] – controls the impedance of a single leg of the robot, and exploits coordinate transforms, the Jacobian ${}_{base}J_{foot}$, and so on. We replaced the original implementation of some of these quantities and we managed to obtain the same behavior from the robot. More importantly, once the task was developed, it was a matter of minutes to generate the same expressions for a different leg and have the software control its impedance instead.

A. Joint space inertia matrix

The joint space inertia matrix M is a symmetric, positive-definite matrix that maps forces and accelerations at the joints of a multi-body system [5]. This matrix is used in rigid body dynamics algorithms but also appears in the operational space formulation. It captures the whole system dynamics in the absence of external forces and body velocities:

$$\boldsymbol{\tau} = M(\mathbf{q})\ddot{\mathbf{q}}$$

As mentioned in the introduction, we can generate the composite-rigid-body algorithm to efficiently compute M. This algorithm is known to be the most efficient for this job, especially because it exploits the sparsity of the matrix induced by the branching of the kinematic tree [5]. Moreover, the algorithm requires almost all the 6D-force-vector transforms in the form ${}_{parent}X_{child}$ and the motion-vector transforms ${}_{child}X_{parent}$, thus its performance depends also on them.

The computation of M is therefore a good test case for our approach, both to check the numerical correctness and especially for the performance of execution. As far as the latter point is concerned, we performed some comparisons

on the execution times with the S-wbc software, and the results are summarized in Figure 5. Tests were executed with

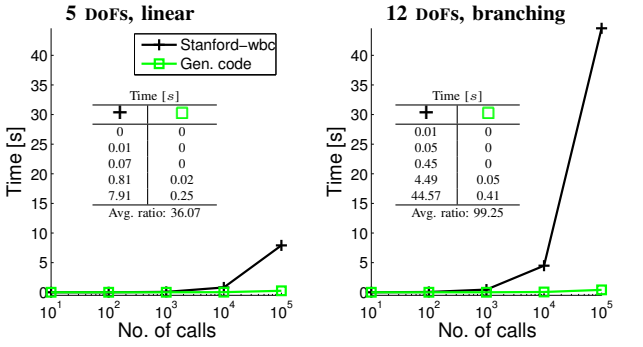

Fig. 5. Performance comparison with the S-wbc for the computation of the joint-space inertia matrix M. The x axis represent the number of calls, the y axis the total execution time (in seconds – the exact values are shown in the table in the center). The graph on the left refers to a 5 DoF linear robot, the one on the right to a 12 DoF quadruped robot (tests executed on a Intel(R) Core(TM)2 Duo CPU, P8700 @ 2.53GHz).

two different robot models:

1) a 5 DoF fictitious robot, composed of a linear kinematic chain with three revolute joints and two prismatic joints
2) a 12 DoF robot (our quadruped HyQ [21]), which has four 3 DoF legs and therefore exhibits a branched kinematic structure

We measured the cumulative execution time of multiple calls to update $M(\mathbf{q})$, by means of the standard library function `std::clock()`. For the S-wbc, for instance, this simply means to measure the single call `computeMassInertia()` of the joint space model class. Even tough we do not claim these to be definite and exhaustive comparisons, the plots clearly show a significant faster execution of our implementation, by a factor of about 35 for the first robot. For the one with a branched structure the gain raises up to 100 most probably because in the S-wbc the sparsity of M cannot be exploited.

In addition, our code is compatible with runtime environments enabling hard real time computations as for instance a Linux system with Xenomai [23]. Together with the fast execution speed this means we can employ the generated code in actual controllers of real robots, for example a hydraulic one [21].

B. Null space projector

We show in this section an additional comparison on performance, this time for the computation of a null space projector N. This is another common ingredient of the operational space control used to compute velocities and torques at the joints that do not result in motion at the end effector. One possible definition is the following:

$$N^T = I - J^T(JM^{-1}J^T)^{-1}JM^{-1}$$

where all the terms are function of the joint status $\mathbf{q}$. As you can see, it requires a Jacobian (e.g. the base to end-effector Jacobian) and the inverse of M, so we are basically comparing the computation of both these terms in the S-wbc and in our software.

We performed the test only in the first, 5 DoF robot without branches, not to include the effect of sparsity, which we have already shown to affect significantly the gap between the S-wbc and the generated code. We selected an hypothetical end-effector at the tip of the last link, and used the corresponding Jacobian. Results are illustrated in Figure 6. Also

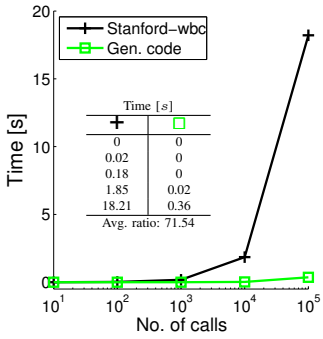

Fig. 6. Performance comparison with the S-wbc for the computation of the inverse of the joint space inertia matrix and the end-effector Jacobian, for a 5 DoF robot. The x axis represent the number of calls, the y axis the total execution time (in seconds – see Fig. 5).

for this case the plot indicates a better performance for the code generated with our approach. The average ratio of the execution time is even higher than before, as one might have expected because we now have two terms computed possibly in a more efficient way. We get M^{-1} by means of the L^TL factorization, performed by generated code implementing the algorithm described again in [5]. The remaining operations $L^{-1}L^{-T} = M^{-1}$ are done with generated code as well, tailored for triangular matrices and exploiting the robot specific sparsity (if any). These operations obviously add a cost to the computation of M, which however seems less than the increase of execution time observed in the S-wbc to compute M^{-1} with respect to M; for these tasks, the current implementation of the S-wbc uses a call respectively to the forward dynamics and the inverse dynamics routines.

The computation of the Jacobian is also relevant in this respect: our implementation is highly optimized since it has been generated in advance for a specific, known point of the kinematic tree, and therefore outperforms a regular, generic implementation that relies only on computations at run-time. It is true that with our approach we cannot have the Jacobians for any arbitrary point determined *dynamically* (i.e. at runtime), but on the other hand it is very often the case that the points of interests on the robot are known or can be estimated in advance, as for instance the end-effectors. However, the framework already provides a robust way to address also these issues: as an example, one might generate the Jacobians for every link of the robot, and pick one of them at runtime once a specific point on a certain link is identified (e.g. a contact point); the velocity of the point is then obtained by a simple motion transform applied to the velocity of the link.

V. Conclusions and future works

In this paper we have described computer tools to support the development of effective controllers for real robots, focusing on common linear algebra expressions such as coordinate transformations or null space projectors. The approach is based on simple high level models of the information and on the corresponding Domain Specific Languages, that provide the infrastructure to generate efficient code out of such models. A general aim of our efforts is to relieve roboticists from spending time on non–problems, which are issues well understood in theory and therefore often of scarce scientific interest, but yet critical for robotics applications. Effective software solutions for these (non–)problems can still be demanding, and contribute significantly to the costs of development and especially maintenance of the system.

Our approach has proven to be effective, in that it exhibits a diversity of desirable features for robotics software: it is easy to use because the user deal only with high level information; the use of shared models among the components improves the observability of the process and documentation in general; it is robust and limits human mistakes because it is based on automatic code generation; yet it is effective for real time robot control, because the generated implementation is fast and efficient. In general we are convinced that this software could be adopted as a building block of composite systems, such as real-time capable operational space controllers.

Several improvements and developments of our work are possible. As an example, it is important to extend our software models with the explicit notion of *task* and *task space*. In this way the software could reason about properties such as the dimension of a task space, for example to generate a Jacobian with the significant rows only, to automatically detect task-specific redundancy, or to identify the subset of joints involved in a specific task.

As far as efficiency is concerned, the use of sparse matrices might make generic computations (sums and products in user code) faster. In principle, given a known matrix, the generator could choose whether to use or not a sparse matrix implementation in the target language when generating code. However it is quite difficult to devise such criteria in general, also because the benefits highly depend on the specific implementation.

Another development concerns the optimization of the number of computations: currently each generated matrix updates itself (given a new joint state vector $\mathbf{q}$) computing once all the terms it is a function of, even if multiple elements depend on them. This kind of optimization is not happening at the global level, among different matrices that might be function of some shared terms (e.g. terms like ${}_{base}X_{link1}$, ${}_{base}X_{link2}$, ${}_{base}X_{link3}$, etc.). However our software system is already designed with this idea in mind, and realizing such improvement would be a matter of tuning our implementation.

Acknowledgements

This research has been funded by the Fondazione Istituto Italiano di Tecnologia.

The authors would like to thank the developers and the maintainers of Xtext, Xtend, Maxima, Eigen.

Appendix

A preliminary version of our software is available for download at: `http://www.iit.it/en/article/10-advanced-robotics/1253-robotics-code-generator.html`

This C++ snippet gives an example of the usage of the generated code:

```
HyQ::JointState q;  // ... to be filled
HyQ::JacobianT<3>&        J = HyQ::jacs::trunk_J_LFfoot; // just an alias
HyQ::JSpaceInertiaMatrix& M = HyQ::jspaceM;              // just an alias
M(q);  J(q);  M.getL();  M.getLinv();  Minv=M.getInv();  // actual computations
I - J.transpose() * (J * Minv * J.transpose()).inverse() * J*Minv; // computes 'N'
```

References

[1] N. Hogan, "Impedance control: An approach to manipulation: Part II – Implementation," *ASME, Transactions, Journal of Dynamic Systems, Measurement, and Control*, vol. 107, pp. 8–16, 1985.

[2] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics. Modelling, Planning and Control*, M. J. Grimble and M. A. Johnson, Eds. Springer, 2009.

[3] M. Fowler, *Domain-Specific Languages*. Addison-Wesley, 2010.

[4] M. Frigerio, J. Buchli, and D. G. Caldwell, "A domain specific language for kinematic models and fast implementations of robot dynamics algorithms," in *2nd International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob)*, 2011.

[5] R. Featherstone, *Rigid Body Dynamics Algorithms*. Springer, 2008.

[6] G. Guennebaud, B. Jacob, *et al.* The eigen library v3. [Online]. Available: http://eigen.tuxfamily.org

[7] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in *ACM international conference companion on Object oriented programming systems languages and applications companion*, ser. SPLASH '10. New York, NY, USA: ACM, 2010, pp. 307–309.

[8] The Xtend language. [Online]. Available: http://www.xtend-lang.org/

[9] Maxima. (2011) Maxima, a computer algebra system. version 5.25.1. [Online]. Available: http://maxima.sourceforge.net/

[10] O. Khatib, "A unified approach for motion and force control of robot manipulators: The operational space formulation," *IEEE Journal on Robotics and Automation*, vol. 3, no. 1, pp. 43–53, 1987.

[11] B. Siciliano and O. Khatib, Eds., *Springer Handbook of Robotics*. Springer, 2008.

[12] J. Park, "Control strategies for robots in contact," Ph.D. dissertation, Stanford University, March 2006.

[13] L. Sentis, "Synthesis and control of whole-body behaviors in humanoid systems," Ph.D. dissertation, Stanford University, July 2007.

[14] J. Nakanishi, R. Cory, M. Mistry, J. Peters, and S. Schaal, "Operational space control: A theoretical and empirical comparison," *The International Journal of Robotics Research*, vol. 27, no. 6, pp. 737–757, 2008.

[15] L. Righetti, M. Mistry, J. Buchli, and S. Schaal, "Inverse dynamics control of floating-base robots with external contraints: an unified view," in *IEEE international conference on robotics and automation (ICRA)*, 2011.

[16] R. Philippsen, L. Sentis, and O. Khatib, "An open source extensible software package to create whole-body compliant skills in personal mobile manipulators," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, September 2011, pp. 1036 –1041.

[17] M. Sherman and D. Rosenthal. SD/FAST. [Online]. Available: http://www.sdfast.com/

[18] Robotran. [Online]. Available: http://www.robotran.be/

[19] P. Corke, "A robotics toolbox for Matlab," *Robotics Automation Magazine, IEEE*, vol. 3, no. 1, pp. 24 –32, mar 1996.

[20] S. Schaal, "The SL simulation and real-time control software package," CLMC lab, University of Southern California, Tech. Rep., 2009.

[21] C. Semini, N. G. Tsagarakis, E. Guglielmino, M. Focchi, F. Cannella, and D. G. Caldwell, "Design of HyQ – a hydraulically and electrically actuated quadruped robot," *Proc. IMechE Part I: J. Systems and Control Engineering*, vol. 225, 2011.

[22] M. Focchi, T. Boaventura, C. Semini, M. Frigerio, J. Buchli, and D. G. Caldwell, "Torque-control based compliant actuation of a quadruped robot," in *12th IEEE International Workshop on Advanced Motion Control (AMC)*, 2012.

[23] P. Gerum, "Xenomai – implementing a RTOS emulation framework on gnu/linux," April 2004.