Preprint October 26, 2016
Accepted for publication at the 2016 IEEE International Conference
on Simulation, Modelling, and Programming for Autonomous Robots,
San Francisco

Fast Derivatives of Rigid Body Dynamics for Control, Optimization and Estimation

Michael Neunert[1], Markus Giftthaler[1], Marco Frigerio[2], Claudio Semini[2] and Jonas Buchli[1]

***Abstract*— Many algorithms for control, optimization and estimation in robotics depend on derivatives of the underlying system dynamics, e.g. to compute linearizations or gradient directions. However, we show that when dealing with Rigid Body Dynamics, these derivatives are difficult to derive analytically and to implement efficiently. To overcome this issue, we extend the modelling tool "RobCoGen" to be compatible with Auto Differentiation. Additionally, we propose how to automatically obtain the derivatives and generate highly efficient source code. Finally, we demonstrate an example application using Trajectory Optimization to underline the potential gain of using these derivatives in a control setting.**

I. Introduction

Most robotic systems consist of multiple rigid links connected via joints. This includes entire systems such as robotic arms, legged robots or exoskeletons, but also components like robotic hands or integrated actuators. We can mathematically model these systems using Rigid Body Dynamics. The latter typically results in non-linear differential equations for which computing a closed-form solution is intractable. Therefore, many state-of-the-art approaches in control, estimation, optimization and planning rely on iterative, gradient-based algorithms which compute linear approximations of the given system equations. These algorithms include Optimal Controllers such as Linear Quadratic Regulators (LQR), optimal estimation approaches such as Kalman Filters and Batch Estimation, as well as parametric design optimization and Trajectory Optimization (e.g. Direct Transcription, Direct Multiple Shooting and Differential Dynamic Programming).

To obtain the required derivatives for these algorithms, there are several options. Often, analytic expressions "manually-derived" on paper are considered the ideal solution since they are accurate and fast to compute. However, as shown later, the expressions obtained from deriving Rigid Body Dynamics derivatives manually are fairly complex and difficult to optimize. Therefore, they often lead to poor runtimes. Additionally, the manual process is error prone. To avoid this step, symbolic toolboxes such as Matlab [1], Mathematica [2] or Maxima [3] can be employed. These tools apply known calculus rules in order to symbolically determine the derivative. While this approach is viable in theory, in practice the derivative expressions easily get too large to be suitable for a computationally efficient implementation.

As a shortcut to the previously mentioned approaches, Numeric Differentiation (Num-Diff) is frequently used. In this method, the input to the function to be differentiated is perturbed in each input dimension to obtain an approximation of the derivative using finite differences. However, these methods are prone to numeric errors and they are computationally costly when the input dimension is high.

As another option, Auto Differentiation can be used. Auto Differentiation (Auto-Diff) is a programming "trick" for obtaining the derivatives from source code instructions. Auto-Diff provides the same accuracy as analytical derivatives but also the comfort of obtaining the derivatives in an automated fashion. Furthermore, Auto-Diff derivatives usually have lower complexity than unoptimized analytical derivatives.

A. Auto Differentiation Tools

Over the years, several Auto Differentiation tools have been developed. Auto-Diff can be achieved by Source Code Transform, i.e. parsing the (uncompiled) source code and generating code for the derivatives. However, for advanced programming languages such as C++, this is challenging to implement. Hence, where supported by the programming language, tools often rely on operator overloading, where instead of a normal numeric type, a special Auto-Diff type is injected into the function to be differentiated. It is then used to build an expression graph which later can be differentiated using the chain rule. Popular tools in C++ that employ operator overloading are e.g. Adept [4], Adol-C [5], CppAD [6] and FADBAD++ [7]. While all these tools are based on the same approach, there are significant differences both in performance as well as in functionality [8]. The performance difference usually stems from the implementation of the expression graph, i.e. how fast it can be differentiated and evaluated. In terms of functionality, especially higher order derivatives make a key difference. They are difficult to implement and thus not supported by many tools. In this work, we are not proposing a new Auto-Diff tool but rather discuss how the existing tools can be leveraged when working with Rigid Body Dynamics.

B. Contributions

Auto-Diff is already widely used in the mathematical optimization community - but it is only slowly gaining popularity in the robotics community. In this work, we illustrate the potential of using Auto-Diff for robotics. We demonstrate that it outperforms analytical derivatives in terms

[1]Agile & Dexterous Robotics Lab, ETH Zurich, Switzerland, {neunertm, mgiftthaler, buchlij} @ethz.ch
[2]Department of Advanced Robotics, Istituto Italiano di Tecnologia, Genova, Italy, {marco.frigerio, claudio.semini} @iit.it

Preprint October 26, 2016
Accepted for publication at the 2016 IEEE International Conference on Simulation, Modelling, and Programming for Autonomous Robots, San Francisco

of computational complexity while preventing significant overhead in obtaining them. To employ Auto-Diff on Rigid Body Dynamics, we extend our open source modelling tool "RobCoGen" [9] to be Auto-Diff compatible. The resulting framework allows for obtaining optimized derivatives of well-known Rigid Body Dynamics quantities and algorithms as used by most optimization, estimation and optimal control based algorithms.

Furthermore, rather than directly applying Auto-Diff at runtime, we perform an Auto-Diff code-generation step for the derivatives. In this step, the dynamic expression graph is converted to pure C code. This eliminates the overhead of the expression graph, which we demonstrate is crucial for good performance. Furthermore, the resulting C code is real-time capable and can be run on micro-controllers, in multi-threaded applications or even on GPUs. This allows for directly using it in hard real-time control loops and embedded platforms but also for massive sampling in data driven methods. Since RobCoGen is an open source library, we provide the community with a tested, efficient and easy to use tool for modelling, analyzing and controlling Rigid Body Systems. As an example use case, we apply it to a Trajectory Optimization approach. The results from this example application underline the benefits of using Auto Differentiation for Rigid Body Dynamics.

C. Related Work

Existing libraries that implement auto differentiable Rigid Body Dynamics are rare. Drake [10] supports Auto-Diff for gradient computations of dynamics. However, it relies on Eigen's [11] Auto-Diff module which cannot provide higher order derivatives and does not support code-generation for derivatives. Also, the code is not optimized for speed. Instead, it relies on dynamic data structures, which introduces significant overhead and limits the usability of the library for hard real-time, multi-threaded and embedded applications. MBSlib [12] also provides Auto-Diff support and does so in a more rigorous manner. Yet, it also relies on dynamic data structures, limiting efficiency and potential embedded and control applications. It is unclear whether MBSlib is compatible with any existing Auto-Diff code-generation framework. There are also efforts of deriving simplified analytic derivatives of Rigid Body Dynamics [13]. However, the resulting expressions have to be implemented manually and do not necessarily fully exploit efficient Rigid Body Dynamics algorithms, leading to more complex expressions with runtime overhead. A thorough discussion on this issue is presented in Section II. A comparison between Symbolic and Automatic Differentiation for Rigid Body Kinematics is conducted in [14]. However, the authors do not perform a code-generation step for Auto-Diff which, as we will see later, significantly improves performance.

There is considerable research on how to use Auto-Diff to model and simulate Rigid Body systems, e.g. [15], [16], [17]. However, this work is focused on solving the Rigid Body Dynamics differential equations in order to obtain equations of motions rather than explicitly computing their derivatives. Similar approaches are also used to perform a sensitivity analysis or parametric design optimizations, as e.g. in [18]. However, this research field is only partially concerned with explicit derivatives of kinematics and dynamics algorithms. Additionally, metrics such as complexity and runtime, which are crucial for online control and estimation, are not the primary focus of this community.

II. Rigid Body Dynamics and Kinematics

When dealing with Rigid Body Systems, we are usually interested in one or multiple of the following quantities (or elements of those): forward/inverse dynamics, forward/inverse kinematics, the transforms between joints/links and end-effector/contact Jacobians. In robotics, Rigid Body Dynamics are often expressed as

$$\boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) + \boldsymbol{G}(\boldsymbol{q}) = \boldsymbol{J}_c^T(\boldsymbol{q})\lambda + \boldsymbol{S}^T\tau \qquad (1)$$

where $\boldsymbol{q}$ represents the rigid body state in generalized coordinates, $\boldsymbol{M}$ is the Joint Space Inertia Matrix, $\boldsymbol{C}$ are the Coriolis and centripetal terms, $\boldsymbol{G}$ is the gravity term, $\boldsymbol{J}_c$ is the contact Jacobian, λ are external forces/torques, $\boldsymbol{S}$ is the selection matrix and τ represents the joint forces/torques. $\boldsymbol{S}$ maps input forces/torques to joints and is used to model underactuated systems. In case of a fully actuated system with directly driven joints, $\boldsymbol{S}$ is identity. For readability, we drop the dependency of these quantities on $\boldsymbol{q}$ and $\dot{\boldsymbol{q}}$ in the following.

A. Forward Dynamics and its Derivatives

If we want to know how a system reacts to given joint torques and external forces, we can compute the forward dynamics expressed as

$$fd(\boldsymbol{q}, \dot{\boldsymbol{q}}, \tau) = \ddot{\boldsymbol{q}} = \boldsymbol{M}^{-1}(\boldsymbol{J}_c^T\lambda + \boldsymbol{S}^T\tau - \boldsymbol{C} - \boldsymbol{G}) \qquad (2)$$

Featherstone [19] shows that computing $\boldsymbol{M}$ (without inverting it) already has a worst-case complexity $\mathcal{O}(n^3)$ where n is the number of rigid bodies, which makes naively evaluating (2) very expensive. Therefore, he proposes several algorithms for factorizing $\boldsymbol{M}$ and its inversion. Additionally, he also propses the Articulated Rigid Body Algorithm, which computes the entire forward dynamics equation (2) and only has complexity $\mathcal{O}(n)$.

To get the derivatives of (2) we can naively apply the chain rule:

$$\begin{aligned}\frac{\partial(fd)}{\partial \boldsymbol{q}} = &\frac{d\boldsymbol{M}^{-1}}{d\boldsymbol{q}}\left(\boldsymbol{J}_c^T\lambda + \boldsymbol{S}^T\tau - \boldsymbol{C} - \boldsymbol{G}\right) \\ &+ \boldsymbol{M}^{-1}\left(\frac{d\boldsymbol{J}_c^T\lambda}{d\boldsymbol{q}} - \frac{d\boldsymbol{C}}{d\boldsymbol{q}} - \frac{d\boldsymbol{G}}{d\boldsymbol{q}}\right)\end{aligned} \qquad (3)$$

$$\frac{\partial(fd)}{\partial \dot{\boldsymbol{q}}} = \boldsymbol{M}^{-1}\frac{d\boldsymbol{C}}{d\dot{\boldsymbol{q}}} \qquad (4)$$

$$\frac{\partial(fd)}{\partial \tau} = \boldsymbol{M}^{-1}\boldsymbol{S}^T \qquad (5)$$

Preprint October 26, 2016
Accepted for publication at the 2016 IEEE International Conference on Simulation, Modelling, and Programming for Autonomous Robots, San Francisco

If we were to further simplify the expression we could use the following identities

$$\frac{d\boldsymbol{M}^{-1}}{d\boldsymbol{q}} = \boldsymbol{M}^{-1}\frac{d\boldsymbol{M}}{d\boldsymbol{q}}\boldsymbol{M}^{-1} \tag{6}$$

$$\frac{d\boldsymbol{M}}{d\boldsymbol{q}} = \sum_{n=0}^{N}\frac{\partial \boldsymbol{J}_n^T}{\partial \boldsymbol{q}}\theta_n \boldsymbol{J}_n + \boldsymbol{J}_n^T\theta_n\frac{\partial \boldsymbol{J}_n^T}{\partial \boldsymbol{q}} \tag{7}$$

where n is the index of a rigid body, θ_n is its fixed inertia matrix and J_n is its state dependent Jacobian. The identity in Equation (7) has been derived in [13] and similar identities are presented for $\frac{d\boldsymbol{C}}{d\boldsymbol{q}}$, $\frac{d\boldsymbol{C}}{d\dot{\boldsymbol{q}}}$ and $\frac{d\boldsymbol{G}}{d\boldsymbol{q}}$. But even without looking at these additional identities, two issues with analytical derivatives become prominent: First, the expressions are fairly large and error prone to implement. Second, there is significant amount of intermediate calculations in the forward dynamics that must be handled carefully to avoid re-computation. Thus, there are three tedious, manual steps involved in implementing analytical derivatives: Careful derivation of the formulas, their correct implementation and intelligent caching of intermediate results. As we show in the experiments, Auto-Diff takes care of all three steps, alleviating the user from all manual work while still providing an equally fast or even faster implementation.

B. Inverse Dynamics and its Derivatives

If we want to know what joint torques are required to achieve a certain acceleration at a certain state, we can compute the inverse dynamics by solving Equation (1) for τ. In case we actuate all joints directly, $\boldsymbol{S}$ is an identity matrix and we get

$$id(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}) = \tau = \boldsymbol{M}\ddot{\boldsymbol{q}} + \boldsymbol{C} + \boldsymbol{G} - \boldsymbol{J}_c^T\lambda \tag{8}$$

For the inverse dynamics computation, Featherstone presents an efficient algorithm, the Recursive Newton-Euler Algorithm [19], which again has complexity $\mathcal{O}(n)$ and avoids computing $\boldsymbol{M}$ and other elements explicitly.

For the fully actuated case, the inverse dynamics derivatives are defined as

$$\frac{\partial(id)}{\partial \boldsymbol{q}} = \frac{d\boldsymbol{M}}{d\boldsymbol{q}}\ddot{\boldsymbol{q}} + \frac{d\boldsymbol{C}}{d\boldsymbol{q}} + \frac{d\boldsymbol{G}}{d\boldsymbol{q}} - \frac{d\boldsymbol{J}_c^T}{d\boldsymbol{q}}\lambda \tag{9}$$

$$\frac{\partial id}{\partial \dot{\boldsymbol{q}}} = \frac{d\boldsymbol{C}}{d\dot{\boldsymbol{q}}} \tag{10}$$

$$\frac{\partial id}{\partial \ddot{\boldsymbol{q}}} = \boldsymbol{M} \tag{11}$$

If we were to analytically simplify these equations further, we could again use the identity in Equation (7) and other identities presented in [13].

In case of an under-actuated robot, such as floating base robots, we can obtain the inverse dynamics by using an (inertia-weighted) pseudo-inverse of $\boldsymbol{S}$ to solve Equation (1) for τ as described in [20]. This is only one possible choice for computing floating base inverse dynamics [21]. However, it will serve as an example and the following discussion extends to the other choices as well. The inertia-weighted pseudo-inverse is defined as $\bar{\boldsymbol{S}} = \boldsymbol{M}^{-1}\boldsymbol{S}^T(\boldsymbol{S}\boldsymbol{M}^{-1}\boldsymbol{S}^T)^{-1}$ and leads to the inverse dynamics expression

$$id(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}) = \tau = (\boldsymbol{S}\boldsymbol{M}^{-1}\boldsymbol{S}^T)^{-1}\ddot{\boldsymbol{q}} + \bar{\boldsymbol{S}}^T(\boldsymbol{C} + \boldsymbol{G} - \boldsymbol{J}_c^T\lambda) \quad . \tag{12}$$

The derivatives of the inverse dynamics then become

$$\frac{\partial(id)}{\partial \boldsymbol{q}} = \frac{d(\boldsymbol{S}\boldsymbol{M}^{-1}\boldsymbol{S}^T)^{-1}}{d\boldsymbol{q}}\ddot{\boldsymbol{q}} + \frac{d\bar{\boldsymbol{S}}^T}{d\boldsymbol{q}}(\boldsymbol{C} + \boldsymbol{G} - \boldsymbol{J}_c^T\lambda) + \bar{\boldsymbol{S}}^T\left(\frac{d\boldsymbol{C}}{d\boldsymbol{q}} + \frac{d\boldsymbol{G}}{d\boldsymbol{q}} - \frac{d\boldsymbol{J}_c^T}{d\boldsymbol{q}}\right) \tag{13}$$

$$\frac{\partial(id)}{\partial \dot{\boldsymbol{q}}} = \bar{\boldsymbol{S}}^T\frac{d\boldsymbol{C}}{d\dot{\boldsymbol{q}}} \tag{14}$$

$$\frac{\partial(id)}{\partial \ddot{\boldsymbol{q}}} = (\boldsymbol{S}\boldsymbol{M}^{-1}\boldsymbol{S}^T)^{-1} \quad . \tag{15}$$

Similar to the forward dynamics case, we see that the derivative expressions become large and their implementation is error prone. Additionally, without careful optimization, we might accidentally introduce significant overhead in the computation, e.g. by not caching intermediate results.

C. Higher-Order Derivatives

In Subsections II-A and II-B we presented the first-order derivatives of the forward and inverse dynamics. However, for some optimal control algorithms such as Differential Dynamic Programming (DDP), we might need second order derivatives of the dynamics in case we want to include them in the system dynamics and/or the cost functions. If we want to further analytically differentiate the derivatives, we have to apply the chain rule to them, resulting in even larger expressions. At this point it becomes highly questionable if analytic derivatives should be implemented manually. Since symbolic computation engines also apply the chain rule and only have limited simplification capabilities, these approaches will face the same issues for second order derivatives. This might also be part of the reason why second order optimal control algorithms, such as DDP, are only rarely applied to more complex robotic systems. Instead, algorithms that approximate the second order derivatives are usually preferred [22]. When using Auto-Diff, we can easily obtain second order derivatives and hence do not face this limitation.

III. Auto Differentiation with RobCoGen

A. The Robotics Code Generator

The Robotics Code Generator (RobCoGen) is a computer program that, given the description of an articulated robot, generates optimized code for its kinematics and dynamics [9]. A simple file format is available to provide the description (model) of the robot. Currently, RobCoGen generates both C++ and Matlab code, implements coordinate transforms, geometric Jacobians and state-of-the-art algorithms for forward and inverse dynamics, as described in [19].

Preprint October 26, 2016
Accepted for publication at the 2016 IEEE International Conference
on Simulation, Modelling, and Programming for Autonomous Robots,
San Francisco

B. Derivatives for RobCoGen

RobCoGen does not natively support any code generation for derivatives of rigid body kinematics or dynamics. However, it can be easily extended to generate C++ code suitable for Auto Differentiation in order to leverage other tools specifically designed for that purpose, such as CppAD. The changes required to support Auto-Diff essentially reduce to generating code templated on the scalar type (rather than using standard `float` or `double`). This simple change enables the use of a variety of Auto-Diff tools based on operator overloading, yet does not prevent the regular usage of the generated code.

It is important to note that RobCoGen uses code-generation to create robot specific Rigid Body Dynamics code. This is not to be confused with Auto-Diff codegen, which uses Rigid Body Dynamics code to compute a derivative function, which is then translated into source code. In this work, we apply Auto-Diff codegen to the output of RobCoGen. Thus, there are two sequential, entirely unrelated code-generation steps involved in this work. First, RobCoGen generates the dynamics and kinematics functions. Then Auto-Diff codegen uses those to create the derivative source code. We will always indicate which type of generated code we are referring to, i.e. whether we refer to the dynamics/kinematics generated by RobCoGen or to the derivatives generated using Auto-Diff codegen[1].

Since RobCoGen implements the most efficient algorithms for dynamics (e.g. the Articulated Rigid Body algorithm for forward dynamics), and since the lowest achievable complexity of Auto-Diff derivatives is proportional to the complexity of the original function [23], it follows that, at least theoretically, we can obtain the most efficient derivatives from RobCoGen as well.

C. Implementation in RobCoGen

Since the interoperability with Auto-Diff tools is ongoing development within the RobCoGen project, we implemented a proof of concept of the approach by modifying existing RobCoGen generated code for the quadruped robot HyQ [24] and a 6 DoF robotic arm.

Our goal is to make the entire RobCoGen generated code auto-differentiable. Thus, we add a scalar template to all algorithms which compute one of the following quantities:

- Forward dynamics: Articulated Rigid Body Algorithm
- Inverse dynamics: Recursive Newton-Euler Algorithm
- Homogeneous-coordinate transforms
- Coordinate transform for spatial vectors
- Jacobians
- Rigid Body quantities such as $\boldsymbol{M}$, $\boldsymbol{C}$ and $\boldsymbol{G}$

We further template all input and parameter types, including state, joint torques, external forces, all inertia parameters etc. Therefore, also uncommon derivatives, e.g. derivatives of inverse dynamics with respect to inertia parameters can be obtained. This feature can be useful for design optimization or parameter estimation applications. Also, Auto-Diff derivatives of essential Rigid Body quantities such as the Joint Space Inertia Matrix $\boldsymbol{M}$ can be computed. This allows to use Auto-Diff to generate custom derivatives for special applications, e.g. when using Pseudo-Inverse based inverse dynamics as shown in Subsection II-B. In such a case, the derivatives in Equations (13) - (15), can still be obtained from Equation (12) using Auto-Diff. The user only has to ensure that the computation of the Pseudo-Inverse is auto-differentiable as well. Furthermore, the user can also generate derivatives of transforms and Jacobians, e.g. for task space control or kinematic planning.

By templating the entire library, the user can specify which methods or quantities to differentiate. Therefore, the differentiation can be tailored to a specific use case, e.g. by computing only required parts of a Jacobian or applying the "Cheap Gradient Principle" [23]. Another benefit is that not all derivatives are generated but only the ones required. To demonstrate how to select and generate derivatives, we provide a reference implementation described below.

D. Reference Implementation of Auto Differentiation

In order to validate the performance and accuracy of the Auto-Diff compatible version of RobCoGen, we provide a reference implementation. For this implementation, we use CppAD as our Auto-Diff tool. CppAD is very mature, well documented, supports higher derivatives and provides an efficient implementation. Still, evaluating the expression graph for computing derivatives comes at an overhead. Therefore, we also employ CppADCodeGen [25], which can generate bare C code for derivatives using CppAD.

Since CppAD operates on scalars rather than full matrices, the generated derivative code is only using scalar expressions and cannot leverage advanced compiler optimizations such as vectorization. While this leads to a slight decrease in performance, the generated code is dependency-free and can easily be run on embedded systems or GPUs. Additionally, memory for intermediate results of the derivative computation can be allocated statically. Thus, the generated code can be run in hard real-time control loops. Despite the fact that the generated code is static, it can be parametrized. Therefore, Rigid Body Dynamics parameters such as mass, Inertia or Center of Mass can be changed at runtime if required. Thus, the generated code is specialized to a certain morphology rather than a specific robot or a set of dynamic properties.

IV. Results

To evaluate the performance of our Auto-Diff approach, we first test the derivatives separately in terms of complexity and accuracy. This test is performed on a model of the underactuated quadruped HyQ, which has a floating base and three joints per leg. In a second test, we use the Auto-Diff derivatives in a Trajectory Optimization approach to verify

[1]While the implementation of both code generation steps differs significantly, the goal is the same. In both cases, specialized code is generated instead of using general dynamic data structures. This eliminates the overhead of transversing such data structures, checking for dimensions, sizes and branches. Furthermore, static data structures are easier to implement on embedded systems and to use in hard real-time applications.

Preprint October 26, 2016
Accepted for publication at the 2016 IEEE International Conference on Simulation, Modelling, and Programming for Autonomous Robots, San Francisco

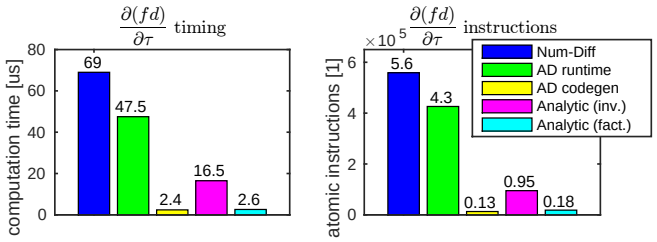

Fig. 1. Comparison between different approaches to compute the forward dynamics derivative with respect to joint torques τ. We measure absolute computation time as well as the number of atomic instructions of the compiled code. Clearly, Num-Diff performs worst. Auto-Diff at runtime performs slightly better. However, Auto-Diff codegen and the analytic factorization perform best. The naive analytic inverse performs significantly worse than both. This underlines that naively implemented analytical derivatives can be significantly inferior to Auto-Diff.

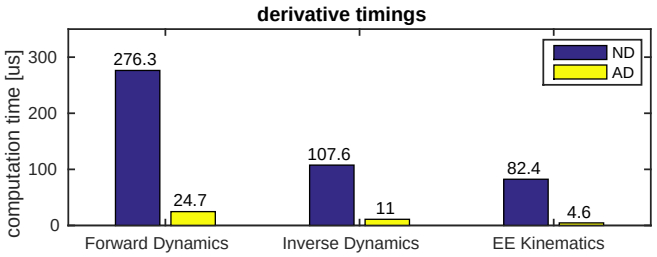

Fig. 2. Comparison between Numeric Differentiation ('ND') and Auto Differentiation using code-generation ('AD') for the computation of three tpyes of derivatives. In the forward dynamics we implement Equations (3)+(4) while in the inverse dynamics we use Equations (9) -(11). Additionally, the derivatives of end-effector position and velocity with respect to the rigid body state $\boldsymbol{q}$ and $\dot{\boldsymbol{q}}$ are timed. We can see that Auto-Diff codegen outperforms Num-Diff by a factor of about 10-20x.

their practical gain. This test is carried out on a model of a fully actuated six degree of freedom robotic arm.

A. Accuracy, Number of Instructions and Timings

In order to compare the different derivative methods, we look at the derivative of the forward dynamics with respect to the joint forces/torques as shown in Equation (5). To ensure that we are computing a dense derivative, we ignore the floating base part of the equation but focus on the bottom right corner of the expression. For the comparison, the derivatives are obtained from different implementations: Single-sided numerical derivatives, Auto-Diff at runtime, Auto-Diff codegen and analytical derivatives. For the analytical derivatives we have two implementations. One computes $\boldsymbol{M}$ using Featherstone's Composite Rigid Body algorithm and later naively inverts it by using Eigen's general $\boldsymbol{L}\boldsymbol{L}^T$ solver. The second analytical implementation uses the Rigid Body Dynamics specific $\boldsymbol{L}^T\boldsymbol{L}$ factorization of $\boldsymbol{M}^{-1}$, which exploits sparsity. Such factorization is described in [19] and implemented by RobCoGen.

We first verify the accuracy by measuring the norm of the difference between the derivative matrices. As expected, the analytical methods and Auto-Diff agree up to machine precision ($< 10^{-13}$). Numerical differentiation however deviates by around $< 10^{-6}$ from all other implementations. While this is small, we will see that it can make a difference in a Trajectory Optimization example.

Additionally, we compare the runtime as well as the number of atomic instructions of all derivative methods. In this test, we average over 10,000 computations and enable the highest optimization level of our compiler. The results of this test are shown in Figure 1. This test shows two interesting findings: Firstly, there is a very significant difference between the Auto-Diff generated derivative code and the Auto-Diff computation at runtime. The generated code is much faster, which is possibly a result of removing the overhead of travelling through the expression graph and enabling the compiler to optimize the code. Secondly, we see that – despite the fact that the RobCoGen's factorization of $\boldsymbol{M}^{-1}$ exploits the sparsity and structure of the problem – it is not able to outperform the Auto-Diff generated derivative in terms of instructions and runtime. The factorization is still about 10% slower than Auto-Diff codegen. This strongly underlines the hypothesis that even carefully implemented analytical derivatives are usually equally or more expensive than those generated by Auto-Diff. On the other hand, naively implemented derivatives, even though analytic, can lead to significant overhead. This overhead, as present in the analytical inversion, is over 600% in the test above.

Finally, we test more complex expressions. We compare the derivatives of the forward dynamics as well as of the fully-actuated inverse dynamics both taken w.r.t. the state, as shown in Equations (3)+(4) and Equations (9)-(11) respectively. Also here, we are not using the naive implementations but efficient Featherstone algorithms for all derivative types. Additionally, we time the computation of the end-effector position $\boldsymbol{p}$ and velocity $\dot{\boldsymbol{p}}$ differentiated with respect to the full state, $\boldsymbol{q}$ and $\dot{\boldsymbol{q}}$. Here, $\boldsymbol{p}$ and $\dot{\boldsymbol{p}}$ are expressed in a fixed inertial frame ("world" frame). The forward kinematics to be differentiated are given by $\boldsymbol{p} = \boldsymbol{T}_{ee}\boldsymbol{q}$ and $\dot{\boldsymbol{p}} = \boldsymbol{J}_{ee}\dot{\boldsymbol{q}}$ wher $\boldsymbol{T}_{ee}$ and $\boldsymbol{J}_{ee}$ are end-effector transform and Jacobian respectively. We compute the forward kinematics derivatives for all four feet of the quadruped simultaneously. Since we do not have optimized analytical derivatives available for this test, we compare the performance of Num-Diff to the one of Auto-Diff codegen. The results are shown in Figure 2. While the difference in runtime between Num-Diff and Auto-Diff codegen is not as large as in the previous example, it is still significant. Auto-Diff codegen is about 10-20x faster than Num-Diff in all three cases. Since approaches like Trajectory Optimization rely heavily on computing derivatives, we evaluate in the next experiment how much of a practical difference this speedup makes.

B. Trajectory Optimization using Direct Multiple Shooting

As a practical test, we apply the Direct Multiple Shooting (DMS) method [26] to a Trajectory Optimization task on a model of a robotic arm. DMS is a widely-spread approach for numerically solving Optimal Control Problems. An originally infinite-dimensional optimal control problem can be transformed into a Nonlinear Program (NLP) by discretizing it into a finite set of state and control decision variables.

Preprint October 26, 2016
Accepted for publication at the 2016 IEEE International Conference
on Simulation, Modelling, and Programming for Autonomous Robots,
San Francisco

Those are used for forward integrating the so-called 'shots', which are matched at the nodes using continuity constraints. DMS can handle inequality path constraints, e.g. task space obstacles or control input constraints. A detailed description of the method is beyond the scope of this paper and we refer the interested reader to [27] for an overview. However, we showcase the performance gain for DMS when using Auto-Diff generated derivatives of Rigid Body Dynamics instead of numerical derivatives.

We have implemented a custom DMS problem generator which hands over the resulting Nonlinear Program to off-the-shelf NLP solvers. Amongst others, it achieves efficiency through exploiting the inherent block-sparse structure of the DMS constraint Jacobian. We are using a standard fourth order Runge-Kutta (RK4) integration scheme for propagating the shots, and its analytic derivative for calculating the trajectory sensitivities w.r.t. neighbouring decision variables on the fly. The sensitivities are functions of the system dynamics derivatives w.r.t. state and control at every integration sub-step in time, which can be evaluated efficiently using the Auto-Diff generated derivatives.

In this example, we use DMS for planning optimal trajectories for a fixed-base 6 DoF robotic arm. The task considered is computing optimal state and input trajectories between an initial and a desired joint configuration which avoid a static, spherical task-space obstacle. The setup is illustrated in Figure 3. Between experiments, we vary different parameters: single/multi threaded implementation, the NLP solvers (SNOPT [28] / IPOPT [29]) and the type of Rigid Body Dynamics derivatives (Auto-Diff codegen / Num-Diff). Additionally, we use two different sets of initial and terminal joint configurations. The first set corresponds to initial and final states with a fully stretched arm, whereas the second one features initial and final joint configurations with a bent elbow. For each problem, we select a time horizon of 4.0 seconds and discretize it into 20 equally spaced shots. We choose zero-order hold interpolation of the control inputs between the nodes and propagate the system state with a constant RK4 step-size of 50 ms. The initial guess for the state decision variables consists of zero joint velocities and equidistantly spaced joint positions that are obtained by direct interpolation between the initial and terminal joint configuration. The initial guess for the control input decision variables are the steady-state joint torques computed by the Recursive Newton Euler inverse dynamics evaluated at the corresponding states. For obstacle avoidance, we define a discrete number of collision points on the last three links of the robot. For each of those collision points, an inequality path constraint results at every node of the DMS problem.

Despite varying parameters, the solvers always converge to the same two solutions for the straight and the bent elbow scenario. Figure 3 shows the resulting trajectory for the case of the bent elbow. In this illustration, we can see the effect of not only performing collision checking on the end-effector. If we were to only check collisions at the end-effector (red), the trajectory would follow the surface of the sphere, leading to collisions in the parent link of the end-effector. Instead, a collision free trajectory is found.

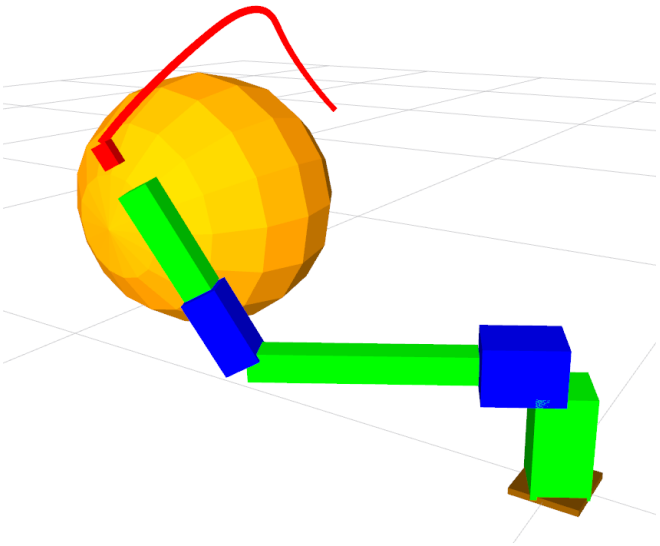

Fig. 3. A screenshot of the Direct Multiple Shooting planning environment showing the 6 DoF robot arm, a static, spherical obstacle and the resulting optimal trajectory plotted for the end-effector frame. Since collision checking is not only performed on the end-effector but on discrete points on the lower links, the end-effector trajectory does not follow the sphere's surface.

To assess the performance, we look at the runtime and the number of iterations that the solvers require for the different scenarios. All tests are run on an Intel Xeon E5 processor. Figure 4 summarizes the results. The scenario is indicated by the solver ('IPOPT'/'SNOPT') and the joint configurations (straight '|' and bent '>' elbow). It is evident that independent of the scenario and the solver, Auto-Diff codegen outperforms Num-Diff in runtime by a factor of usually 200% or more. In three out of four cases, Auto-Diff also uses slightly less iterations. Furthermore, Auto-Diff reduces the difference between the multi threaded and single threaded implementation and we obtain runtimes per iteration of less than 10 ms. This would even allow for running Direct Multiple Shooting as a single-iteration Model Predictive Controller at 100 Hz.

V. Discussion and Outlook

In this work, we propose how Auto-Diff can be efficiently applied to Rigid Body Dynamics algorithms. Based on this study we extended our open source Robotics Code Generator, enabling full Auto Differentiation compatibility. Further, we underline the importance of generating source code for the derivatives obtained from Auto-Diff, avoiding the usual overhead during evaluation. The resulting derivative code outperforms even carefully hand-designed and optimized analytical derivative implementations. In a practical example, we show that Auto-Diff codegen can significantly improve performance in control and optimization applications.

Currently, we are finalizing tests on the beta implementation of RobCoGen's Auto-Diff feature. Additionally, we are applying the presented approach to improve the runtime of

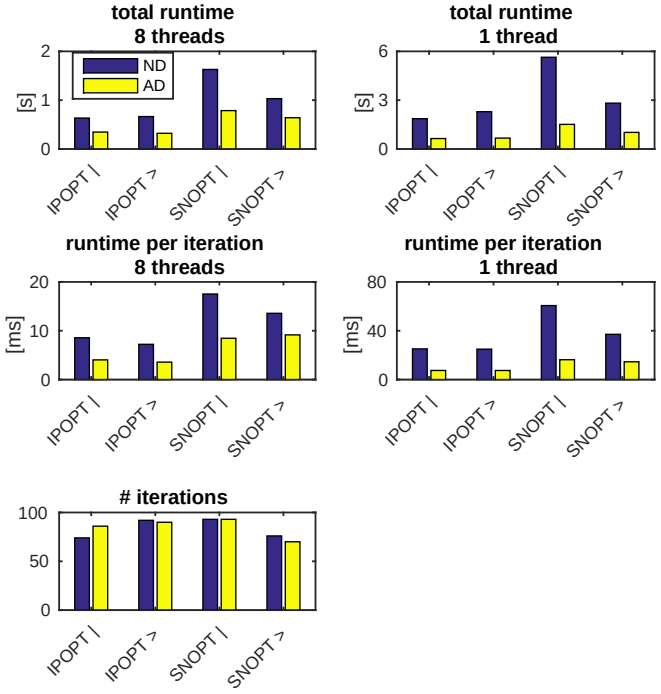

Fig. 4. Comparison between Num-Diff ('ND', blue) and Auto-Diff codegen ('AD', yellow) for various test cases. We vary solvers (IPOPT/SNOPT), the number of threads (1/8) and the task (straight '|' or bent '>' elbow). Results show that Auto-Diff codegen is significantly faster in runtime, especially in single core operation. Also, with one exception, Auto-Diff requires less iterations than Num-Diff to converge to the same solution.

our Trajectory Optimization algorithms, enabling their usage as online planners and Model Predictive Controllers.

Source Code and Example Code

The latest version of RobCoGen is available at `https://bitbucket.org/mfrigerio17/roboticscodegenerator`, where we will also release the Auto-Diff compatible version. As an example, the Auto-Diff compatible RobCoGen output for the quadruped HyQ as well as all derivative timing and accuracy tests are available at `https://bitbucket.org/adrlab/hyq_gen_ad`.

Acknowledgements

The authors would like to thank Diego Pardo for testing the implementation and Markus Stäuble for his support with the Trajectory Optimization case study. This research has been funded through a Swiss National Science Foundation Professorship award to Jonas Buchli, the NCCR Robotics and the NCCR Digital Fabrication.

References

[1] MathWorks Inc. (2016) Matlab & simulink software. [Online]. Available: www.mathworks.com/

[2] Wolfram Research. (2016) Mathematica software. Wolfram Research, Inc. Champaign, Illinois. [Online]. Available: www.wolfram.com/

[3] Maxima. (2016) Maxima, a Computer Algebra System. [Online]. Available: maxima.sourceforge.net/

[4] R. J. Hogan, "Fast reverse-mode automatic differentiation using expression templates in C++," *ACM Transactions on Mathematical Software (TOMS)*, vol. 40, no. 4, p. 26, 2014.

[5] A. Walther and A. Griewank, "Getting started with adol-c," *Combinatorial scientific computing*, vol. 20121684, 2012.

[6] B. M. Bell, "CppAD: a package for C++ algorithmic differentiation," *Computational Infrastructure for Operations Research*, 2012. [Online]. Available: www.coin-or.org/CppAD/

[7] C. Bendtsen and O. Stauning, "Fadbad, a flexible C++ package for automatic differentiation," *Department of Mathematical Modelling, Technical University of Denmark*, 1996.

[8] B. Carpenter, M. D. Hoffman, M. Brubaker, D. Lee, P. Li, and M. Betancourt, "The stan math library: Reverse-mode automatic differentiation in C++," *CoRR*, vol. abs/1509.07164, 2015. [Online]. Available: http://arxiv.org/abs/1509.07164

[9] M. Frigerio, J. Buchli, D. G. Caldwell, and C. Semini, "RobCoGen: a code generator for efficient kinematics and dynamics of articulated robots, based on Domain Specific Languages," *Journal of Software Engineering for Robotics (JOSER)*, vol. 7, no. 1, pp. 36–54, 2016.

[10] R. Tedrake, "Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems," 2014. [Online]. Available: http://drake.mit.edu

[11] Eigen Linear Algebra Library. (2016). [Online]. Available: eigen.tuxfamily.org

[12] J. Wojtusch, J. Kunz, and O. von Stryk, "MBSlib - An Efficient Multibody Systems Library for Kinematics and Dynamics Simulation, Optimization and Sensitivity Analysis," *Robotics and Automation Letters, IEEE*, vol. 1, no. 2, pp. 954–960, 2016.

[13] G. Garofalo, C. Ott, and A. Albu-Schffer, "On the closed form computation of the dynamic matrices and their differentiations," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2013, pp. 2364–2359.

[14] A. Dürrbaum, W. Klier, and H. Hahn, "Comparison of automatic and symbolic differentiation in mathematical modeling and computer simulation of rigid-body systems," *Multibody System Dynamics*, vol. 7, no. 4, pp. 331–355, 2002.

[15] P. Eberhard and C. Bischof, "Automatic differentiation of numerical integration algorithms," *Mathematics of Computation of the American Mathematical Society*, vol. 68, no. 226, pp. 717–731, 1999.

[16] D. T. Griffith, J. D. Turner, and J. L. Junkins, "Some applications of automatic differentiation to rigid, flexible, and constrained multibody dynamics," in *ASME 2005 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers, 2005, pp. 271–281.

[17] A. Callejo, S. H. K. Narayanan, J. G. de Jalón, and B. Norris, "Performance of automatic differentiation tools in the dynamic simulation of multibody systems," *Advances in Engineering Software*, vol. 73, pp. 35–44, 2014.

[18] C. H. Bischof, "On the automatic differentiation of computer programs and an application to multibody systems," in *IUTAM Symposium on Optimization of Mechanical Systems*. Springer, 1996, pp. 41–48.

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

[20] J. Nakanishi, M. Mistry, and S. Schaal, "Inverse dynamics control with floating base and constraints," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*. IEEE, 2007, pp. 1942–1947.

[21] L. Righetti, J. Buchli, M. Mistry, and S. Schaal, "Inverse dynamics control of floating-base robots with external constraints: A unified view," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 1085–1090.

[22] A. Sideris and J. E. Bobrow, "An efficient sequential linear quadratic algorithm for solving nonlinear optimal control problems," *Automatic Control, IEEE Transactions on*, vol. 50, no. 12, pp. 2043–2047, 2005.

[23] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Siam, 2008.

[24] 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," *IMechE Part I: J. of Systems and Control Engineering*, vol. 225, pp. 831–849, 2011.

[25] J. R. Leal. (2016) CppADCodeGen. [Online]. Available: github.com/joaoleal/CppADCodeGen

[26] H. G. Bock and K.-J. Plitt, "A multiple shooting algorithm for direct solution of optimal control problems," in *Proceedings of the IFAC World Congress*, 1984.

[27] M. Diehl, H. G. Bock, H. Diedam, and P.-B. Wieber, "Fast direct multiple shooting algorithms for optimal robot control," in *Fast motions in biomechanics and robotics*. Springer, 2006, pp. 65–93.

[28] P. E. Gill, W. Murray, and M. A. Saunders, "SNOPT: An SQP algorithm for large-scale constrained optimization," *SIAM review*, vol. 47, no. 1, pp. 99–131, 2005.

[29] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical programming*, vol. 106, no. 1, pp. 25–57, 2006.