Automatic Differentiation on Differentiable Manifolds as a Tool for Robotics

Hannes Sommer, Cédric Pradalier, and Paul Furgale

Abstract Automatic differentiation (AD) is a useful tool for computing Jacobians of functions needed in estimation and control algorithms. However, for many interesting problems in robotics, state variables live on a differentiable manifold. The most common example are robot orientations that are elements of the Lie group SO(3). This causes problems for AD algorithms that only consider differentiation at the scalar level. Jacobians produced by scalar AD are correct, but scalar-focused methods are unable to apply simplifications based on the structure of the specific manifold. In this paper we extend the theory of AD to encompass handling of differentiable manifolds and provide a C++ library that exploits strong typing and expression templates for fast, easy-to-use Jacobian evaluation. This method has a number of benefits over scalar AD. First, it allows the exploitation of algebraic simplifications that make Jacobian evaluations more efficient than their scalar counterparts. Second, strong typing reduces the likelihood of programming errors arising from misinterpretation that are possible when using simple arrays of scalars. To the best of our knowledge, this is the first work to consider the structure of differentiable manifolds directly in AD.

1 Introduction

Computation of the Jacobian matrix of a nonlinear function is an essential part of many estimation and control algorithms and, as such, it is a ubiquitous task within robotics and computer vision. When faced with the task of implementing Jacobian

C. Pradalier
GeorgiaTech Lorraine, Metz, France, e-mail: cedric.pradalier@gatech.edu

H. Sommer and P. Furgale
ETH Zürich, Zürich, Switzerland, e-mail: firstname.lastname@mavt.ethz.ch

computation within a computer program, there are essentially four choices[1]. First, one can hand-compute the analytical expression. This is easy for simple functions, careful attention may be paid to the correct handling of singularities in the operations, and the computations may be hand-optimized for speed. However, it may become arduous to compute and verify Jacobians every time a small change is needed. Second, one may approximate the derivatives numerically. This is easy to implement but the resulting Jacobians may be inaccurate for highly nonlinear functions and finite differencing schemes can fail completely when the functions include conditional statements. Third, one may use symbolic differentiation tools to generate code from the nonlinear functions. The resulting Jacobians are accurate and highly efficient to evaluate but this method involves a pre-processing step and there is no guarantee that the automatically generated code correctly handles singularities. Finally, one may use Automatic Differentiation (AD) to compute the Jacobian matrices algorithmically. AD algorithms compute derivatives by exploiting the deterministic nature of derivative computation. The derivatives of atomic operations are implemented by the AD toolkit. The derivatives of more complex functions are constructed by applying the chain rule at each operation and bookkeeping the results. The application of AD is extremely easy using one of the many tools available[2]. Jacobians computed by this method are as accurate as their hand-coded counterparts but they are less efficient to compute and, again, may not handle all singularities.

In terms of accuracy and evaluation speed, hand coding or code generation are the clear choices. However, they share the common drawback that any change in the original function requires the Jacobian computation to be updated in lockstep. When prototyping, AD alleviates this requirement as changes in the nonlinear function are automatically reflected in the Jacobian computation. In our experience, researchers live continually in the prototyping phase and so, for problems in robotics focusing on estimation and control, AD is a tool to accelerate research.

However, for many interesting problems in robotics, state variables live on a Differentiable Manifold (DM). In robotics the most important DMs are the proper rotation and Euclidean groups in two- and three-dimensional Euclidean space ($\{$SO(2), SO(3)$\}$ and $\{$SE(2), SE(3)$\}$ respectively) for rigid-body kinematics, as well as the two-dimensional sphere, S^2, (e.g. for bearing vectors in sensor models [6]). Unit-length quaternions (elements of $S^0(\mathbb{H})$) are another popular choice for representing orientations.

The handling of state variables on DMs within estimation and control has been the subject of active research for many years in robotics and aerospace [12, 4, 1, 3, 5], but this analysis has not made it into the AD literature. The theory and implementation of AD is decidedly focused on the derivatives of *scalar operations* as the computational atoms [11]. It is possible to coax AD packages to compute the correct Jacobians for functions that operate on elements of a DM, but

[1] This paper will focus on the scale of problem generally encountered in estimation and control algorithms in robotics and not deal with methods used for larger-scale problems such as finding the Jacobian of a fluid simulation with respect to airfoil parameters (c.f. [7]).

[2] See http://www.autodiff.org/ for an extensive list of AD tool-kits spanning many popular programming languages.

this involves lifting the derivative computations to the the outer space. Working in the outer space precludes the possibility to utilize the special structure of the manifold to simplify derivative computation.

In this paper we extend the theory of AD to DMs by considering block operations as the atoms of computation, and exploiting the specific structure of the manifold to increase the efficiency of computation. Throughout this paper we will use expressions involving unit-length quaternions as our main example but the method is applicable to any DM. A pictorial representation of our contribution is presented in Figure 1. This figure compares scalar AD with our approach by plotting the computation graph for a simple example in which a unit-length quaternion, q, is used to rotate a point, $\mathbf{v}$.

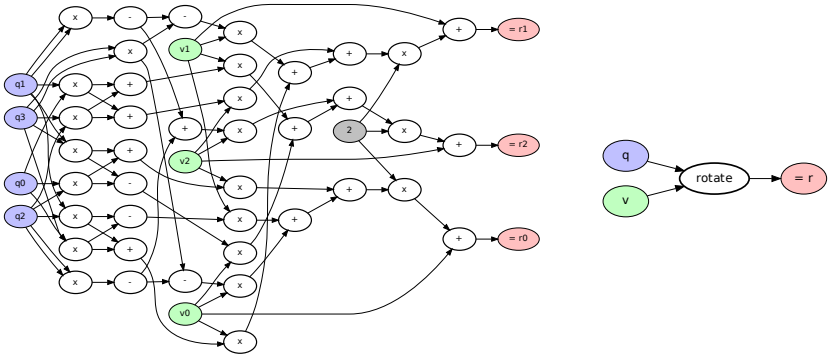

Fig. 1 A simple example in which a unit-length quaternion, q, is used to rotate a point, $\mathbf{v}$. Left: The computational graph associated with the calculation from the point of view of standard scalar-focused AD algorithms. Right: The computational graph from the point of view of our method. Although the evaluation of these two graphs is the same, computing the Jacobian from the left graph with a dual-number approach requires **168 multiplication and 156 additions**, whereas only **3 multiplications** are required by taking advantage of the DM structure on the right (see section 2.3.2 for details).

We see the contributions of this work to be the following:

- We extend the theory of AD to operate on DMs. Because the atoms of computation become block operations, we call this Block Automatic Differentiation (BAD). To the best of our knowledge, this is the first work to make this extension.
- We present a prototype C++ implementation of BAD with a number of desirable properties. First, it is faster than AD that does not explicitly consider differentiation with respect to the underlying manifold. Second, rather that simply overloading operators on scalar or matrix types (a standard method of developing AD), we develop a type-safe system where every value belongs to a specific mathematical type with a well-defined tangent space for differentiation. This makes the system easier to use than standard scalar AD and guards against coding errors possible from the accidental misinterpretation of arrays of scalar types.

- We perform a rigorous comparison of our method against the dual-number approach to AD implemented in the Google Ceres optimization package [2].

The rest of the paper is organized as follows. Section 2 reviews the background theory and presents the basis for our BAD algorithm. Section 3 describes our prototype implementation of BAD as a C++ library. Our experimental results are presented in Section 4 and we conclude in Section 5.

2 Theory

This section reviews the relevant theory on AD and DMs, and then describes our extension to bring the two concepts together.

2.1 Automatic Differentiation

An excellent introduction to Automatic (or Algorithmic) Differentiation is presented in [11] and [10]. This section attempts to summarize the content therein to bring the minimum of context on the topic. The interested reader is referred to these documents and the references therein for further details.

In the realm of computer science, AD is an ancient field, first developed to compute derivatives on specialized computers in the late 1960s. Widespread application of the technique came with the development of numeric tools in Fortran in the last two decades of the 20^{th} century.

The basic idea of AD is easy to explain by imagining the evaluation of mathematical expressions represented by a *computation graph*. When interpreted by a compiler, a numeric expression is represented as a computation tree, with constants or variables as leaves and operators ('$+$','$-$',...) and functions (cos, sin, exp, ...) as interior nodes. Because of sub-expressions being reused or compiler optimization, this tree becomes a computation graph (more precisely, a directed acyclic graph). Figure 1 depict such computation graph for rotating a 3D point with a quaternion.

The foundation of AD stems from the fact that the rules of differentiation are deterministic and they can be applied mechanically and recursively to the computation graph. Rather than deriving a formula for the derivative of an expression, the derivative is computed algorithmically by traversing the graph and using a combination of the chain rule and known operator differentiation rules. Traversing the graph from the leaves to the root is known as 'forward' AD and traversing it from the root to the leaves is known as 'backward' AD. Backward AD is more complex to implement and requires more storage but typically requires fewer operations, whereas forward AD can be implemented in a straightforward manner with operator overloading and/or dual numbers. In the latter case, the AD system evaluate the expression of interest, initially designed for numeric values, on a specific data structure grouping the expression value and its derivative(s) at each node of the graph.

A recent example of such dual-numbers is the Jet class used by Google Ceres[2] to implement its AD feature.

In traditional languages not supporting operator overloading (e.g. Fortran 77), AD has been implemented by a pre-processor stage that would parse an expression in the original language and generate code to compute the expression and its derivative(s), to be compiled and linked in the final program.

Our contribution lies in the fact that adding knowledge about the underlying DM, we can dramatically simplify the computational graph (e.g. Figure 1). By considering differentiation and the chain rule at the block level, we are able to exploit specific problem structure and increase computational efficiency. We compare our approach to the "dual-number" AD [9] implemented in Google Ceres[2].

The basic idea of the dual-number concept to do AD for scalar functions is to calculate the derivatives value alongside the functions values through the computation graph from the leaves to the root. This is done by applying all scalar operations to pairs of value v and derivative's value d $\langle v,d\rangle$. The derivative's value starts with 1, while the values start with the value the function is evaluated at. The scalar operations apply normally on the value parts, but a special corresponding operation to the derivative. For example $\langle v_1,d_1\rangle * \langle v_2,d_2\rangle := \langle v_1v_2, d_1v_2 + d_2v_1\rangle$. To calculate gradients of functions in multiple scalar variables one simply keeps one second number for each variable and initializes each variable with the corresponding second number as 1 and the others as zero. For example $a*b$ one would then calculate the value and gradient at $a = 2, b = 3$ as follows : $\langle 2,1,0\rangle + \langle 3,0,1\rangle = \langle 2*3, 1*3+0*2, 0*3+1*2\rangle = \langle 6,3,2\rangle$. The gradient can then be extracted as the row vector of all the resulting second numbers $(3,2)$.

2.2 Differentiable Manifolds and Jacobians

The concepts of differential geometry have been adopted by the robotics community as the mathematical underpinnings of kinematics and dynamics. An excellent introduction is available in Murray et al. [8] but we will provide a brief overview of the concepts necessary to understand the idea of BAD.

For the scope of this paper a m-dimensional DM (with $m \in \mathbb{N}$) is a set, $\mathscr{M}$, together with an *m-atlas*, $A_{\mathscr{M}}$, inducing a second-countable Hausdorff topology on $\mathscr{M}$. To precisely define the notion of an atlas is beyond the scope of this paper, but informally $A_{\mathscr{M}}$ is a collection of *charts*, where each chart, $\varphi : U_\varphi \to \mathscr{M}$, is an invertible mapping defined on an open subset U_φ of $\mathbb{R}^m$ onto a subset of $\mathscr{M}$. The atlas provides a *differentiable structure* to the manifold $\mathscr{M}$. Using these charts, we can do differential calculus for functions between DMs, which include all finite dimensional vector spaces (by assigning DMs that reproduce the usual calculus on vector spaces).

2.2.1 Defining a Local Jacobian

Consider a mapping $f : \mathscr{M} \to \mathscr{N}$ between the m-dimensional DM $\mathscr{M}$ and the n-dimensional DM $\mathscr{N}$. At a point $p \in \mathscr{M}$, for f there exist a notion of differentiability and a formal local linearization, the *differential*, denoted with $\mathrm{d}_p f$. Its rigorous definition is also beyond the scope of this paper. Informally it plays the role of a derivative in a DM context. For algorithmic treatment, one requires a matrix representing this differential, but the usual Jacobian of nonlinear functions is only defined for mappings between vector spaces.

However, after choosing charts $\varphi_{\mathscr{M}}$ and $\varphi_{\mathscr{N}}$ around p and $f(p)$ respectively, one may define $\hat{\mathbf{f}} := \varphi_{\mathscr{N}}^{-1} \circ f \circ \varphi_{\mathscr{M}}$ at $\hat{\mathbf{p}} := \varphi_{\mathscr{M}}^{-1}(p)$, a mapping $\mathbb{R}^m \supset \mathrm{dom}(\varphi_{\mathscr{M}}) \to \mathbb{R}^n$, where $\mathrm{dom}(\varphi_{\mathscr{M}})$ denotes the domain that $\varphi_{\mathscr{M}}$ is defined on. We call f differentiable at p iff $\hat{\mathbf{f}}$ is differentiable at $\hat{\mathbf{p}}$ and define f's *Jacobian* in these charts with,

$$\mathrm{J}_p f := \mathrm{J}_{\hat{\mathbf{p}}} \hat{\mathbf{f}} = \mathrm{J}_{\varphi_{\mathscr{M}}^{-1}(p)} (\varphi_{\mathscr{N}}^{-1} \circ f \circ \varphi_{\mathscr{M}}). \tag{1}$$

These Jacobians can then play the same role in algorithms dealing with manifolds as the usual Jacobians do for nonlinear mappings between vector spaces. The algorithmic complexity to calculate them depends not only on f but also on the chosen charts. The latter feature is one of the underlying principles that BAD tries to exploit.

2.2.2 Defining a Global Jacobian

Usually manifolds are algorithmically represented as *embedded sub-manifold* of an *outer* $\mathbb{R}$*-vector space* $\mathscr{O}_{\mathscr{M}} := \mathbb{R}^{O_{\mathscr{M}}}$, with $O_{\mathscr{M}} > m$. This means informally that $\mathscr{M}$ is represented by a subset of $\mathscr{O}_{\mathscr{M}}$ for which the differentiable structure inherited from $\mathscr{O}_{\mathscr{M}}$ makes it a DM diffeomorphic to $\mathscr{M}$. Hence, points of $\mathscr{M}$ can easily be represented as the corresponding elements of $\mathscr{O}_{\mathscr{M}}$. A simple example is the Lie group of unit-length quaternions, a three-dimensional manifold that is often stored and manipulated as the S^3 sub-manifold of $\mathbb{R}^4$. In the following, we will assume that $\mathscr{M}$ and $\mathscr{N}$ are embedded sub-manifolds of $\mathbb{R}^{O_{\mathscr{M}}}$ and $\mathbb{R}^{O_{\mathscr{N}}}$ respectively.

This situation allows for a special way to acquire a Jacobian for f. One can choose a mapping $\tilde{\mathbf{f}} : V \to \mathscr{O}_{\mathscr{N}}$ defined on an open environment $V \subset \mathscr{O}_{\mathscr{M}}$ of p such that $f|_{\mathscr{M}\cap V} = \tilde{\mathbf{f}}|_{\mathscr{M}\cap V}$ and calculate the Jacobian $\mathrm{J}_p \tilde{\mathbf{f}} \in \mathbb{R}^{O_{\mathscr{N}} \times O_{\mathscr{M}}}$ of $\tilde{\mathbf{f}}$ instead. Here, $F|_A$ denotes the restriction of a mapping F on a set A. We will call $\mathrm{J}_p \tilde{\mathbf{f}}$ a *global Jacobian* of f. Note that it does not require a choice of charts but depends on the choice of $\tilde{\mathbf{f}}$ instead. For some applications this global Jacobian is already usable but for many it introduces instabilities and performance loss because it calculates a matrix that is bigger than necessary (recall that the *local* $\mathrm{J}_p f \in \mathbb{R}^{n \times m}$) introducing extra degrees of freedom. In those cases one can calculate $\mathrm{J}_p f$ in a second step after choosing charts from $\mathrm{J}_p \tilde{\mathbf{f}}$ by exploiting,

$$\varphi_{\mathscr{N}}^{-1} \circ f \circ \varphi_{\mathscr{M}}|_U = \vartheta \circ \tilde{\mathbf{f}} \circ \varphi_{\mathscr{M}}|_U \tag{2}$$

$$\Longrightarrow \mathrm{J}_p f = \mathrm{J}_{f(p)} \vartheta \cdot \mathrm{J}_p \tilde{\mathbf{f}} \cdot \mathrm{J}_{\hat{\mathbf{p}}} \varphi_{\mathscr{M}}, \tag{3}$$

for an open set $U \subset \mathrm{dom}(\varphi_{\mathscr{M}}) \subset \mathbb{R}^m$ containing $\hat{\mathbf{p}}$, small enough, and a differentiable $\vartheta : \tilde{\mathbf{f}}(\varphi_{\mathscr{M}}(U)) \to \mathbb{R}^n$, such that $\vartheta|_{\mathrm{dom}(\varphi_{\mathscr{N}}^{-1}) \cap \mathrm{dom}(\vartheta)} = \varphi_{\mathscr{N}}^{-1}|_{\mathrm{dom}(\varphi_{\mathscr{N}}^{-1}) \cap \mathrm{dom}(\vartheta)}$.

Even though this introduces an unnecessary step in the algorithm, it is precisely the path suggested by a scalar-based AD. Because the manifolds are represented as embedded sub-manifolds and thus the algorithm evaluating f maps, in practice, elements of $\mathscr{C}_{\mathscr{M}}$ to elements of $\mathscr{C}_{\mathscr{N}}$. Hence, applying a scalar-based AD approach directly on this evaluating algorithm will calculate a global Jacobian, $\mathrm{J}_p\tilde{\mathbf{f}}$, typically concealing the fact that a choice of $\tilde{\mathbf{f}}$ does happen in this step.

The missing two Jacobians in (3) can either be calculated using scalar-based AD or by manually supplied algorithms. The latter is exactly the concept behind the current Ceres implementations when one uses its *local parametrization*, which plays here the role of the chosen chart $\varphi_{\mathscr{M}}$. As Ceres does not currently support manifold-valued error terms it can only be used in cases where $\mathscr{N}$ is a vector space and thus ϑ can be chosen trivial.

2.3 Block Automatic Differentiation

This section provides an overview of the BAD concept and a simple worked example showing the magnitude of speedup that is possible.

2.3.1 Overview

The motivation behind the development of BAD is to be able to inject knowledge from the specific structure of a DMs into the AD process in order to speed up the calculation of Jacobian matrices. The speedups gained by this approach will be specific to each manifold.

To enable an AD library to do that, it needs to know which DMs are involved in a mapping $f : \mathscr{M} \to \mathscr{N}$ that should be differentiated. To achieve that, the primary idea is to consider a computation graph on a mathematically higher level. Instead of primitive scalar operations $(+, *, \ldots)$ on a single scalar type, we consider a set of basic operations that operate on points in manifolds. For example, this encompasses the usual vector and matrix operations, but also geometric operations like exponential maps, or special operations like rotation of a 3D point by a unit-length quaternion.

In practice most differentiable functions of interest between DMs can be deconstructed into a computation graph compounding such basic operations. These are the computational atoms that we think in when building up mathematical models and, in robotics, there is a surprisingly small set of such basic operations needed to implement many fundamental algorithms.

Such a high level computation graph can then be grouped by a BAD library into sub-blocks for which there is optimized (Jacobian) evaluation code. This optimized

code can be either be manually written or the output of an code generator of a symbolic toolkit.

In this way, the BAD concept is a mixture of manual, symbolic, and automatic differentiation, allowing a series of novel opportunities to optimize the computational complexity:

- to derive a suitable intermediate value-cache structure for the evaluation of the compound function and Jacobian evaluation;
- to automatically apply mathematically simplified algorithms for whole compound functions;
- to choose, based on the expression, the Jacobian evaluation direction or even mixtures of forward and backward steps; and
- to use highly optimized matrix manipulation libraries to do the final calculations.

However, there is one important obvious drawback when compared to AD: there are many more combinations of manifolds and basic operations on them than scalar operations. While it is easily possible to make a scalar AD library complete, a BAD library will never be. Because of this it is very important for a BAD library to be user extensible and to grow over time, eventually reverting to scalar AD as a last resort. To address this, our implementation efforts have been focused on building up a core framework that tries to make it as easy as possible to add support for manifolds and operations.

Ultimately, a BAD library will be less optimal than the output of an ideal symbolic tool optimizing the whole function f. Nevertheless, current symbolic tools have many important drawbacks compared to the BAD concept:

- the work flow from the mathematical model to the running code involves extra steps (i.e. code generation during the compilation phase) that may take much more processing time, especially when the expressions get quite complex;
- they are bad in factoring out repeated blocks to functions and thus produce huge code that is impossible to read and hard to maintain;
- they don't allow dynamic (at run-time) construction of the function f, which can be essential for special problems; and
- they usually do not incorporate matrix manipulation libraries and thus neglect a very important intermediate level of optimization.

2.3.2 Example

Here we provide the simple example of the rotation of a 3×1 vector, $\mathbf{v}$, by a rotation, $\mathbf{C}$, represented by a unit-length quaternion, q, to illustrate the potential of mathematical simplification of the Jacobian evaluation by exploiting knowledge about the underlying structures. Let v denote the pure imaginary quaternion corresponding to $\mathbf{v}$. The function we will analyze can then be defined as,

$$\mathbf{r} : S^0(\mathbb{H}) \to \mathbb{R}^3, \;\; q \mapsto \mathrm{Im}(qv\bar{q}), \tag{4}$$

where multiplication of non-bold quantities is quaternion multiplication, the overbar denotes the quaternion conjugate operation, and $\mathrm{Im}(\cdot)$ extracts the imaginary components of the quaternion as a 3×1 vector. Here, $\mathbf{r}(\cdot)$ is only defined on the unit-length quaternions.

To be able to write an algorithm that evaluates this function or its Jacobian, we have to define how to represent the involved quantities. We will represent q with a 4-tuple (q_0, q_1, q_2, q_3) identified with q's coordinates in default ordered basis of $\mathbb{H}$, $(1, i, j, k)$. $\mathbf{v}$ and $\mathbf{r}(q)$'s value will be represented by the usual 3-tuples. To evaluate (4), the scalar calculations shown in Listing 1 can be executed (taken directly from the Ceres source code). (see Figure 1 for the equivalent computation graph).

Listing 1 Algorithm evaluating $\mathbf{r}(q)$ defined in eq. (4)

```
t1=-q3*q3; t2= q0*q1; t3= q0*q2; t4= q0*q3; t5=-q1*q1;
t6= q1*q2; t7= q1*q3; t8=-q2*q2; t9= q2*q3;
r0=2*((t8+t1)*v0+(t6-t4)*v1+(t3+t7)*v2)+v0;
r1=2*((t4+t6)*v0+(t5+t1)*v1+(t9-t2)*v2)+v1;
r2=2*((t7-t3)*v0+(t2+t9)*v1+(t5+t8)*v2)+v2;
```

These are $21m + 18a$ Floating Point Operation (FLOP)s for a single evaluation[3].

To calculate the local Jacobian matrix $\mathrm{J}_q\mathbf{r}$ at an arbitrary $q \in S^0(\mathbb{H})$ using a dual-number approach, we follow the method utilizing the outer vector space encompassed by (3)). To start we analyze the complexity of the first step in which one would calculate the Jacobian of $\tilde{\mathbf{r}} : \mathbb{H} \to \mathbb{R}^3$, defined by the pseudo code in Listing 1 (corresponding to $\tilde{\mathbf{f}}$ in Section 2.2). Using the dual-number approach, it is necessary to propagate four extra variables through each operation (one partial derivative per component of (q_0, q_1, q_2, q_3)) (see 2.1). For each extra variable, a multiplication in the original code requires an extra $2m + 1a$ and each addition requires an extra $1a$. This results in $21(2m + 1a) + 18a = 42m + 39a$ FLOPs per variable and $4(42m + 39a) = 168m + 156a$ for the full $4x3$ global Jacobian matrix. To build the local $\mathrm{J}_q\mathbf{r}$ we have to choose a chart around q. We will use the common exponential chart $\varphi_q : U \subset \mathbb{R}^3 \to S^0(\mathbb{H}), \mathbf{w} \mapsto \exp(w)q$, where w denotes the pure imaginary quaternion with vector part $\mathbf{w}$ and U is an environment of $\mathbf{0}$ small enough to make the map injective. Its Jacobian at $\mathbf{w} = \mathbf{0}$ can be calculated from q with negations only. The remaining step is to multiply these matrices, making the total cost $9(4m + 3a) = 36m + 27a$, assuming a straightforward implementation.

Our approach would exploit the following simplification of the general directional derivative in direction $\mathbf{w} \in \mathbb{R}^3$ to calculate $\mathrm{J}_q\mathbf{r} = \mathrm{J}_\mathbf{0}(\mathbf{r} \circ \phi_q)$.

[3] Here, m represents multiplication operations and a represents addition operations. We ignore negations.

$$\begin{aligned}
\mathrm{J}_{\mathbf{0}}(\mathbf{r}\circ\varphi_q)\cdot\mathbf{w} &= (\mathrm{d}_q\mathrm{Im}(qv\bar{q}))(wq) && \text{(5a)}\\
&= \mathrm{Im}(wqv\bar{q}+qv\overline{wq}) && \text{(5b)}\\
&= \mathrm{Im}(wqv\bar{q}+qv\bar{q}\bar{w}) && \text{(5c)}\\
&= \mathrm{Im}(wqv\bar{q}-qv\bar{q}w) && \text{(5d)}\\
&= \mathrm{Im}([w,qv\bar{q}]) && \text{(5e)}\\
&= 2\mathbf{w}\times\mathrm{Im}(qv\bar{q}) && \text{(5f)}\\
&= 2\mathbf{w}\times\mathbf{r}(q) && \text{(5g)}
\end{aligned}$$

In the above, $[\cdot,\cdot]$ is the commutator of quaternions and $\times$ represents the cross product of 3×1 vectors. Note that the equations above just sketch the proof, sometimes taking advantage of concepts requiring more in-depth knowledge of DM and specifically of $\mathbb{H}$. For the sake of brevity we will stay at this level of details here.

To calculate the columns of $\mathbf{r}$'s Jacobian matrix, one would evaluate the last expression for w substituted with each default basis vector of $\mathbb{R}^3$. In these evaluations, the cross product becomes trivial (i.e. only requiring negations), reducing the FLOP count to $3m$ needed to scale the components by a factor 2.

The example above shows how we can go from $204m+183a$ FLOPs to $3m$ FLOPs to calculate the Jacobian by choosing a beneficial local chart and injecting knowledge about the underlying DM and the outer space at the block level. Any manifold has the potential to benefit from this method based on its specific structure, or by choosing advantageous embeddings or tangent space basis vectors. For unit-length quaternions representing rotations, one would implement multiplication and inversion, along with the log and exponential map, resulting in a full-featured BAD system for manipulation of expressions for this specific DM. For any other manifold of interest, one would follow the same procedure, writing down the operations to be supported, deriving some efficient analytical expressions for the Jacobians induced by these operations, and implementing these as operations supported by the library.

3 Implementation

This section describes the usage of our implementation and compares it to the AD package provided by the Ceres optimizer.

Using the Ceres AutoDiffCostFunction to calculate the local Jacobian of the expression $\mathbf{r}(q)$, one would write the C++ code presented in Listing 2. Please note the necessary extra step on line 27-29 to convert the Jacobian in global coordinates (of the embedding space) to the Jacobian in local parametrization.

Listing 2 Calculating the local Jacobian of the function $\mathbf{r}(\cdot)$ with Ceres AutoDiffCostFunction

```
1  using namespace ceres;
2  struct Cv {
3    const double *v;
4    Cv(const double *v) : v(v) {}
5
6    template <typename T>
7    bool operator()(const T* const q, T* residuals) const
8    {
9      UnitQuaternionRotatePoint(q, v, residuals);
10     return true;
11   }
12 };
13
14 QuaternionParameterization quaternionParameterization;
15
16 void calcJacobian(const double *q, const double *v,
17     double *result, double *qJ)
18 {
19   const double *parameters[] = {q};
20   double qGlobalJ[3 * 4];
21   double *jacobians[] = {qGlobalJ};
22   double qLocalParamJ[4 * 3];
23
24   AutoDiffCostFunction<Cv,3,4> r(new Cv(v));
25
26   r.Evaluate(parameters, result, jacobians);
27
28   quaternionParameterization.ComputeJacobian(q, qLocalParamJ);
29   internal::MatrixMatrixMultiply<3, 4, 4, 3, 0>(
30     qLocalParamJ, 3, 4, localParamJ, 4, 3, qJ, 0, 0,  3, 3);
31 }
```

Listing 3 illustrates how our approach could be used to solve the same problem. In this example, we use the "auto" keyword from C++ 2011 to simplify the code and let the compiler deduce the type of an expression[4].

In line 6, the values pQ is converted into something one can later take a derivative with respect to (=Diffable). The Ref template only enforces capturing per reference (as also in line 7 with v an pV).
In line 9, the high-level computation graph converted to a type and becomes (thanks to *auto*) the type of r. The data of r will only contain the references to pQ and pV.
In line 11, an intermediate derived-value cache will be created. Its type will depend on r's type and essentially include storage to store intermediate values that could be needed repeatedly in the (Jacobian) evaluation (e.g. the conjugate of q or its rotation matrix counterpart). The resulting storage is embedded per reference into the cache, to have a unified storage for all values, without the need to copy the result data back.
In line 13, intermediate values that are needed will be calculated (e.g. always the final evaluation result — here the rotated vector). In line 14, the Jacobian is computed directly into the provided storage qJ.

[4] In fact without "auto" the whole concept would become quite cumbersome to use because the type names quickly become huge and unreadable. To allow the use of the library without knowing about these generated types is one of the big implementation challenges.

Listing 3 Using the BAD to calculate the local Jacobian of the function $\mathbf{r}(\cdot)$.

```
1  using namespace tex; using namespace Eigen;
2  void calcJacobian(const UnitQuaternion & pQ,
3      const EuclideanPoint<3> & pV,
4      EuclideanPoint<3> & result, Matrix3d & qJ)
5  {
6    Diffable<Ref<UnitQuaternion>, 0> q(pQ);
7    Ref<EuclideanPoint<3>> v(pV);
8
9    auto r = q.rotate(v);
10
11   auto cache = createCache(r, result);
12
13   cache.update(r);
14   evalFullDiffInto(r, q, cache, qJ);
15 }
```

Because the mathematical types of all variables and operations in the expression are encoded in their C++ types, it is possible to do the following:

- to derive a suitable intermediate and derived value cache structure for the evaluation of this expression and especially its Jacobian;
- to automatically apply mathematically simplified algorithms (using template specialization and overload resolution to look them up) to evaluate the expression and especially its derivatives using this cache structure;
- to implement the Jacobian evaluation using (block) forward or backward evaluation (or some mixture of the two), whatever is the fastest for this particular fragment of the computation graph.

4 Experiments

In this section we describe two experiments comparing our BAD prototype implementation with the dual-number approach implemented by Ceres. In the first experiment, we compare the timing of Jacobian evaluations on increasingly large problems. In the second experiment, we use our approach on real-world data in a nonlinear optimization problem whose goal is to estimate the time-varying orientation of an elephant seal in a wildlife monitoring context.

4.1 Comparison with state-of-the-art Automatic Differentiation

In the first experiment, we compared the AD implementation of Ceres to our prototype BAD implementation. To measure performance on an increasingly complex problem, we measured the time required to calculate the value and the Jacobian of expressions of the form,

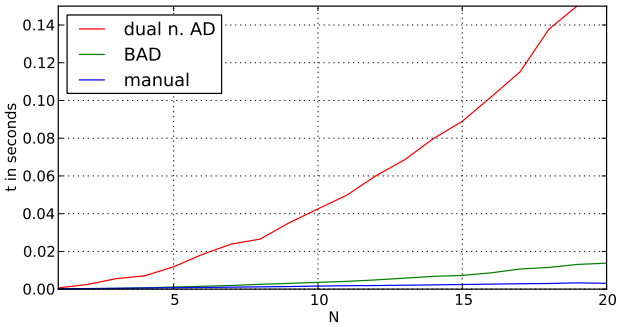

Fig. 2 Performance comparison for evaluating the value and Jacobian of $\prod_{i=1}^{N} \mathbf{C}_i\mathbf{v}$, as a function of N. The red line (dual n. AD) refers to the dual-number approach implemented by Ceres, the green line corresponds to BAD and the blue line is a hand-tuned version of the evaluation.

$$\begin{array}{l} \mathbf{C}_1\mathbf{v} \\ \mathbf{C}_2\mathbf{C}_1\mathbf{v} \qquad \text{for } N \in 1\ldots20 \\ \mathbf{C}_N\ldots\mathbf{C}_1\mathbf{v}, \end{array} \tag{6}$$

where $\mathbf{C}_i$ is a rotation (represented in our case by a unit-length quaternion), and $\mathbf{v}$ is a 3×1 vector. The time required by Ceres (denoted as dual-number AD) and by our approach, as a function of the problem size, is shown in Figure 2. On these artificial problems, there is an obvious benefit for using the specific structure of the DM to compute the Jacobian. However, we also compared it with a hand-tuned implementation of the Jacobian calculation. This results in the lowest curve (blue) in Figure 2. Numerically, on this specific example, we find that BAD is 12 times faster than Ceres, but still 4 time slower than the hand-tuned evaluation, independently of the problem size.

4.2 Application: Elephant-Seal Attitude Estimation

In this section, we will compare the performance of the different optimization solutions on a specific application: the estimation of the attitude of elephant seals based on accelerometer and magnetometer recording collected over weeks or months with sensors attached to the animals while they are at sea. Although the data-set is peculiar and might seem far outside the field of robotics, the problem of batch attitude or position estimation from initial measurement unit is a common issue for underwater and flying robots.

The sensors attached to the animals record the following data at 1Hz: depth, acceleration, magnetic field. A sound-based system estimates the animal linear velocity but due to energy saving considerations, this is switched on only for 3 hours every 12 hours. At the surface, global positioning is retrieved from the ARGOS

satellites. To illustrate the point of this paper, we will focus on using the accelerometer and the magnetometer to retrieve the animal attitude. In a later stage, this could be used in combination with the velocity measurement to estimate a 3D trajectory, or in combination with GPS data to estimate sea currents. In a real situation, it is also necessary to use the batch estimation process to estimate some sensor calibration parameters. All these extensions will be kept out of scope for this paper.

4.2.1 Problem Statement

Formally, the state we are estimating is the seal attitude as a rotation matrix C_t at time t and the propulsion force P_t it applies along the x axis in its body frame. To this end, we have the measurements of two constant vectors in the world frame: the acceleration $G = [0,0,9.81]^T$ and the magnetic field $B = [B_x, B_y, B_z]^T$. The exact value of the magnetic field can be found using the IGRF[5] magnetic field model, which depends on the time, the latitude and the longitude. For sake of simplicity, we will assume here that the reference magnetic field is constant between two GPS fix. We will denote b_t the magnetic field reported by the sensor in the body frame, and a_t the accelerometer output. Note that the measured acceleration results from the combination of gravity and the propulsion force $P_t \cdot \mathbf{x}$ applied by the seal, where $\mathbf{x}$ is the longitudinal axis of the animal.

With these variables we can build the following error terms for the accelerometer and magnetometer:

$$F_{acc}(t) = C_t \cdot [a_t - P_t \cdot \mathbf{x}] - G \text{ and } F_{mag}(t) = C_t \cdot b_t - B \tag{7}$$

In addition, we can make some continuity hypothesis on the attitude and the propulsion and express them as the following error terms:

$$F_p(t) = P_t - P_{t-1} \text{ and } F_C(t) = C_t \cdot C_{t-1} \tag{8}$$

The initial values used for the optimizer is propulsion at zero and Euler angles (roll, pitch and yaw) estimated independently from the accelerometer and magnetometer.

4.2.2 Comparison of optimization performance

Figure 3(a) and 3(b) shows the time, t, required to calculate the Jacobians necessary to solve the simplified seals optimization problem using single-threaded execution on a benchmark PC after loading the first N input lines of the measured data. In Figure 3(a), we measure the time (using the x86 RDTSC instruction) in total spent in the Jacobian calculation function per error term. In Figure 3(b), we show the time measured by the Ceres optimizer for the whole Jacobian evaluation step (with the

[5] http://www.mathworks.fr/matlabcentral/fileexchange/28874-igrf-magnetic-field

clock used in Figure 3(a) disabled). The solid line represents our implementation of BAD and the dashed one shows Ceres implementation of the dual-number approach. Both show the performance improvement by BAD. In Figure 3(a), BAD is approximately 2.5-times faster, in Figure 3(b) it is only about 13.5%. The explanation for this example is that Ceres spends a relatively large amount of time constructing the overall Jacobian for the complete optimization problem from the Jacobians of the individual error term, which are rather simple error terms. Ceres includes the full sparse Jacobian matrix construction in what it reports as the Jacobian calculation time.

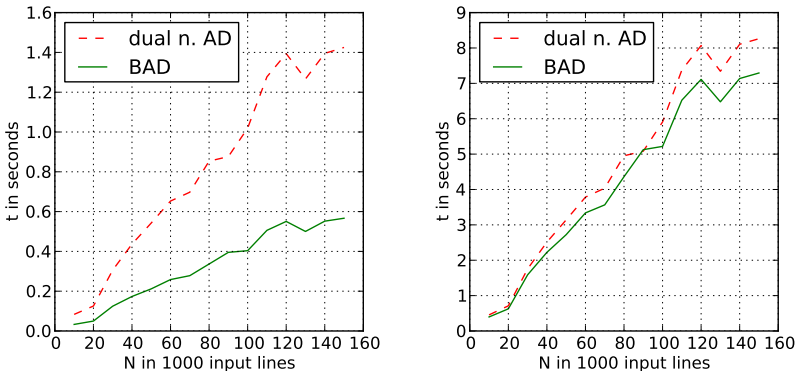

Fig. 3 Time required to compute the Jacobian as a function of the number of error terms for the simplified seal problem. On the left, only Jacobian computation time is included, on the right, the time for the creation of the full sparse Jacobian matrix is also included. See text for details.

To be complete, we must state here that the comparison here is slightly unfair in favor of Ceres for two reasons. Ceres expects global Jacobians (the natural output of a scalar AD) instead of the local Jacobians that they later calculate from these global ones by multiplying by the Jacobian of the local parametrization (see (3)). To fit with this requirement, we calculate $n \times 4$ matrices from our $n \times 3$ Jacobian and Ceres then calculates the $n \times 3$ back (n is the dimension of the error term). Each of these conversions requires a matrix multiplication (first 3×4 then 4×3). The first multiplication delays our implementation in Figure 3. It would be fairer to skip these artificial steps but it would require changes in the Ceres code, which would further improve the performance of BAD in Figure 3(b). In addition, Ceres uses non-aligned Eigen::Vectors, while our implementation needs aligned ones, which requires us to copy the data into aligned vectors. The time spent copying these vectors is also counted in the data depicted in Figure 3.

5 Conclusions

In this paper we have brought together the concepts of AD and DM to develop a block AD approach that we cal BAD. This approach has the potential to be much more computationally efficient than traditional scalar AD by exploiting the specific structure of the DM. We presented a worked example of a unit-length quaternion rotating a 3×1 vector and showed how it can greatly reduce the number of instructions needed for Jacobian computation. Finally, we presented experimental results on simulated and real data that show that our prototype implementation of BAD is indeed faster than a state-of-the-art AD approach.

Our next step will be to finalize the interface and implement a full-featured BAD library in C++ encompassing the most common DMs and operations needed in robotics.

Acknowledgements Cédric Pradalier is supported by the European Commission within the Noptilus project, EC Grant agreement 270180. The elephant-seal data-set was kindly provided by Ch. Guinet and B. Picard from the CNRS Center for Biological Study in Chizé, France, based on data collected on the Kerguelen Islands, South Indian Ocean.

References

1. Absil, P.A., Mahony, R., Sepulchre, R.: Optimization algorithms on matrix manifolds. Princeton University Press (2009)
2. Agarwal, S., Mierle, K., Others: Ceres solver (2013). https://code.google.com/p/ceres-solver/
3. Barfoot, T.D., Forbes, J.R., Furgale, P.T.: Pose estimation using linearized rotations and quaternion algebra. Acta Astronautica **68**(1–2), 101–112 (2011). DOI 10.1016/j.actaastro.2010.06.049
4. Gwak, S., Kim, J., Park, F.C.: Numerical optimization on the euclidean group with applications to camera calibration. Robotics and Automation, IEEE Transactions on **19**(1), 65–74 (2003)
5. Hertzberg, C., Wagner, R., Frese, U., Schrder, L.: Integrating generic sensor fusion algorithms with sound state representations through encapsulation of manifolds. Information Fusion **14**(1), 57 – 77 (2013). DOI http://dx.doi.org/10.1016/j.inffus.2011.08.003. URL http://www.sciencedirect.com/science/article/pii/S1566253511000571
6. Lambert, Furgale, Barfoot, Enright: Field testing of visual odometry aided by a sun sensor and inclinometer. Journal of Field Robotics **29**(3), 426–444 (2012). DOI 10.1002/rob.21412
7. Martins, J.R., Sturdza, P., Alonso, J.J.: The complex-step derivative approximation. ACM Transactions on Mathematical Software (TOMS) **29**(3), 245–262 (2003)
8. Murray, R.M., Sastry, S.S.: A mathematical introduction to robotic manipulation. CRC press (1994)
9. Piponi, D.: Automatic differentiation, c++ templates, and photogrammetry. Journal of Graphics Tools **9**(4), 41–55 (2004)
10. Rall, L.B.: Perspectives on automatic differentiation: Past, present, and future? In: Automatic Differentiation: Applications, Theory, and Implementations, pp. 1–14. Springer (2006)
11. Rall, L.B., Corliss, G.F.: An introduction to automatic differentiation. In: M. Berz, C.H. Bischof, G.F. Corliss, A. Griewank (eds.) Computational Differentiation: Techniques, Applications, and Tools, pp. 1–17. SIAM, Philadelphia, PA (1996)
12. Shuster, M.D.: A survey of attitude representations. Journal of the Astronautical Sciences **41**(4), 439–517 (1993)