Eigen-AD: Algorithmic Differentiation of the Eigen Library

Patrick Peltzer, Johannes Lotz, Uwe Naumann *

Abstract

In this work we present useful techniques and possible enhancements when applying an Algorithmic Differentiation (AD) tool to the linear algebra library Eigen using our in-house AD by overloading (AD-O) tool dco/c++ as a case study. After outlining performance and feasibility issues when calculating derivatives for the official Eigen release, we propose Eigen-AD, which enables different optimization options for an AD-O tool by providing add-on modules for Eigen. The range of features includes a better handling of expression templates for general performance improvements as well as implementations of symbolically derived expressions for calculating derivatives of certain core operations. The software design allows an AD-O tool to provide specializations to automatically include symbolic operations and thereby keep the look and feel of plain AD by overloading. As a showcase, dco/c++ is provided with such a module and its significant performance improvements are validated by benchmarks.

1 Introduction

Linear algebra operations are often implemented as dedicated software libraries, which are then applied by the user to his or her specific use case. In this work, the popular C++ library Eigen is used as a base software implementing linear algebra operations for which derivatives are to be computed using Algorithmic Differentiation (AD) by overloading. Derivatives of computer programs can be of interest, e.g. when performing uncertainty quantification, sensitivity analysis or shape optimization. AD enables the computation of derivatives of the output of such programs with respect to their inputs. This is done using the tangent model or the adjoint model, where the latter is also known as adjoint AD (AAD). In AAD, the program is first executed in the augmented primal run, where required data for later use is stored. Derivative information is then propagated through the tape in the adjoint run. For both, tangent and adjoint, the underlying original code is called the primal, and the used floating point data type and its variables are called passive. Vice versa, code where derivatives are computed and its respective data type and variables are called active.

On the topic of combining linear algebra and AD, several works have looked at different approaches so far. Besides utilizing AD for computing functions relevant for linear algebra, efforts have been made for gaining insight into AD applied to linear algebra computations, e.g. for scalar-matrix functions. Derivatives, e.g. for matrix-matrix multiplications and inversions, have been determined and reformulated for usage within AD. Regarding deep learning, linear algebra primitives like the Cholesky decomposition were incorporated as differentiable operators into the deep learning development system MXNet.

*Informatik 12: Software and Tools for Computational Engineering, RWTH Aachen University, 52056 Aachen, Germany; Email: info@stce.rwth-aachen.de

http://eigen.tuxfamily.org
A wide collection of AD tools can be found on the community website\(^2\). In general, one can divide the available software into *source transformation* and *operator overloading* tools. While source transformation essentially has the potential to create more efficient code, supporting complex language features like they are available in C++ is connected to higher expense for the tool authors. AD by overloading (AD-O) on the other hand can be applied to arbitrary code as long as operator overloading is supported by the programming language. In terms of AD-O, the recorded data in the augmented primal run is referred to as the *tape*, and creating the tape is called *taping*. The propagation in the adjoint run is known as *interpreting* the tape.

Applying an AD-O tool to dedicated libraries poses a significant issue, as by principle the floating point type must be changed to a custom data type provided by the AD-O tool (from now referred to as the *custom AD data type*). This switch in data type is often impractical and breaks hand tuned performance gains\(^1\). Therefore, the goal of a software combining AD-O and linear algebra has been realized with Adept\(^\text{11}\). Another approach is implemented in the *Stan Math Library*, which is an AD-O tool evaluating linear algebra operations by utilizing Eigen\(^\text{12}\). Since the Stan Math Library uses similar techniques as presented in this work, more details on it will be given in section 4.2.1.

Eigen allows the direct utilization of AD-O tools due to its extensive use of C++ templates. It implements a wide range of features on a relatively high abstraction level, which makes it attractive for a broad field of applications. At the same time, it allows the use of lower level library implementations of LAPACK and BLAS. Further performance improvements result from vectorization support. At a later point in this paper, concrete implementations and benchmarks will be presented using the AD-O tool *dco/c++*, which is actively developed by NAG Ltd\(^\text{3}\) in collaboration with RWTH Aachen University.

To our best knowledge, there has not been a work focusing on the application of an AD-O tool to Eigen while preserving the philosophy of plain AD-O. The goal is that the AD-O tool user benefits from optimizations without explicitly being aware of them. Swapping the data type of Eigen and using the AD-O tool as usual should be all that is required to compute derivatives. However, several problems concerning performance and feasibility of the derivative computation will arise from this concept. This work proposes approaches and solutions to overcome them.

The next section provides more background on AD-O and also introduces the concept of symbolic derivatives. In Chapter 2\(^2\) already possible applications of AD to Eigen are presented and problems that may occur are outlined. Chapter 3\(^4\) is the main part of this paper and proposes *Eigen-AD*, which was produced during the course of this work. It contains several optimizations and improvements for the application of an AD-O tool, which are demonstrated and benchmarked using *dco/c++* in Chapter 4\(^4\). Chapter 5\(^5\) summarizes the results and suggests possible future works.

### 1.1 Using AD-O and Symbolic Derivatives

Some of the performance improvements presented at a later point in this paper are based on symbolic differentiation (SD), in which derivatives are evaluated analytically at a higher level than with AD. This section demonstrates the differences between evaluating derivatives symbolically and with AD-O by using the matrix-matrix product \(C = AB\) with \(A, B \in \mathbb{R}^{2 \times 2}\) as an example.

\(\text{http://www.autodiff.org/}\)
\(\text{https://www.nag.co.uk/}\)
Let this specific product kernel be implemented using Eigen as follows:

```cpp
1 template<typename T>
2 void matmul(const Matrix<T,2,2>& A, const Matrix<T,2,2>& B, Matrix<T,2,2>& C) {
3     for (int i = 0; i < 2; i++)
4         for (int j = 0; j < 2; j++)
5             for (int k = 0; k < 2; k++)
6                 C(i, j) += A(i, k) * B(k, j);
7 }
```

Listing 1: 2 × 2 matrix-matrix multiplication kernel.

The primal code is called using the passive data type `double` as template argument `T`. For an active evaluation, the function must be called using the custom AD data type of an AD-O tool as `T`. As mentioned in the previous section, the AD-O tool first performs the augmented primal run when in adjoint mode. Both, the `+=` and the `*` operators in line 6, are overloaded by the tool and act as the entry points for taping. The tape is an implementation dependent representation of the computational graph of the program, which contains all performed computations and their corresponding partial derivatives. Figure 1 displays the computational graph of the matrix-matrix multiplication kernel in Listing 1. Vertices represent variables accessed in the augmented primal run, including temporary instantiations from the `*` operator in line 6 (denoted as `z`). The edge weights are the partial derivatives of the respective computations. In the adjoint run, the graph is traversed in reverse order, propagating the adjoint value of the output towards the inputs. This is done by multiplying subsequent edge weights and adding parallel edge weights. Effectively, the loops of Listing 1 are executed in reverse order; derivatives are computed on scalar level.

In contrast to the differentiation of all occurring scalar computations, it may also be possible to rewrite the derivative using matrix expressions so that derivatives are computed on matrix level. Staying with the example above, the adjoint propagation on the computational graph in Figure 1 can be written as follows:

\[
\begin{align*}
\bar{A}_{0,0} &= \bar{C}_{0,0}B_{0,0} + \bar{C}_{0,1}B_{0,1} \\
\bar{A}_{0,1} &= \bar{C}_{0,0}B_{1,0} + \bar{C}_{0,1}B_{1,1} \\
\bar{A}_{1,0} &= \bar{C}_{1,0}B_{0,0} + \bar{C}_{1,1}B_{0,1} \\
\bar{A}_{1,1} &= \bar{C}_{1,0}B_{1,0} + \bar{C}_{1,1}B_{1,1}
\end{align*}
\]

\[
\Rightarrow \bar{A} = \bar{C}B^T \quad (1.1)
\]

\[
\begin{align*}
\bar{B}_{0,0} &= \bar{C}_{0,0}A_{0,0} + \bar{C}_{1,0}A_{1,0} \\
\bar{B}_{0,1} &= \bar{C}_{0,0}A_{0,1} + \bar{C}_{1,1}A_{1,0} \\
\bar{B}_{1,0} &= \bar{C}_{0,0}A_{0,1} + \bar{C}_{1,1}A_{1,1} \\
\bar{B}_{1,1} &= \bar{C}_{0,1}A_{0,1} + \bar{C}_{1,1}A_{1,1}
\end{align*}
\]

\[
\Rightarrow \bar{B} = A^T\bar{C} \quad (1.2)
\]

Adjoint values are denoted with a bar. Equations (1.1) and (1.2) compute the adjoints of the input data using matrix-matrix multiplications. Using these equations, it is not necessary to tape any computations in Listing 1. Instead, the adjoints can directly be computed in the adjoint run on matrix level as long as the values of the input data are available. Further details on using such SD equations and on their advantages are given in Section 3.2.
2 AD-O in Eigen

Eigen comes with its own tangent type called *AutoDiff*. This type can be used as a wrapper around existing Eigen types for evaluating derivatives. An adjoint type is not available.

However, due to Eigen’s versatile nature, AD-O can be applied to it with any AD-O tool almost straight away. A custom scalar type $T$ is supported by Eigen if the following conditions are fulfilled: $T$ offers all required operator overloads; all required math functions are defined for $T$ and a specialization for `Eigen::NumTraits<T>` exists. The first two points can be assumed to be satisfied by an AD-O tool. The `NumTraits` struct containing compile time information of the custom AD data type can easily be implemented by an AD-O tool author. Therefore, an AD-O tool can – in principle – directly be used to compute adjoint derivatives of Eigen.

However, applying an AD-O tool to Eigen will lead to severe limitations sooner or later. Due to the use of the custom AD data type, serious run time slow downs can be expected. This is due to the fact that Eigen comes with optimized kernels, e.g. for 8-byte double precision data. Traditionally, custom AD data types are larger, since they either contain tangent information or virtual address information in the adjoint mode. Hence, the optimizations do not work anymore. Regarding AAD application, it is important to note that the complexity of many frequently used linear algebra operations scales cubically with the input dimension. This is the case for, e.g. matrix decompositions or matrix products. Since the memory required by the tape scales roughly linearly with the number of operations required by an algorithm, the tape size can quickly exceed the amount of available RAM and therefore makes an evaluation of the

---

https://eigen.tuxfamily.org/dox/unsupported/group__AutoDiff__Module.html
derivatives not feasible at all. Furthermore, writing the tape introduces an additional run time overhead due to the continuous memory access.

In the light of the above problems, this work proposes several improvements to make AD-O feasible and faster for use with Eigen. The resulting Eigen-AD software is presented in the next chapter.

3 Eigen-AD

Although Eigen provides options to change its behaviour via plug-ins or C++ specializations without modifying its source code, these are rather limited and not sufficient to realize satisfying results. Therefore, the Eigen source code was adjusted to help optimize the application of AD-O tools. The resulting software has been named Eigen-AD.

All source code changes are generically written and do not modify the original Eigen API, but provide entry points which can be used by additional unsupported modules. Based on that, we have added a generic Eigen-AD base module which provides a clean interface for developers to control and implement optimized operations in their tool specific AD-O tool module. Refer to Figure 2 for the package architecture. The Eigen-AD API is documented in the Eigen-AD base module user guide, but important aspects will also be presented in this paper at corresponding points. Actual implementations and here discussed optimizations are dco/c++ specific and are bundled in the dco/c++ module, which should be included by the user when using dco/c++. Similar implementations are possible for other AD-O tools as well.

The existing Eigen test system has been extended so that every Eigen test can also be performed for an AD-O tool's tangent and adjoint data types. Although the types are used passively inside the test, i.e. no derivatives are computed, compiling and running the tests successfully is a solid hint that computing derivatives should also be possible. The philosophy is that an AD-O tool is able to determine derivatives of all Eigen operations algorithmically, while selected operations are provided with optimized computations for their derivatives. Another aim is to keep the look and feel of AD-O, i.e. optimizations and improvements shall not require a separate interface but be used automatically whenever the AD-O tool is applied.

While Section 3.1 introduces a general improvement concept for Eigen coupled with AD-O tools, Section 3.2 proposes optimizations for certain Eigen operations. Implementation nuances for the dco/c++ module are presented in Section 3.3 and the improvements are then validated by measurements in Chapter 4. Access to the Eigen-AD fork can be requested from the authors.

Future plans are outlined in Chapter 5.

Figure 2: Eigen-AD package architecture: Authors can implement an AD-O tool module for their AD-O tool.

https://eigen.tuxfamily.org/dox/unsupported/index.html
3.1 Nesting Expression Templates

The concept of expression templates was originally proposed to eliminate temporaries when evaluating vector and matrix expressions and to be able to pass algebraic expressions as arguments to functions \[13\]. While the first aspect, also known as lazy evaluation, was widely seen as the method for accomplishing high performance computations, later works have shown that this approach is only favorable in certain situations. The concept of Smart Expression Templates was derived \[14\]. Eigen follows this modified approach as it can be inferred from its documentation\[6\]: “Eigen has intelligent compile-time mechanisms to enable lazy evaluation and removing temporaries where appropriate”.

In the context of AD, using expression templates is especially relevant, since every temporary contributes to the computational graph as it was also demonstrated in Figure 1 in Section 1.1. In the AAD case, the computational graph needs to be stored in memory and is then traversed in the reverse run, increasing memory and run time requirements with each additional temporary. This can be avoided by constructing expression templates for the right hand side of the assignment and evaluate them altogether. Therefore, some AD-O tools also implement an expression template mechanism, e.g. dco/c++ or Adept \[15, 11\].

When applying such an AD-O tool to Eigen, both expression template engines are nested, where the AD-O tool layer is accessed by the scalar operations of Eigen. This is not an intended use case for Eigen, and therefore Eigen is not aware that it may receive expression template types. The returned expression template is then implicitly casted back to the custom AD data type, resulting in a temporary which must be considered for the derivative evaluation. This destroys the gains originally made by using expression templates in the AD-O tool.

As an example, consider the unary minus operator, implemented in Eigen as a functor named scalar\_opposite\_op as shown in Listing 2. The class template parameter Scalar corresponds to the custom AD data type, which is then used in the functor to specify the in- and output types. An assignment of the form \(A = -B\), where \(A\) and \(B\) are Eigen 1 × 1 matrices containing a single scalar of the adjoint data type will result in a tape structure as displayed in Figure 3. The boxes represent the vertices of the computational graph, where the first entry indicates its unique tape id (virtual address) and the second entry the adjoint value. Analogous to the temporaries \(z\) of the computational graph in Figure 1 an additional tape entry is added due to the unary minus operator.

Expanding this problem to longer expressions with larger matrices, it becomes obvious that creating inner temporaries is not desired and has negative influence on the performance. When looking at the way the Eigen functors are used, it is not necessary to explicitly prescribe what types they return. Due to Eigen’s generic design and as long as the occurring types are compatible – meaning the required casts/specializations/overloads are available – there is no need to force the scalar type at this level.

\[https://eigen.tuxfamily.org/dox/TopicLazyEvaluation.html\]
This is a fitting case to use the C++-14 feature of *auto return type deduction* which allows a function to deduce the return type “from the operand of its return statement using the rules for template argument deduction”[^7]. Therefore, replacing the return type of the functors with the *auto* keyword allows the passing of expression types from the AD-O tool to the Eigen internals. Besides that, it must be ensured that the functors allow arbitrary input types, as they can now be called with expression types as parameters as well. The resulting functor with highlighted changes is shown in Listing 3. Evaluating the same case as above, the tape structure in Figure 4 visualizes that no additional temporary is created.

[^7]: http://en.cppreference.com/w/cpp/language/auto

---

```cpp
1 template<typename Scalar> struct scalar_opposite_op {
2  ... 
3  const Scalar operator() (const Scalar& a) const {
4    return -a; 
5  };

Listing 2: Eigen unary scalar opposite functor implementation.
```

```cpp
1 template<typename Scalar> struct scalar_opposite_op {
2  ... 
3  template<typename T>
4  auto operator() (const T& a) const { return -a; }
5  ... 
6 };

Listing 3: Eigen-AD unary scalar opposite functor implementation with auto return type deduction.
```

This optimization can be applied to all Eigen scalar functors and also to several Eigen math functions which are supported by the AD-O tool’s expression templates. However, there are other things to note which make the implementation slightly more complicated than shown in the above example:

1. While the above mentioned changes adjust the Eigen low level type handling, it is important to know that there are points in Eigen where the correct scalar type must be determined. Since expression types of the AD-O tool are now allowed inside the Eigen internals, they must be changed to the custom AD data type in such situations. When looking at the `Product` class of Eigen which represents the matrix-matrix product, it is obvious that the scalar type of the evaluated expression (i.e. of the resulting matrix) should not be an expression template type but the respective custom AD data type instead. The `Product` class uses the so-called `ScalarBinaryOpTraits` struct to determine the scalar type of the evaluated expression, which by default only allows certain type combinations as input for a binary operation. This struct needs to be specialized by the AD-O tool developer for the corresponding expression template types and prescribe the custom AD data type as the return type.
2. Other Eigen classes like CwiseBinaryOp or CwiseUnaryOp use the std::result_of function to determine the scalar type for their evaluation. Fortunately, these classes use an internal Eigen wrapper for this function call which also needs to be specialized by the AD-O tool developer for the corresponding expression template types. This specialization should either return the custom AD data type directly in the unary case or can simply refer to the ScalarBinaryOpTraits typedef in the binary case.

3. To maintain compatibility with other custom types, the implementation explicitly requires the AD-O tool developer to specify which types should use the auto return type deduction. All other types have the same return type behaviour as before.

3.2 Symbolic Derivatives and Entry Points

As introduced in Section 1.1, mathematical insight can be exploited to evaluate a derivative symbolically. Such an evaluation can be superior to the AD-O solution in terms of performance, run time-wise and also memory-wise in the adjoint case. This observation motivates the inclusion of symbolic derivatives for certain linear algebra routines, yielding a hybrid implementation [16].

The Eigen-AD base module provides an interface for AD-O tool developers to implement symbolic derivatives. At the moment, entry points for products as well as any computation concerning a dense solver are supported. Refer to the Eigen-AD base module user guide for further information. In the next sections, symbolic adjoints for the following operations are introduced:

- Matrix-matrix product
- Solving of a system of linear equations
- Inverse of a matrix
- Log-abs-determinant of a matrix

The necessary Eigen source changes to provide entry points are presented as well as the corresponding Eigen-AD base module API which can be used by an AD-O tool. Actual implementation nuances for the symbolic derivatives using dco/c++ are demonstrated in Section 3.3.

3.2.1 SD Solvers

Consider the system of linear equations:

\[ Ax = b \quad (3.1) \]

where \( A \in \mathbb{R}^{n\times n} \) is the system matrix, \( b \in \mathbb{R}^n \) is the right hand side vector and \( x \in \mathbb{R}^n \) is the solution vector. There exists a wide variety of approaches to solve the problem shown in Equation (3.1) which make use of decomposing the matrix \( A \) into a product of other matrices, exploiting given conditions and special forms. An example is the widely used \( LU \) decomposition. AAD for the solution of a system of linear equation includes the taping and the interpretation of the decomposition, which yields a run time and memory overhead of \( \mathcal{O}(n^3) \). However,
when evaluating the adjoints symbolically using Equations (3.2)-(3.3) as presented in [8], the decomposition is completely excluded from taping and interpreting.

\[ A^T \cdot \bar{b} = \bar{x} \] (3.2)

\[ \bar{A} = -\bar{b} \cdot x^T \] (3.3)

Adjoint values of the right hand side vector \( \bar{b} \) can be determined by solving an additional linear system. By saving the computed decomposition of \( A \) in the augmented primal run, it can then be reused in the adjoint run. This reduces the run time and memory overhead for differentiating to \( O(n^2) \) [17].

Regarding the implementation, it is desired that the symbolic code is embedded smoothly between Eigen and the AD-O tool, i.e. it will automatically be called during the derivative computation without requiring the user to adjust any code. The look and feel of classic AD-O should be preserved while the symbolic implementation introduces a significant performance improvement.

Eigen implements one class for each available dense solver, e.g. FullPivLU or HouseholderQR. It is favorable to specialize each class for the custom AD data type in order to obtain an entry point from where the symbolic evaluation can be initiated, the so-called SD solver. Since the symbolic evaluation in Equations (3.2)-(3.3) is independent from the underlying decomposition, it motivates to implement the symbolic code in a parent class which each SD solver should inherit from. Regarding the existing Eigen class hierarchy, the documentation page specifies that any matrix decomposition \( \text{dec} \) inheriting from \( \text{SolverBase} \) must provide the following API:

1. \( x = \text{dec.solve(b)} \); // solve \( A \cdot x = b \)
2. \( x = \text{dec.transpose().solve(b)} \); // solve \( A^T \cdot x = b \)
3. \( x = \text{dec.adjoint().solve(b)} \); // solve \( A' \cdot x = b \)

Such an API is beneficial since the transposed system needs to be solved in Equation (3.2). The resulting desired class hierarchy can be seen in the class diagram in Figure 5 for two example SD solver specializations. However, the Eigen 3.3 implementation only has two solvers

\[ \text{https://eigen.tuxfamily.org/dox/classEigen_1_1SolverBase.html} \]
Figure 6: New class diagram of all Eigen dense solvers with their respective Eigen module.

inheriting SolverBase since the others do not comply with the above given interface. This has been changed in the Eigen-AD fork, where missing functionalities have been implemented and the solver hierarchy has been fully established, i.e. all solvers now inherit the SolverBase class as can be seen in Figure 6. These changes have been accepted by the Eigen development team and will be included in the upcoming Eigen 3.4 release. Each Eigen solver has also been modified to offer an additional dummy template parameter, which can be used to enable/disable the solver specializations using `enable_if` and the SFINAE technique at compile time. As an entry point, a new `eigen_enable_custom_solver` struct has been added to the Eigen source which is specialized by the Eigen-AD base module.

Implementing the symbolic operations is the task of each individual AD-O tool module. The generic Eigen-AD base module therefore only provides the necessary framework with a SDSolverBase class and the corresponding solver specializations, which only act as wrapper classes and forward relevant operations to the AD-O tool module. By providing a specialization of `eigen_ad_solver_enable` for a respective `_MatrixType`, the AD-O tool module can activate the solver specializations for its custom AD data type. Furthermore, the AD-O tool module should specialize the `eigen_ad_solver` struct for respective `_MatrixType` and `_SolverType` template parameters. The Eigen-AD base module calls several functions in this struct, e.g. `compute(matrix)`, `solve(rhs, dst)` or `inverse(dst)`, which can be defined by the AD-O tool module in order to provide symbolic operations. Additionally, the template parameter `_ValueSolverType` can be used by the AD-O tool module to perform all computations on passive data types (also refer to Section 3.3). By using the special `_WrapperSolverType` template parameter, the Eigen-AD base module also provides default implementations if a function is not defined by an AD-O tool module. As an experimental feature, this default implementation will attempt to compute algorithmic derivatives in order to still provide the full range of solver features. The solver API for AD-O tool modules is displayed in Listing 4.

References:

[1] https://bitbucket.org/eigen/eigen/pull-requests/567
[2] https://en.cppreference.com/w/cpp/types/enable_if
[3] https://en.cppreference.com/w/cpp/language/sfinae
3.2.2 Symbolic Inverse

Inverting a matrix, i.e. computing

$$C = A^{-1}$$ (3.4)

for $A, C \in \mathbb{R}^{n \times n}$ is directly linked to solving a system of linear equations and is implemented in Eigen for a left hand side dst as follows:

```cpp
dst = solver.solve(MatrixType::Identity(src.rows(), src.cols()));
```

where solver is any Eigen solver providing the inverse() function. Note that the right hand side of the solve call is not a vector, but the identity matrix of size $n \times n$. Regarding the SD solvers, the solve(b) implementation expects b to be vector, which disallows the usage of the built-in inverse() function displayed above. Instead, a dedicated symbolic evaluation is implemented using Equation (3.5) [8].

$$\bar{A} = -C^T \bar{C} C^T$$ (3.5)

Compared to AAD, the memory overhead is reduced to $O(n^2)$; however, the adjoint run still has a run time overhead of $O(n^3)$ due to the matrix multiplications.

As stated in Section 3.2.1, an AD-O tool can provide a symbolic implementation by defining the inverse(dst) function of the eigen_ad_solver struct.

3.2.3 Symbolic Log-Abs-Determinant

Another example for an additional symbolic implementation in the SD solver wrapper classes is the computation of $x \in \mathbb{R}$ using the log-abs-determinant of a matrix $A \in \mathbb{R}^{n \times n}$ as shown in Equation (3.6). Such a computation is relevant for, e.g. computing the log-likelihood of a multivariate normal distribution.

$$x = \log |\det(A)|$$ (3.6)

$$\bar{A} = A^{-T} \bar{x}$$ (3.7)
Equation (3.6) is implemented in Eigen for the QR dense solvers, and their respective SD solvers use Equation (3.7) [18] to provide the symbolic derivatives. The inverse can be computed by reusing the decomposition which was kept in memory for the adjoint run (see Section 3.3). While the run time overhead is still $O(n^3)$, the symbolic implementation improves the memory overhead to $O(n^2)$.

An AD-O tool can provide a symbolic implementation by defining the logAbsDeterminant() function of the eigen_ad_solver struct.

### 3.2.4 Symbolic Matrix-Matrix Product

Beside the presented symbolic implementations involving dense solvers, computing derivatives for other Eigen operations can also be improved. For $A \in \mathbb{R}^{n \times m}, B \in \mathbb{R}^{m \times p}, C \in \mathbb{R}^{n \times p}$, computing the adjoints of the matrix-matrix product in Equation (3.8) symbolically can be done using Equations (3.9)-(3.10) according to [8].

$$
C = AB \tag{3.8}
$$
$$
\bar{A} = \bar{C}B^T \tag{3.9}
$$
$$
\bar{B} = A^T\bar{C} \tag{3.10}
$$

Note that this matches the results derived in Section 1.1 for the $2 \times 2$ matrix-matrix product. While differentiating the matrix-matrix product with AAD has a run time and memory complexity of $O(nmp)$, utilizing the symbolic evaluation reduces the memory overhead to $O(nm + mp)$, or respectively from $O(n^3)$ to $O(n^2)$ in the quadratic case. The input matrices $A$ and $B$ must be saved in the augmented primal run and then be multiplied with the adjoint values of the output according to Equations (3.9)-(3.10) in the adjoint run. Note that the run time complexity can not be improved using the symbolic evaluation.

Regarding the implementation, it is again desired that the symbolic evaluation is automatically performed when computing the derivatives. To find an appropriate entry point, it makes sense to investigate how Eigen determines what kind of product evaluation to use. This is done via the following struct:

```cpp
template<int Rows, int Cols, int Depth> struct product_type_selector;
```

Different evaluation strategies are determined by several specializations of this struct. The idea is to introduce a custom evaluation strategy so that AD-O tool modules can provide their own implementations. In order to specialize the selector for custom AD data types, it needs to be extended to include the resulting scalar type of the product operation. Since Eigen also offers a sparse matrix type, the two operand types of the product are also of interest to only hit for, e.g. dense matrices. Like in the case of the SD solver, an additional dummy template parameter is added as well to allow the use of `enable_if`. The modified struct is used to provide an entry point for the Eigen-AD base module named `eigen_enable_custom_product`. AD-O tool modules can use the provided API displayed in Listing 5 to enable a symbolic product for their custom AD data type `Scalar`, depending on the input dimensions $M$, $N$ and $Depth$, as well on the operand types `Lhs` and `Rhs`.

1 //
2 // API - An AD Tool can specialize/define these structs
3 //
template<typename Scalar, int M, int N, int Depth, typename Lhs, typename Rhs, typename enable_if = void>
struct eigen_ad_product_enable {};

template<typename Lhs, typename Rhs, typename T>
struct eigen_ad_product;

Listing 5: Eigen-AD base module product API.

The symbolic product implementation has to be placed inside the `evalTo(dst, lhs, rhs)` function in the definition of the `eigen_ad_product` struct.

### 3.3 Further remarks on the implementation with dco/c++

The following general properties are shared among all symbolic implementations in dco/c++:

- All linear algebra computations are performed using passive data types. This preserves all optimizations Eigen implements for the intrinsic data types (e.g., vectorization) and leads to a significant performance improvement. It is worth noting that the Stan Math Library has followed the same principle [12].

- For higher-order derivative computations, the symbolic implementations can be used recursively so that again the linear algebra arithmetic is performed with passive data types.

- dco/c++ provides a constant copy-overhead which is independent of the differentiation order; i.e., no matter what order, only a single copy operation to the passive data type is required.

### SD Solvers

Referring to the information given in Section 3.2.1 it makes sense to first list which data must be available in the reverse run before presenting the implementation details:

- Equations (3.2)-(3.3) compute adjoints of the inputs, which are the system matrix and the right hand side vector. As these inputs might be used for other computations in the program, their adjoints should not be overwritten but instead be incremented by the computed values. Therefore, a reference to these input adjoints must be available.

- The adjoints of the solution vector are required to evaluate Equation (3.2).

- Both equations also utilize non-adjoint data, namely the solution vector and the system matrix, where for the latter the computed decomposition should be used in the reverse run.

Based on this, the implementation design can be explained. dco/c++ comes with a so-called adjoint callback interface, which – among other things – allows the user to implement "symbolic adjoints of numerical methods" [14]. Input and output variables are registered in a so-called callback object which is inserted at the current tape position during the augmented primal run. The corresponding code section is evaluated without computing derivative information; in this case, the respective Eigen solver is called with the passive data type. During the adjoint run,
the user defined callback object is provided with the propagated output adjoints and is then responsible to increment the referenced input adjoints, which fulfills the first two points above.

Regarding the third point, the solution vector can simply be saved in the callback object. In order to provide the decomposition in the adjoint run, it would be possible to save the input matrix in the augmented primal run and then decompose it again in the adjoint run. However, this would introduce a run time overhead of $O(n^3)$ in the adjoint run. Instead, it is beneficial to save the whole decomposition. In the augmented primal run, the decomposition is performed using a primal solver, i.e. the default Eigen implementation with the passive data type. This solver is kept in memory by dco/c++ and is reused in the adjoint run to solve the transposed system in Equation (3.2). Since it is possible to have multiple solve calls for different right hand sides using the same solver and even reusing solver objects for different input matrices, it is then a matter of data bookkeeping and organizing to ensure that all data is correctly stored in the augmented primal run and later accessed in the adjoint run. This has been realized by using two different types of callback objects, one for each decomposition and one for each solve call. The big advantage of this design is that the decomposition, which is the dominant factor run time and memory-wise, is only performed in the augmented primal run, effectively canceling the overhead from AD completely for larger matrix dimensions.

Other symbolic functions use their own callback object and may save additional data in it (e.g. the transposed inverse for the symbolic inverse() function). They also make use of the decomposition callback and can use the primal solver, like the symbolic logAbsDeterminant() function to compute the inverse of the input matrix in the adjoint run. As a side note, the symbolic operation is less favorable for the HouseholderQR solver in this example, as it does not provide an inverse() function. This requires an additional decomposition using a different primal solver in the adjoint run. However, the other QR solvers use their inverse() function and can even utilize it symbolically in the adjoint run when computing higher-order derivatives.

Matrix-Matrix Product

In the same way as for the SD solvers, input Matrices $A$ and $B$ of Equation (3.8) for the matrix-matrix product are saved in a callback object which evaluates Equations (3.9)-(3.10) in the adjoint run.

Measurements have shown that the symbolic implementation only pays off for the matrix-matrix product. Other operations, like the matrix-vector product, have an algorithmic run time and memory complexity of $O(n^2)$. The symbolic implementation therefore has the same complexity as the algorithmic one, but requires additional copies. Since these linear algebra operations are usually very fast, the overhead introduced by copying makes the symbolic implementation not beneficial and it is therefore not imposed in this case.

4 Benchmarks

In order to verify the changes and implementations presented above, extensive measurements were made. They were performed using a single thread on an i7-6700K CPU running at 4 GHz with AVX2 enabled and 64 GB RAM available for the tape recording, using the g++ 7.4 compiler on Ubuntu 18.04. The respective linear algebra operations were performed for increasing matrix size $n$ using dynamic-sized quadratic matrices $\mathbb{R}^{n \times n}$ and one evaluation of the first-order adjoint model was computed with all output adjoints set to 1. The inverse() results shown here use the
underlying PartialPivLU, the logAbsDeterminant() function the FullPivHouseholderQR solver. From now on, the 
\texttt{dco/c++} Eigen module is referred to as \texttt{dco/c++/eigen}, and computations which are not using symbolic implementations but only plain overloading are denoted as algorithmic or simply as AD.

### 4.1 \texttt{dco/c++/eigen} benchmarks

In this section, the theoretical considerations from Section 3.2 are validated with benchmarks. To emphasize the improvements, reference measurements for the corresponding algorithmic computations only using \texttt{dco/c++} are given where appropriate.

#### 4.1.1 Memory consumption

All symbolic evaluations introduced in Section 3.2 lower the memory overhead introduced by an AD-O tool to $O(n^2)$. In order to visualize this effect, the tape size of \texttt{dco/c++} has been measured for the algorithmic and for the symbolic implementations and is displayed in Figure 7.

For clarity reasons, only selected dense solvers are shown in Figure 7a but similar patterns were measured for the other solvers as well. The symbolic implementation keeps a complete primal solver in memory so it can be reused in the adjoint run, which makes measuring the exact amount of occupied memory complicated. However, every dense solver in Eigen consists of class members which do not consume more than $O(n^2)$ of memory. Therefore, for simplicity reasons, all SD solvers are accounted with a $n \times n$ matrix on the tape and are summarized in a single measurement. Since the symbolic logAbsDeterminant() function does not require any additional data, it has the same memory usage as its corresponding solver. The symbolic inverse() function additionally saves the transposed input matrix, the symbolic matrix-matrix product stores the two input matrices.

As it was expected, all new implementations have a memory complexity of $O(n^2)$, while the algorithmic versions display a cubic behaviour. Since the AD implementations quickly exceed the amount of available RAM, they are not suited for higher input dimensions and no data is...
Figure 8: Total run times and run times of the adjoint run of the new symbolic operations: As described in Sections 3.2.1 and 3.3, the symbolic solve(b) function improves the run time complexity of the adjoint run to $O(n^2)$ and effectively cancels the AD overhead compared to the $O(n^3)$ primal run time complexity. The other symbolic operations still have the same run time complexity of the adjoint run as the primal code, but are noticeably faster.

4.1.2 Run time analysis

Figure 8 visualizes the run time measurements for the symbolic operations, split into total execution time and run time of the adjoint section. As stated in Section 3.2.1, solving a system of linear equations reduces the adjoint run time to $O(n^2)$, which is confirmed by the measured run times in Figure 8a. All other symbolic evaluations do not lower the complexity, since a matrix-matrix product or an inverse must be computed in the adjoint run. However, as it can be inferred from the gap between total and adjoint run times in Figure 8b, the overhead by the adjoint run is rather moderate.

4.1.3 Comparison to primal operations

To put the symbolic run times into perspective, the primal run times have been recorded as well. Comparing them both by computing the factor between the respective run times is a good measure to assess the performance of the derivative computation. The results are displayed in Figure 9.

In contrast to the previous run time analysis, we now compare to the primal code which is highly optimized. Beside the overhead introduced by the AD adjoint section, additional copy instructions are performed in the augmented primal run by the symbolic operations. Due to the convenient fact that the symbolic solve(b) evaluation reduces the run time overhead to $O(n^2)$, all solvers will converge towards a factor of 1 with increasing matrix size, since the ratio is dominated by the $O(n^3)$ primal code. However, as it can be seen in Figure 9a, the conversion rate depends on the specific solver. For the other symbolic operations, the run time complexity of the adjoint run can not be improved, which makes a factor of 1 impossible. Instead, the factor depends on the additional computations performed in the adjoint run. For the presented
symbolic evaluations, an obtainable factor of 2-3 appears to be reasonable. Figure 9(b) shows a corresponding convergence pattern. Like for the solve(b) implementation, the patterns differ among the symbolic operations.

Generically speaking, for very fast primal operations – like the optimized Cholesky (LLT) solver or the matrix-matrix product – it is hard to achieve good factors for smaller input dimensions due to the additional copy overhead introduced by the symbolic implementation. In contrast to that, operations with a higher computational complexity, like the JacobiSVD decomposition or the logAbsDet() function, are dominated by the primal code run time-wise and no significant overhead can be measured even for small matrices.

4.2 Comparison to other AD-O tools

To put the above given measurements into perspective, it is reasonable to compare them to results from other AD-O tools. The following tools were considered:

- Adept[11]
- ADOL-C[19]
- CoDiPack[20]
- FADBAD++ 2.0[17]
- Stan Math Library[12]

Although it was stated in Chapter 2 that it is straight forward to apply an AD-O tool to Eigen, incompatibilities may still occur depending on the specific tool. As an example, Adept specifies generic operator overloads and does not explicitly formulate them for all its expression

\[\text{http://www.fadbad.com/fadbad.html}\]
types. Eigen on the other hand makes repeated use of MatrixBase<T> as a generic parameter type for operators in order to accept all its expression types. Combining both tools then leads to compile-time ambiguousness for the operator overloads. Adept also states in its user guide that it “does not work optimally (if it works at all)” with Eigen. Issues have also been observed for FADBAD++. Nonetheless, small tweaking allowed running the matrix-matrix product benchmark for both tools as well, therefore it will be used later to compare the run times of all tools.

4.2.1 Stan Math Library

As mentioned in Chapter 1, the Stan Math Library is an AD-O tool which internally relies on Eigen for its features. Its ”extensive built-in library includes functions for matrix operations, linear algebra, differential equation solving, and most common probability functions” [12]. It is embedded in Stan, a platform for statistical modeling and high-performance statistical computations[15]. The aim of this section is to briefly sketch the similarities and differences of the Stan Math Library with dco/c++ and dco/c++/eigen.

The latest version of the Stan Math Library can be accessed via GitHub[16]. Beside the adjoint mode, it includes a tangent mode and is able to compute higher-order derivatives using any combinations of both modes. It can be applied to an arbitrary code by using its custom AD data type and can therefore be considered as an AD-O tool. However, it must be noted that the Stan Math Library does not offer an API for advanced AD users, e.g. lacking a callback system or the possibility to compute arbitrary adjoint directional derivatives in the adjoint mode, only offering a grad() functionality.

Although the Stan Math Library can be applied to arbitrary code, we see its advantages rather on the applications it was especially designed for, e.g. linear algebra using its own API and storage types from Eigen. dco/c++ on the other hand is aimed to be used for more generically written code and therefore requires less modifications or restrictions to be applied, while still providing a good performance. This can be inferred from Figure 10, where std::vector was used as the storage type for computing

\[ y = \left( \sum_{i}^{n} x_i^2 \right)^2 \]

The Stan Math Library is more suited for applications where functions from its API can be utilized. These require Eigen types as input parameters, and then call the respective Eigen function with passive types while a symbolic evaluation takes place in the adjoint run. This is very similar to dco/c++/eigen, except that a special API is required by the Stan Math Library in contrast to specializations. However, there are very few cases where the Stan Math Library also specializes Eigen code for its own custom AD data type. In addition to the matrix-matrix

---

[14] https://eigen.tuxfamily.org/dox/TopicFunctionTakingEigenTypes.html
[15] https://mc-stan.org/
[16] https://github.com/stan-dev/math

---

Figure 10: Run time comparison of AAD on non-Eigen specific code using the Stan Math Library with dco/c++ and dco/c++/eigen.
Figure 11: Run time comparison of different AD-O tools for the matrix-matrix product: Besides plain algorithmic overloading versions, Adept and Stan also offer optimized functions via their API (denoted by parenthesis).

product benchmark, more measurements were made to compare \texttt{dco/c++/eigen} with the Stan Math Library (from now on referred to as Stan).

4.2.2 Benchmark Comparison

As mentioned, all tools were able to evaluate the matrix-matrix product benchmark. At this point, it must be said that the shown run times do not imply the feasibility of the tools in general, since they are all designed with different use cases and restrictions in mind. They were utilized to our best knowledge, but no tool specific experts were involved in these measurements. While \texttt{dco/c++/eigen} provides its best performance with this setup, we believe that other tools can be optimized by their developers to get similar results. Therefore, the given results only represent the current situation and are likely to change in the future. The purpose here is to visualize the run times of the AD-O tools for the matrix-matrix product, where only plain overloading of the Eigen library is performed or a special API is used for this particular computation. Since Stan only provides a \texttt{grad()} function, its benchmark was modified to compute the scalar value \( z = (A \ast B).\text{sum()} \) and the corresponding gradient of \( z \). All of the following results were produced using Eigen-AD. However, internal benchmarks have not shown a considerable difference to the standard Eigen version.

The measured run times are displayed in Figure 11. Note that the notion \texttt{dco/c++} refers to plain overloading, and the remark \texttt{auto only} describes the usage of the \texttt{dco/c++/eigen} module without any symbolic implementations, i.e. only with the optimization from Section 3.1 in place. In contrast to that, \texttt{full} names the default behaviour when using the module, with the auto return type deduction and symbolic implementations enabled.

All non-specialized tools show the same computational complexity. Differences are non-negligible, though. The feasibility of the auto return type deduction of \texttt{dco/c++/eigen} introduced in Section 3.1 can be observed, since the smaller amount of temporaries speeds up the computation. In contrast to the other general purpose AD-O tools, Adept also allows the computation of a matrix-matrix product using the \texttt{matmul} function from its API. In this case, no Eigen is used
but instead the storage types defined by Adept. As it can be expected, this specially designed feature from Adept is faster than the general AD-O tool approach. This also applies to Stan, although in this case it really is plain AD-O using Eigen storage types. Recall that Stan specializes a few Eigen functions; in fact, it internally evaluates optimized matrix-vector products which speed up the computation.

Referring to the insight gained in Section 3.2, symbolic evaluations have the potential to drastically improve the performance of AD application. In the case of the matrix-matrix product, actual computations are made using the passive data type and profit from all optimizations in Eigen, while only two additional matrix-matrix products need to be evaluated in the adjoint run. This explains why `dco/c++/eigen` as well as the implementation in the multiply function of Stan drastically outperform all other tools.

Due to the incompatibility of some tools with other Eigen functions, only the matrix-matrix product was chosen for a complete comparison. However, since Stan and `dco/c++/eigen` share similarities, further operations were compared between them. Figure 12 visualizes the run time for differentiating the log-abs-determinant computation for a quadratic matrix with increasing size. In this case, Stan provides a symbolic variant via its `log_determinant()` function. It is internally implemented using the `FullPivHouseholderQR` decomposition. Therefore, the algorithmic versions as well as the symbolic version of `dco/c++/eigen` also utilize this decomposition for this benchmark. The measurements again show that both symbolic variants perform almost identical. Regarding the algorithmic variants, it can be said that `dco/c++` stops the computation earlier because it restricts its memory usage, but otherwise performs in a similar time like Stan.

A big advantage of `dco/c++/eigen` are the SD solvers, which were presented in Section 3.2.1. Stan provides a `mdivide_left(A,b)` function to solve a system of linear equations which will always use the `ColPivHouseholderQR` decomposition internally to solve the system. Therefore, the algorithmic and the `dco/c++/eigen` measurements displayed in Figure 13 utilize this solver class. Although Stan also evaluates the adjoints symbolically, the implementation of `dco/c++/eigen` presented in Section 3.2.1 is faster. The reason for that is that Stan performs another decomposition in the adjoint run, while the implementation of `dco/c++/eigen` keeps the decomposition from the augmented primal run in memory and reuses it as described in Section 3.3. While `dco/c++/eigen` reduces the run time overhead for differentiating the computation to $O(n^2)$, Stan keeps it at $O(n^3)$.

The current API implementation of Stan limits the user to prescribed underlying decompositions like `ColPivHouseholderQR` and `FullPivHouseholderQR` as stated above. `dco/c++/eigen` on the other hand allows the utilization of any of the supporting Eigen decompositions. There are several
more symbolic functions of other linear algebra operations implemented inside Stan which Eigen-AD or \texttt{dco/c++/eigen} are currently lacking. When used, \texttt{dco/c++/eigen} has to fallback to algorithmic evaluation which will be much slower than the evaluation of Stan. Adding such symbolic implementations is planned for the future.

5 Conclusion & Outlook

In this work, we have outlined challenges which occur when calculating derivatives for linear algebra operations using an AD-O tool with the Eigen library. To overcome these issues, the modified library fork Eigen-AD was developed, aimed at authors of AD-O tools to help them improve the performance of their software when applying it to Eigen. Changes to the Eigen source code were kept generic and entry points are provided for a general Eigen-AD base module which can be utilized by an individual AD-O tool module via a dedicated API. Care was taken to realize the improvements via \texttt{C++} specializations, which keep the look and feel of plain AD-O. General performance improvements were made regarding the usage of expression templates by the AD-O tool and specific operations can now be reimplemented conveniently by an AD-O tool module in order to provide symbolic implementations.

As a showcase, such a module has been implemented for the AD-O tool \texttt{dco/c++}, where important linear algebra operations like the matrix-matrix product or the solving of a linear system are differentiated symbolically. Benchmarks have validated the theoretical considerations, and clearly show the improvements in computational complexity regarding run time and memory usage compared to the corresponding AD-O computations. Operations involving larger matrix dimensions are now feasible to be differentiated, as AD-O tools were previously limited by the amount of available RAM due to the extensive memory usage. For linear system solvers, the overhead of an AD-O tool cancels for increasing matrix sizes, since its complexity of $\mathcal{O}(n^2)$ is of lower order than the primal operation. Comparisons with other AD-O tools were made to put the produced results into context. Run times of the Stan Math Library, a tool which is highly optimized for the usage with Eigen, could be matched with the presented implementation and even surpassed in the case of solving a linear system of equations due to the reuse of the decomposition object in the adjoint run.

The presented fork Eigen-AD has passed its internal development phase and access to it can be requested from the authors. Depending on the demand, a public release is considered. Other AD software authors are welcome to provide a module for their AD-O tool which can be included in the fork, as well as participate in the development of Eigen-AD. Investigation into more parts of Eigen are aimed for the future in order to extend the API provided by the Eigen-AD base module. This includes, e.g. the \texttt{Eigenvalues} and \texttt{Sparse} features of Eigen.
Regarding the implementation of dco/c++/eigen, more symbolic implementations for the SD solvers will be made. Parallelization, using, e.g. OpenMP, is also of interest. While the presented implementations are only aimed at the adjoint mode, the Eigen-AD base module API also allows the inclusion of optimized tangent operations. These can be implemented in dco/c++/eigen in the future as well.

Furthermore, there has been communication with the Eigen development team and best efforts were made to keep changes to the Eigen source as general as possible. In combination with the modular setup regarding the Eigen-AD base module and individual tool modules, a possible integration of the changes into the stock Eigen version in the future can be discussed.

All in all, this work has shown the potential of adjusting a linear algebra library to optimize the evaluation of derivatives using an AD-O tool. In the case of Eigen, relatively small changes to its source code allow to provide a general API which can be utilized by other AD-O tools. Possible symbolic implementations have a significant performance advantage over the plain application of an AD-O tool.
References

[1] Andreas Griewank. *Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, 2000, second edition 2008.

[2] Uwe Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2012.

[3] Mu Wang, Guang Lin, and Alex Pothen. Using automatic differentiation for compressive sensing in uncertainty quantification. *Optimization Methods and Software*, 33(4-6):799–812, 2018, https://doi.org/10.1080/10556788.2017.1359267.

[4] Christian H. Bischof, H. Martin. Bcker, and Arno. Rasch. Sensitivity analysis of turbulence models using automatic differentiation. *SIAM Journal on Scientific Computing*, 26(2):510–522, 2004, https://doi.org/10.1137/S1064827503426723.

[5] Nicolas R. Gauger, Andrea Walther, Carsten Moldenhauer, and Markus Widhalm. Automatic differentiation of an entire design chain for aerodynamic shape optimization. In Cameron Tropea, Suad Jakirlic, Hans-Joachim Heinemann, Rolf Henke, and Heinz Hönlinger, editors, *New Results in Numerical and Experimental Fluid Mechanics VI*, pages 454–461, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[6] Koichi Kubota. Computation of matrix permanent with automatic differentiation. In Martin Bücker, George Corliss, Uwe Naumann, Paul Hovland, and Boyana Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, pages 67–76, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[7] Peder Olsen, Steven Rennie, and Vaibhava Goel. Efficient automatic differentiation of matrix functions. *Lecture Notes in Computational Science and Engineering*, 87, 01 2012.

[8] Mike B. Giles. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In Christian H. Bischof, H. Martin Bücker, Paul Hovland, Uwe Naumann, and Jean Utke, editors, *Advances in Automatic Differentiation*, pages 35–44, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[9] Matthias W. Seeger, Asmus Hetzel, Zhenwen Dai, and Neil D. Lawrence. Automatic differentiation of linear algebra. *CoRR*, abs/1710.08717, 2017, 1710.08717.

[10] Benjamin Z. Dunham. *High-Order Automatic Differentiation of Unmodified Linear Algebra Routines via Nilpotent Matrices*. Dissertation, Department of Aerospace Engineering Sciences, University of Colorado at Boulder, Boulder, USA, 2017.

[11] Robin J. Hogan. Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Trans. Math. Softw.*, 40(4):26:1–26:16, July 2014.

[12] Bob Carpenter, Matthew D. Hoffman, Marcus Brubaker, Daniel Lee, Peter Li, and Michael Betancourt. The stan math library: Reverse-mode automatic differentiation in C++. *CoRR*, abs/1509.07164, 2015, 1509.07164.

[13] Todd Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 06 1995.
[14] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Ruede. Expression templates revisited: A performance analysis of current methodologies. *SIAM Journal on Scientific Computing*, 34(2):42–69, 2012.

[15] The Numerical Algorithms Group Ltd. (NAG), Wilkinson House, Jordan Hill Road, Oxford OX2 8DR, United Kingdom. *dco/c++ User Guide version 3.2.0*.

[16] Johannes Lotz. *Hybrid Approaches to Adjoint Code Generation with dco/c++*. PhD thesis, RWTH Aachen University, 03 2016.

[17] Uwe Naumann and Johannes Lotz. Algorithmic differentiation of numerical methods: Tangent-linear and adjoint direct solvers for systems of linear equations. Technical report, RWTH Aachen, Department of Computer Science, 06 2012.

[18] Kaare B. Petersen and Michael S. Pedersen. The matrix cookbook, nov 2012. Version 20121115.

[19] Andrea Walther. Getting started with ADOL-C. In *Combinatorial Scientific Computing*, 2009.

[20] Max Sagebaum, Tim Albring, and Nicolas R. Gauger. High-performance derivative computations using CoDiPack. 2017, 1709.07229.