Eigen-AD: Algorithmic Differentiation of the Eigen Library

Patrick Peltzer, Johannes Lotz, and Uwe Naumann

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

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.

Keywords: Algorithmic Differentiation · Linear Algebra · Eigen.

1 Introduction

In this work, the `C++` linear algebra library Eigen[1] is used as a base software implementing linear algebra operations for which derivatives are to be computed using Algorithmic Differentiation (AD) [6,10] by overloading. Derivatives of computer programs can be of interest, e.g. when performing uncertainty quantification [18], sensitivity analysis [1] or shape optimization [4]. AD enables the computation of derivatives of the output of such programs with respect to their inputs. This is done using the tangent model in *tangent mode* or the adjoint model in *adjoint mode*, 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*.

[1] http://eigen.tuxfamily.org

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_51

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 they require the usage of an extended floating point data type (from now referred to as the *custom AD data type*). This change in data type is often impractical and breaks hand tuned performance gains [3]. Therefore, software combining AD-O and linear algebra has been realized with, e.g. Adept [7] or the *Stan Math Library*[2], where the latter makes heavy use of Eigen. Eigen allows the direct utilization of AD-O tools due to its extensive use of `C++` templates. At a later point in this paper, concrete implementations and benchmarks for AD-O in Eigen will be presented using `dco/c++`, which is an AD-O tool actively developed by NAG Ltd.[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. Section 3 presents *Eigen-AD* which is a fork of Eigen. It contains several optimizations and improvements for the application of an AD-O tool, which are demonstrated and benchmarked using `dco/c++` in Section 4. Section 5 summarizes the results and suggests possible future works.

Note than an extended version of this work exists [13]; refer to it for further details.

2 Using AD-O and Symbolic Derivatives

Most 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.

[2] http://www.autodiff.org/
[3] https://www.nag.co.uk/

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_51

Let this specific product kernel be implemented using Eigen as follows:

```
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 1.1. 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. Fig. 1 displays the computational graph of the matrix-matrix multiplication kernel in Listing 1.1. 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 1.1 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 Fig. 1 can be written as follows:

$$\begin{aligned}
\bar{A}_{0,0} &= \bar{C}_{0,0}B_{0,0} + \bar{C}_{0,1}B_{0,1} & \bar{B}_{0,0} &= \bar{C}_{0,0}A_{0,0} + \bar{C}_{1,0}A_{1,0} \\
\bar{A}_{0,1} &= \bar{C}_{0,0}B_{1,0} + \bar{C}_{0,1}B_{1,1} & \bar{B}_{0,1} &= \bar{C}_{0,1}A_{0,0} + \bar{C}_{1,1}A_{1,0} \\
\bar{A}_{1,0} &= \bar{C}_{1,0}B_{0,0} + \bar{C}_{1,1}B_{0,1} & \bar{B}_{1,0} &= \bar{C}_{0,0}A_{0,1} + \bar{C}_{1,0}A_{1,1} \\
\bar{A}_{1,1} &= \bar{C}_{1,0}B_{1,0} + \bar{C}_{1,1}B_{1,1} & \bar{B}_{1,1} &= \bar{C}_{0,1}A_{0,1} + \bar{C}_{1,1}A_{1,1}
\end{aligned}$$

$$\Rightarrow \bar{A} = \bar{C}B^T \quad (1) \qquad \Rightarrow \bar{B} = A^T\bar{C} \quad (2)$$

Adjoint values are denoted with a bar. Equations (1) and (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 1.1. 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.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_51

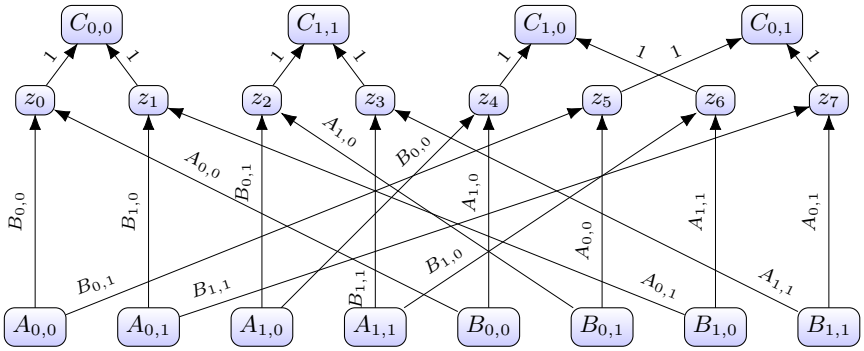


Fig. 1. Computational graph for the 2×2 matrix product kernel displayed in Listing 1.1: Vertices represent variables accessed in the augmented primal run, edges define computation operands and edge weights are the respective partial derivatives. Temporaries z storing the results of the $*$ operator in line 6 may be optimized away by an AD-O tool when taping.

3 Eigen-AD

Applying an AD-O tool to Eigen will lead to severe limitations sooner or later. Eigen comes with optimized kernels, e.g. for 8-byte double precision data. Traditionally, custom AD data types are larger and these optimizations do not work anymore. Regarding AAD application, 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 derivatives not feasible at all.

To overcome these issues, the Eigen source code has been adjusted and extended 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 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 Fig. 2 for the package architecture.

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. Compiling and running the tests successfully ensures compatibility of the AD-O tool with all of the tested Eigen functions. 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

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_51

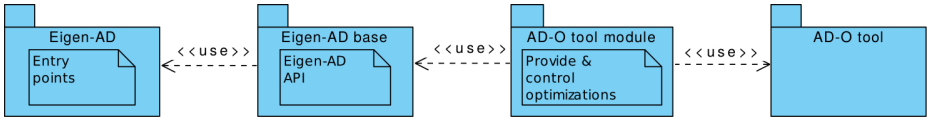


Fig. 2. Eigen-AD package architecture: Authors can implement an AD-O tool module for their AD-O tool.

improvements shall not require a separate interface but be used automatically whenever the AD-O tool is applied.

The next sections present optimization approaches realized in Eigen-AD.

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 [16]. The first aspect, also known as *lazy evaluation*, has been complemented by the concept of *Smart Expression Templates* [8] which both are implemented in Eigen.

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 Fig. 1 in Section 2. 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 [12,7].

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 template expressions. The returned template expression 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. Its class template parameter Scalar corresponds to the custom AD data type and it is also used in the parenthesis operator as 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 an additional vertex in the computational graph, analogous to the temporaries z of the computational graph in Fig. 1.

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

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_51

at this level. 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. 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.

Evaluating the modified scalar_opposite_op functor will avoid the additional vertex in the computational graph. This optimization can be applied to all Eigen scalar functors and also to several Eigen math functions like sin or exp which are supported by the AD-O tool's expression templates.

3.2 Symbolic Derivatives

As introduced in Section 2, 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 [9].

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 for any computation concerning a dense solver are supported. Refer to the Eigen-AD base module technical guide for further information. In the next sections, equations for symbolic adjoints of selected operations are introduced.

SD Dense System Solver Consider the system of linear equations:

$$A\mathbf{x} = \mathbf{b} \tag{3}$$

where $A \in \mathbb{R}^{n \times n}$ is the system matrix, $\mathbf{b} \in \mathbb{R}^n$ is the right hand side vector and $\mathbf{x} \in \mathbb{R}^n$ is the solution vector. There exists a wide variety of approaches to solve the problem shown in Equation (3) which make use of decomposing the matrix A into a product of other matrices, e.g. the *LU decomposition*. Eigen offers one dense solver class for each decomposition type.

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 (4)-(5) as presented in [5], the decomposition is completely excluded from taping and interpreting.

$$A^T \cdot \bar{\mathbf{b}} = \bar{\mathbf{x}} \tag{4}$$

$$\bar{A} = -\bar{\mathbf{b}} \cdot \mathbf{x}^T \tag{5}$$

As it can be seen, the adjoint values of the right hand side vector $\mathbf{b}$ 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 $\mathcal{O}(n^2)$ [11].

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_51

Symbolic Inverse Inverting a matrix, i.e. computing

$$C = A^{-1} \tag{6}$$

is implemented in Eigen as a member function of a dense solver. Corresponding adjoints can be computed using Equation (7) [5].

$$\bar{A} = -C^T \bar{C} C^T \tag{7}$$

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

Symbolic Log-Abs-Determinant Another member function of the dense solvers 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 (8). Such a computation is relevant for, e.g. computing the log-likelihood of a multivariate normal distribution.

$$x = \log |\det(A)| \tag{8}$$

$$\bar{A} = A^{-T} \bar{x} \tag{9}$$

Equation (8) is implemented in Eigen for the QR dense solvers, and adjoints can be computed according to Equation (9) [14]. The inverse can be computed by reusing the decomposition which was kept in memory for the adjoint run. While the run time overhead is still $\mathcal{O}(n^3)$, the symbolic implementation improves the memory overhead to $\mathcal{O}(n^2)$.

Symbolic Matrix-Matrix Product For $A \in \mathbb{R}^{n \times m}, B \in \mathbb{R}^{m \times p}, C \in \mathbb{R}^{n \times p}$, the adjoints of the matrix-matrix product in Equation (10) can be computed using Equations (11)-(12) according to [5].

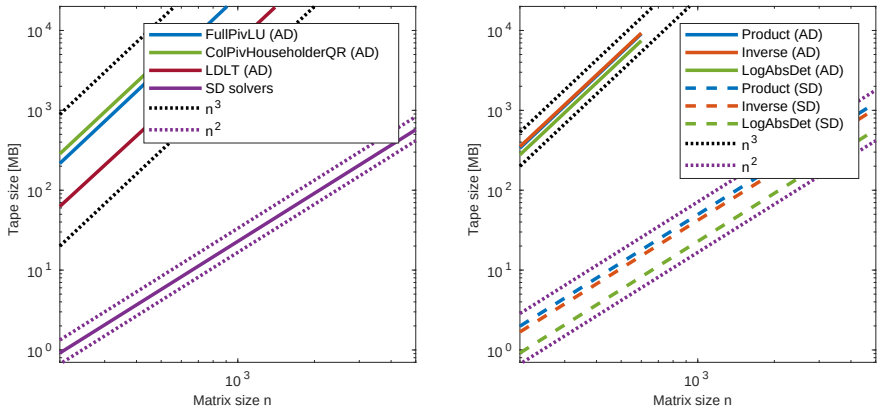
$$C = AB \tag{10}$$

$$\bar{A} = \bar{C} B^T \tag{11}$$

$$\bar{B} = A^T \bar{C} \tag{12}$$

Note that this matches the results derived in Section 2 for the 2×2 matrix-matrix product. While differentiating the matrix-matrix product with AAD has a run time and memory complexity of $\mathcal{O}(nmp)$, utilizing the symbolic evaluation reduces the memory overhead to $\mathcal{O}(nm+mp)$. 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 (11)-(12) in the adjoint run. Note that the run time complexity can not be improved using the symbolic evaluation.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_51



(a) solve(b) of selected solvers **(b)** Other symbolic operations

Fig. 3. Tape size comparison between AD and SD solvers: All new symbolic implementations have a memory complexity of $\mathcal{O}(n^2)$ compared to $\mathcal{O}(n^3)$ of the algorithmic versions.

4 Benchmarks

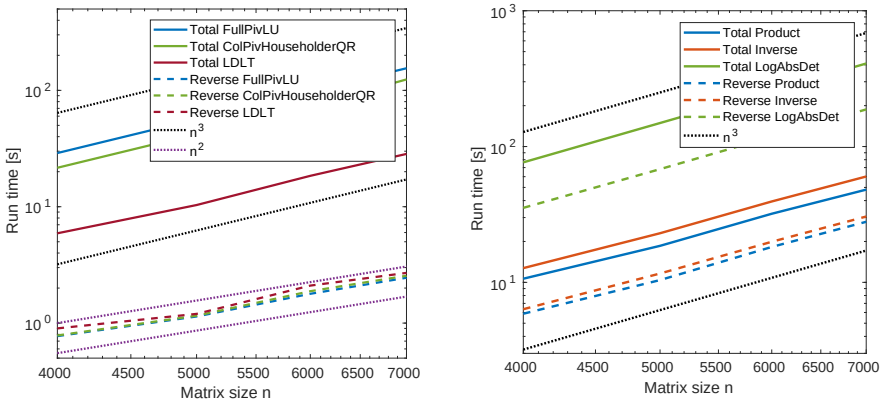
As mentioned in the beginning of this paper, an Eigen-AD tool module has been implemented for `dco/c++`. In order to verify the implementation, extensive measurements were made. They were performed using a single thread on an i7-6700K CPU running at 4 GHz with AVX2 enabled and 64 GB RAM available for 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 `dco/c++` Eigen module is referred to as `dco/c++/eigen`, and computations which are not using symbolic implementations but only plain overloading are denoted as algorithmic or as AD.

4.1 `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 without the `dco/c++/eigen` module are given where appropriate.

Memory consumption All symbolic evaluations introduced in Section 3.2 lower the memory overhead introduced by an AD-O tool to $\mathcal{O}(n^2)$. In order to visualize this effect, the tape size of `dco/c++` has been measured for the algorithmic and for the symbolic implementations and is displayed in Fig. 3. For clarity

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_51



(a) solve(b) of selected solvers **(b)** Other symbolic operations

Fig. 4. Total run times and run times of the adjoint run of the new symbolic operations: As described in Section 3.2, the symbolic solve(b) function improves the run time complexity of the adjoint run to $\mathcal{O}(n^2)$ and effectively cancels the AD overhead compared to the $\mathcal{O}(n^3)$ 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.

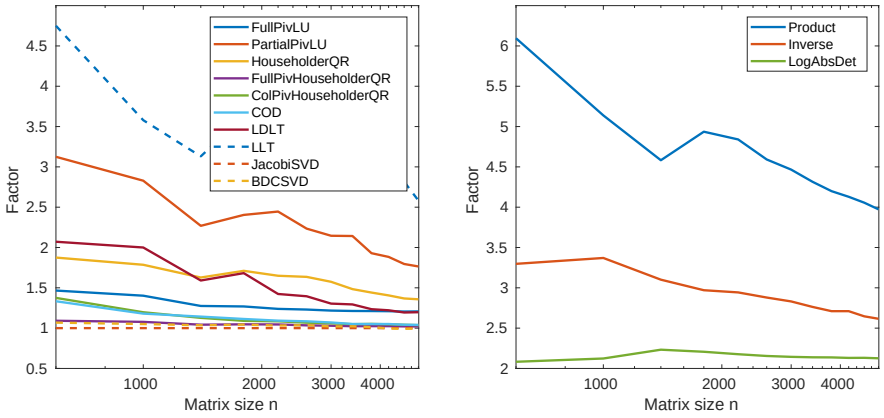
reasons, only selected dense solvers are shown in Fig. 3a, but similar patterns were measured for the other solvers as well. The symbolic implementations keep a complete primal solver in memory which is accounted with a $n \times n$ matrix on the tape. Since the symbolic logAbsDeterminant() function 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 $\mathcal{O}(n^2)$, while the algorithmic versions display a cubic behaviour and quickly exceed the amount of available RAM.

Run time analysis Fig. 4 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, solving a system of linear equations reduces the adjoint run time to $\mathcal{O}(n^2)$, which is confirmed by the measured run times in Fig. 4a. 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 Fig. 4b, the overhead introduced by the adjoint run is rather moderate.

Comparison to primal operations To put the symbolic run times into perspective, the primal run times have been recorded as well. Comparing them both

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_51



(a) solve(b) **(b)** Other symbolic operations

Fig. 5. Run time relation symbolic to primal operations: The run time overhead for the symbolic solve(b) implementation vanishes for larger matrices as all factors converge towards 1. For the other symbolic operations, the overhead does not vanish but still appealing factors are achieved.

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 Fig. 5.

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 $\mathcal{O}(n^2)$, all solvers will converge towards a factor of 1 with increasing matrix size, since the ratio is dominated by the $\mathcal{O}(n^3)$ primal code. However, as it can be seen in Fig. 5a, 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 between 2 and 3 is reasonable. Fig. 5b shows a corresponding convergence pattern.

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 higher computational costs – like the JacobiSVD decomposition or the logAbsDet() function – are dominated by the primal code run time-wise and no significant overhead from the AAD code can be measured even for small matrices.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_51

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[7]
- ADOL-C[17]
- CoDiPack[15]
- FADBAD++ 2.0[4]
- Stan Math Library [2]

All tools evaluate a matrix-matrix product of two randomly filled $\mathbb{R}^{n\times n}$ matrices. One evaluation of the first order adjoint model is performed using plain AD-O of the Eigen library or using the tool's special API if available. Since Stan only provides a grad() function, its benchmark was modified to compute the scalar value z = (A∗B).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.

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 `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 measured run times are displayed in Fig. 6. Note that the notion `dco/c++` refers to plain overloading, and the remark *auto only* describes the usage of the `dco/c++/eigen` module without any symbolic implementations, i.e. only with the optimization from Section 3.1 in place. In contrast to that, *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 `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 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. Stan specializes a few Eigen functions such that it internally evaluates optimized matrix-vector products for matrix-matrix products.

[4] http://www.fadbad.com/fadbad.html

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_51

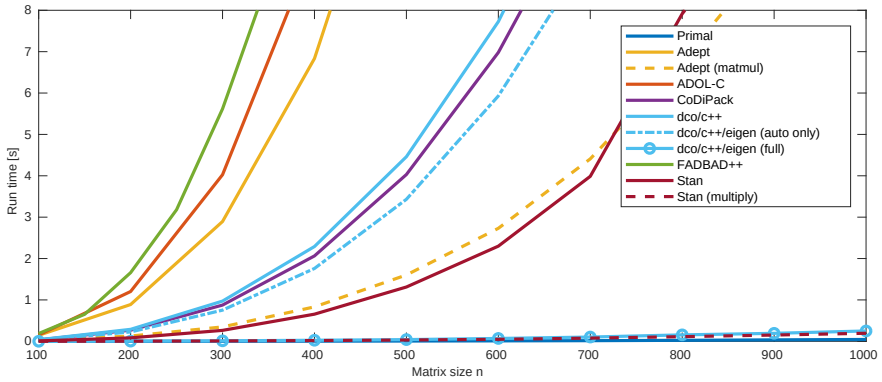


Fig. 6. Run time comparison of different AD-O tools for the matrix-matrix product: Besides plain algorithmic overloading versions, Adept and Stan also offer optimized functions via their API (denoted by parenthesis).

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.

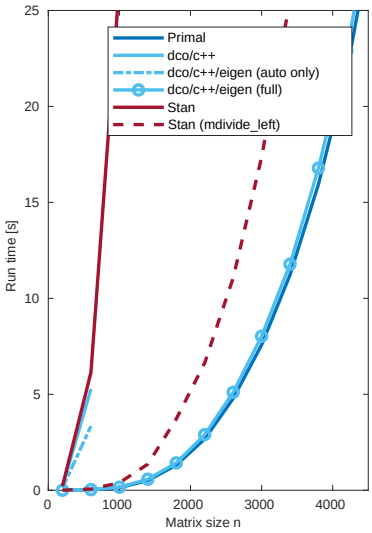


Fig. 7. Algorithmic and symbolic solve(b): Due to saving of the whole decomposition object, `dco/c++/eigen` achieves a faster run time than Stan.

Since Stan provides optimized linear algebra functions using Eigen, another benchmark was performed for solving a dense system. Stan offers a mdivide_left(A,b) function to solve a system of linear equations which internally will always use the ColPivHouseholderQR decomposition. Therefore, the algorithmic and the `dco/c++/eigen` measurements displayed in Fig. 7 also utilize this solver class. While Stan uses the same symbolic evaluation from Equations (4)-(5), it performs another decomposition in the adjoint run. `dco/c++/eigen` on the other hand keeps the decomposition from the augmented primal run in memory and reuses it later. While Stan keeps the AAD run time overhead `dco/c++/eigen` at $\mathcal{O}(n^3)$, the implementation in `dco/c++/eigen` improves it to $\mathcal{O}(n^2)$.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_51

5 Conclusion & Outlook

In this work, we have 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 `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 `dco/c++`, where important linear algebra operations like the matrix-matrix product or solving of a linear system are differentiated symbolically. Benchmarks have validated the theoretical considerations and underlined the improvements in computational complexity regarding run time and memory usage. It was shown that AD-O tool modules can cancel the AAD overhead for dense solvers with a corresponding implementation and comparisons with other AD-O tools were made to put the produced results into context which further confirmed the improvements.

Eigen-AD is publicly available[5] and 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 future development. Investigation into more parts of Eigen are planned in order to extend the Eigen-AD API. 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 partial integration of the changes into future Eigen versions should 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 and provide a superior performance compared to ordinary AD-O.

[5] https://gitlab.stce.rwth-aachen.de/stce/eigen-ad

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_51

References

1. Bischof, C.H., Bcker, H.M., Rasch, A.: Sensitivity analysis of turbulence models using automatic differentiation. SIAM Journal on Scientific Computing **26**(2), 510–522 (2004). https://doi.org/10.1137/S1064827503426723
2. Carpenter, B., Hoffman, M.D., Brubaker, M., Lee, D., Li, P., Betancourt, M.: The stan math library: Reverse-mode automatic differentiation in C++. CoRR **abs/1509.07164** (2015)
3. Dunham, B.Z.: 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)
4. Gauger, N.R., Walther, A., Moldenhauer, C., Widhalm, M.: Automatic differentiation of an entire design chain for aerodynamic shape optimization. In: Tropea, C., Jakirlic, S., Heinemann, H.J., Henke, R., Hönlinger, H. (eds.) New Results in Numerical and Experimental Fluid Mechanics VI. pp. 454–461. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
5. Giles, M.B.: Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In: Bischof, C.H., Bücker, H.M., Hovland, P., Naumann, U., Utke, J. (eds.) Advances in Automatic Differentiation. pp. 35–44. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
6. Griewank, A.: Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation. No. 19 in Frontiers in Appl. Math., SIAM, Philadelphia (2000, second edition 2008)
7. Hogan, R.J.: Fast reverse-mode automatic differentiation using expression templates in C++. ACM Trans. Math. Softw. **40**(4), 26:1–26:16 (Jul 2014). https://doi.org/10.1145/2560359
8. Iglberger, K., Hager, G., Treibig, J., Ruede, U.: Expression templates revisited: A performance analysis of current methodologies. SIAM Journal on Scientific Computing **34**(2), 42–69 (2012)
9. Lotz, J.: Hybrid Approaches to Adjoint Code Generation with dco/c++. Ph.D. thesis, RWTH Aachen University (03 2016)
10. Naumann, U.: The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2012)
11. Naumann, U., Lotz, J.: Algorithmic differentiation of numerical methods: Tangent-linear and adjoint direct solvers for systems of linear equations. Tech. rep., RWTH Aachen, Department of Computer Science (06 2012)
12. The Numerical Algorithms Group Ltd. (NAG), Wilkinson House, Jordan Hill Road, Oxford OX2 8DR, United Kingdom: dco/c++ User Guide version 3.2.0
13. Peltzer, P., Lotz, J., Naumann, U.: Eigen-ad: Algorithmic differentiation of the eigen library (2019), arXiv:1911.12604
14. Petersen, K.B., Pedersen, M.S.: The matrix cookbook (nov 2012), version 20121115
15. Sagebaum, M., Albring, T., Gauger, N.R.: High-performance derivative computations using CoDiPack (2017)
16. Veldhuizen, T.: Expression templates. C++ Report **7**(5), 26–31 (06 1995)
17. Walther, A.: Getting started with ADOL-C. In: Combinatorial Scientific Computing (2009)
18. Wang, M., Lin, G., Pothen, A.: 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

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_51