The reverse mode of automatic differentiation applied to the MATLAB language – advanced methods for adjoint code generation

Dem Fachbereich Informatik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doctor rerum naturalium (Dr. rer. nat.)
vorgelegte

Dissertation

von

Johannes Willkomm, Dipl.-Inf.

aus
Aachen

The reverse mode of automatic differentiation applied to the MATLAB language – advanced methods for adjoint code generation

Dissertation by Johannes Willkomm

1. Review: Prof. Dr. Christian Bischof
2. Review: Prof. Dr. Uwe Naumann

Date of submission: September 23, 2020
Date of thesis defence: –

Darmstadt – D17

Jede Zeit hat ihre Aufgabe, und durch die Lösung derselben rückt die Menschheit weiter.
Heinrich Heine (Reisebilder, Italien, Kap. 19)

Ich widme diese Arbeit meiner Familie, meinen Eltern, meiner Frau und meinen Kindern, die mich während der gesamten Zeit geduldig unterstützt haben.

Preface

This work was written over the course of several months of the year 2020. In this time, I was finally able wrap up my studies regarding the reverse mode of automatic differentiation for the MATLAB language (ADiMat) and write these pages. When putting to paper the ideas that guided me during the design and implementation of ADiMat, even some new results were obtained.

I would like to thank Prof. Christian Bischof and Prof. Martin Bücker for many fruitful discussion on the many subjects regarding and relating to automatic differentiation, from which oftentimes would spring up new ideas. I would also like to thank my wife and my family for supporting me throughout the entire time which I spend working on the software ADiMat and this dissertation, without which this work would not have been possible.

Aachen, in September 2020
Johannes Willkomm

Zusammenfassung

Das Software-Werkzeug Automatisches Differenzieren für MATLAB (ADiMat) wird um den Rückwärts- oder Adjungiertenmodus erweitert. Dieser erlaubt die hocheffiziente Berechnung der Ableitungen von skalaren Funktionen, d.h. langer Gradienten und großer Hesse-Matrizen, und spielt damit eine zentrale Rolle in vielen Verfahren zur numerischen Optimierung, beispielsweise Formoptimierung, Datenassimilierung und beim Deep Learning, beim Trainieren tiefgeschachtelter neuronaler Netze. ADiMat erlaubt die automatische Generierung von Adjungiertencode für beliebige in MATLAB geschriebene Funktionen und Programme. Zur Umsetzung des Adjungiertencodegenerators in ADiMat wurde erstmals XML zur Repräsentation des abstrakten Syntaxbaumes und XSLT zur Transformation desselben eingesetzt. Dieser innovative Ansatz im Compilerbau hat sich dabei als tragfähig und vielversprechend erwiesen.

Abstract

The software tool Automatic Differentiation for MATLAB (ADiMat) is enhanced with support for the reverse mode or adjoint mode. The adjoint mode allows the highly efficient evaluation of derivatives of scalar functions, i.e. of long gradients and large-scale Hessian matrices, and thus plays a central role in many algorithms for numerical optimization, such as shape optimization, data assimilation and deep learning, the training of deeply nested neural networks. ADiMat allows the automatic generation of adjoint code for arbitrary functions and programs written in MATLAB. For the first time, in the implementation of the adjoint code generator in ADiMat XML was used for representation of the abstract syntax tree while XSLT was used for the transformation of said tree. This innovative approach to compiler construction has proven itself viable and promising.

Contents

1 Introduction

In this dissertation thesis we describe our work to implement the *reverse mode* (RM) of *automatic differentiation* (AD) in the ADiMat software. This mainly takes the form of a *code generator* or *transpiler* for so-called *adjoint code*. This adjoint code is an augmented version of the original code which is able to evaluate the derivatives of some numerical results with respect to selected numerical parameters.

The reverse mode has several interesting properties which make it a valuable tool to obtain derivatives, in particular in the context of large-scale optimization. The relative costs for computing gradients and Hessian matrices with the reverse mode are on an entirely different asymptotical regime than all other methods for computing derivatives, both standard numerical methods or the forward mode of automatic differentiation. The only exception is solving adjoint PDEs to obtain the gradient of a PDE solution, hence the name adjoint code. Thus, an adjoint code generator enabling the reverse mode of automatic differentiation for the MATLAB language is an important step in the furthering of more wide-spread application of automatic differentiation to optimization and other problems were derivatives are required.

One serious challenge is the huge amount of storage that is required for the reverse mode, in the form of a stack. Here however, given the historic, recent and current technological developments, we can safely say that the time is working for the reverse mode. With both fast flash storage and ever faster data busses, and ever larger volatile RAM, the RM automatically becomes applicable to ever larger problems. Given that the basic typography of the memory hierarchy is likely to remain, it is still worthwhile to consider techniques to exploit the memory gaps at the different levels. Here we developed efficient techniques to handle the stack memory. The first solution is obviously to keep it all in RAM, and in the MATLAB interpreter, but when that is not sufficient, we can store the objects in the stack away in the farther, slower levels of the hierarchy, e.g. on disks. Given that the rate at which the date is produced and written by the AD process is generally much lower than the CPU throughput we can employ asynchronous writing and asynchronous prefetching before reads with much advantage and thus ideally avoid I/O wait times. This is then reflected in the fact that the measured AD time overhead reaches the vicinity of what is predicted by the theory [WBMB15].

In comparison to classical adjoint code, the generated adjoint code for MATLAB requires in several instances additional elements to ensure a correct operation in all cases. These are inserted for working around special cases in the program reversal, which generally concern the undoing of implicit structural changes in variables [WBB12], such as binary expansion and reshaping. While these additional elements are really only needed in some cases they must be inserted everywhere because it cannot be predicted where they are required due to the lack of type and shape information.

Another case were additional measures are required in the adjoint code for MATLAB are indexed expressions. Here, in the case of repeated integer values, MATLAB has different semantics in indexed expressions on the *left hand side* (LHS) and on the *right hand side* (RHS) of assignments. This makes it impossible to move an adjoint together with its index to the other side of an assignment in general. Fortunately this problem can be solved fairly efficiently by constructing the sparse binary matrices representing the corresponding partial derivatives of the index operations.

When we wish to apply differentiation to a non-analytic function in complex variables $c = (\Re c, \Im c)$, we can obtain both derivatives $\frac{\mathrm{d}f}{\mathrm{d}\Re c}$ and $\frac{\mathrm{d}f}{\mathrm{d}\Im c}$ by treating $\Re c$ and $\Im c$ as two separate variables, which doubles the expense in terms of numbers of directional derivatives. Interestingly, when we apply the RM we obtain both in one run as the RM returns in fact $(\frac{\mathrm{d}f}{\mathrm{d}\Re c}, -\frac{\mathrm{d}f}{\mathrm{d}\Im c})$. This surprising result has obvious applications in the field of complex optimization, but that is in fact much less spectectacular that it looks at first glance, since the reverse mode itself obtains the main advantage: in the case of long gradients, that is for the derivative of $f(c) \in \mathbb{R}$ w.r.t. $c \in \mathbb{C}^N$ we need $O(2N) = O(N)$ additional operations in FM versus $O(1)$ additional operations in RM, which are the same asymptotical overheads as in the real domain, in both FM and RM.

1.1 The reverse mode of automatic differentiation

The reverse mode of AD arises when the derivatives are accumulated in reverse direction of the original program flow in the form of *adjoints*, starting with the adjoint of the function result and working backwards until the adjoint of the function parameters are obtained [GW08a]. The advantage of this approach is that the computational expense depends linearly on the number of components M in the function results, that is, the time overhead of a reverse mode process over the original function is bound by $O(M)$. This contrasts with the corresponding bound of $O(N)$ in the forward mode of AD, where N is the number of parameter components. In particular, the RM can evaluate gradients of a scalar function with constant time overhead $O(1)$, independent of the length N of the gradient. When the forward and the reverse mode are combined to evaluate second order derivatives, a Hessian-vector product of a scalar function also has constant time overhead $O(1)$, while a full Hessian matrix can be obtained in time overhead $O(N)$. The same results in second order forward mode would require $O(N)$ and $O(N^2)$, resp.

The huge challenge to successfully apply the reverse mode is that it requires a large amount of memory, which is bound by $O(t_f)$, where t_f is the runtime of the original program. These unusual memory requirements stem from the fact that intermediate values in the program are required to evaluate the partial derivatives during the adjoint accumulation. However, the amount of memory required can be reduced drastically to $O(\log(t_f))$ using the techniques of *checkpointing* and *recomputation*. Also, software solutions such as asynchronous I/O can help to enhance the performance of the huge data movements.

The reverse mode was applied very early already to higher-level mathematical computer systems, like Maple [MR96]. The reverse mode is applied to C++ by the AD software ADOL-C [Cor+92; GJU96] and CppAD [BB08], which both perform operator overloading to capture the computations to differentiate. The reverse mode is also implemented by source transformation in the AD software Tapenade [Cou+03; HAP05], where so-called adjoint code is generated for Fortran and Fortran90 [PH05]. In ADiMat we also rely on source transformation, and hence an adjoint code generator is constructed to implement the reverse mode.

An interesting property of the reverse mode when applied to complex arithmetic is that the adjoints carry a different derivative value in the imaginary part than the derviatives resulting from the forward mode.

1.2 Scientific contributions of the author in this work

This dissertation contains both scientific contributions that have already been published in journal or conference papers and as of yet unpublished results:

1. The new user interface for ADiMat, such as the driver functions **admDiffFor**, **admDiffRev**, and **admHessian**, which automate the code generation and invocation, was published in [WBB14]. This is contained in abbreviated form in Chapter 2 of this work

2. The handling of dynamic reshaping and scalar expansion in the adjoint code was published in [WBB12] and is summarized in this work in Chapter 3

3. RIOS, the efficient asynchronous prefetching I/O subsystem used to store the stack data produced by the adjoint code was published in [WBMB15], which constitutes the bulk of the content of Chapter 4 in this work

4. ADiMat was applied to several projects with the authors help, including one using exact Jacobians in an implicit Newton method for solving multiphase flow in porous media [Büs+14] and the sensitivity analysis of a force and microstructure model for plate rolling [Seu+12; Seu+13]

5. The derivatives of the **legendre** builtin function where derived and implemented into ADiMat in the context of work on estimating the expansion coefficients of a geomagnetic field model [BW18], briefly referred to in this work in Section 5.2

6. Work on automatic differentiation of ODE integration was done by the author and published as a preprint [Wil18]

7. The correct handling of repeated integer indices in index expression can be achieved by constructing the partial derivatives of these operations, which is efficiently possible by performing a mock index operation on suitable adjuvant objects. This is a new unpublished contribution of this work, in Section 3.3

8. Expanding on the idea of constructing the partial derivatives of index expressions, we devise a method to construct those of the **kron** multiplicative operator, as described in Subsection 5.3.4, which is previously unpublished

9. For the multiplicative operator **conv** we derive the adjoint expression, which is a convolution with the reversed filter. While an equivalent technique is evidently used in the backpropagation procedure of convolutional neural networks, we could not find a description of the mathematical derivation and hence publish our method here in Subsection 5.3.3

10. Another unpublished contribution of this work is the Chapter 7 dealing with the peculiarities of complex-valued arithmetic in the reverse mode of AD, in particular explaining why forward and reverse mode may yield different results when a complex-valued computation is not analytic, that is, not complex differentiable

1.3 Structure of this work

This work is structured as follows: In Section 2 we introduce ADiMat and present its functionality, its user interface and how it is organized internally. In Section 3 we discuss peculiarities of the adjoint code generator for MATLAB and the measures that we take to cover the many challenges that arise from the interpreted nature of the language. In Section 4 we present work done for the efficient handling of the large scale stack data sets that may be produced by runs of adjoint code. In Section 5 we discuss for some selected builtins how the correct derivative propagation rules were derived and implemented in ADiMat, as well as some general techniques that are available for that purpose. In Section 6 we present the architecture of the adjoint code generator which is implemented in XSLT and thus explores new avenues in compiler construction. In Section 7 we present our results regarding the different values that are obtained in complex-valued arithmetic from the derivatives in forward mode of AD versus the adjoints produced by the reverse mode. We finish this work with some conclusions and an outlook in Section 8.

2 ADiMat

The main development goal of ADiMat is the provision of AD for the MATLAB language. Today this is a complex system of two MATLAB source code transformations for the forward and reverse mode of AD and several so called *derivative classes* with a user friendly interface that hides many of the complexities of the AD evaluation process. For second and higher order derivatives, in particular for Hessians evaluated in forward-over-reverse mode, ADiMat also has a classical derivative propagation class with overloaded operators. A special feature for the reverse mode is the provision of several high-performance I/O functions used for the numerous push and pop operations arising in the reverse mode evaluation.

ADiMat was developed at RWTH Aachen University and TU Darmstadt. Development was started by André Vehreschild in 2006 and continued until 2010 at RWTH Aachen University. This work encompasses roughly speaking the forward mode implementation in use until now, including the parser for the MATLAB language [Veh09]. Development continued with the author beginning work on the reverse mode in 2008 at RWTH Aachen University and continued until 2013 at TU Darmstadt. Some additional features, such as the fully correct implementation of complex valued derivatives in reverse mode, the handling of repeated indices in index expressions in reverse mode,

and the addition of derivate propagation rules for several builtins, such as **legendre**, **conv**, and **kron** amont others, were added over time in the years leading up to the present, by the author.

The user interface for ADiMat has been explicitely designed to maximize the ease of use. Basically the user can provide seed matrices and is presented with Jacobian and Hessians, as described in the AD literature, while compression is handled transparently when the non-zero pattern of the Jacobian is provided [WBB14]. The user interface also delegates the actual source code transformation to the ADiMat transformation server, automatically and transparently sending the code to be differentiated to the server and receiving the differentiated code in return. For maximum efficiency, the differentiated code can still be invoked manually of course.

In the following subsections the high-level driver functions are mentioned alongside the more technical description of the underlying AD process, after a short introduction to AD and ADiMat in general, and a non-exhaustive list of use cases found in the literature.

2.1 What AD can do and where AD is employed

Automatic differentiation (AD) can be employed wherever derivatives of numerical functions and algorithms are required. In particular, derivatives are in practice often approximated by finite differences, which has the great advantage that just the function definition and implementation is required. However, since finite differences are by necessity accurate to at most half the number of available digits only [Wik20e], it is almost always much better to use the fully accurate derivatives. These may be obtained with analytical differentiation, by the complex variable method in case of a real analytic function [LM67], by evaluating the adjoint partial diffential equation in case of a partial diffential equation (PDE), or with AD. Numerous studies have been conducted to investigate the advantages that can be gained from using correct, fully accurate, derivatives [MC05].

An particularly interesting field are adjoint approaches, that is, the reverse mode of AD, which can compute very long gradients in a runtime that is a small multiple of the original function runtime. This has been used in large scale shape optimization [GKS05] and in recent times in deep learning, where the famous backpropagation algorithm is just the special case of the reverse mode applied to a neural network [GBC16].

Since the reverse mode has no equivalent method finite differences, the reverse mode is a sine qua no to obtain such efficient gradients. There is however one exception: when solving a partial differential equation (PDE), it is possible to set up the adjoint PDE and solve that to obtain the gradient. This approach has allowed large scale shape optimization in aerospace [Reu+96; NJ01] and in electromagnetic design [LK+13] and also data assimilation used in climate and weather modeling [Li+93; HHG05].

While this approach is often not trivial, given that the adjoint equation must first be found and that it may then require different methods for the solution than the original PDE, the result is equivalent to the reverse mode gradient. For this reason, the reverse mode is also called *adjoint mode*, and the resulting gradient is called a *discrete adjoint* [Nad03], given that in a PDE context it will be obtained by differentiation through the discretization that is used to approximate the PDE solution.

Also in this case the relative merits and advantages of using AD gradients versus adjoint solutions have been investigated [Cou+03; PZG05; Rac+18; CM05], also for example by the author in a work on discrete and continuous adjoint approaches to estimate boundary heat fluxes in falling films [Büc+10], where the DROPS [Ber+10; GR13] numerical package was differentiated with ADOL-C [Gri+99] in reverse mode and a reversal scheme using the checkpointing tool `revolve` [GW00] was used to evaluate the gradient of a Poisson PDE very efficiently. Note that in this case of the Poisson equation the PDE is self-adjoint, that is, the adjoint PDE is identical to the original PDE.

2.2 The design and development of ADiMat

The first beginnings of ADimat where the idea of combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs [Bis+02]. This so called hyprid approach allows that a single version of the differentiated code can be invoked either with native data types for the derivatives to obtain the scalar mode of AD, or with special overloaded data types to obtain the vector mode. In comparison with a classical AD approach using an overloaded data type on the main advantage of the hybrid approach is that the set of operations that the overloaded type must support is defined by the generated code, and hence much more limited in scope. Over the years the system was completed to a full implementation of the forward mode of automatic differentiation for MATLAB programs [BLV03]. A system for interfacing MATLAB with external software geared toward automatic differentiation was devised [BEV06]. Special cases, such as coping with a variable number of arguments when transforming MATLAB programs where covered by dedicated approaches [BV08]. Code optimization techniques in source transformations for interpreted languages, including constant folding, loop unrolling, constant propagation, forward substitution, and common subexpression elimination, also in the case of ADiCape, where explored [BPV08]. Finally, in his PhD thesis André Vehreschild detailed his work on the design and implementation of ADiMat [Veh09].

The author took first steps generating adjoint expressions for Matlab [Wil10] and developed a new user interface for ADiMat [WBB14]. It turned out the impact of dynamic data reshaping on adjoint code generation for weakly-typed languages such as Matlab has to be handled by dedicated techniques [WBB12], as well as the adjoint transformation of binary MATLAB operators [WB10]. The stack which is required for data storage in the reverse mode was handled by a dedicated software layer called RIOS, which facilitates efficient I/O in reverse direction [WBMB15]. The development was continued with work on source transformation for the optimized utilization of the MATLAB runtime system for automatic differentiation [HWB15], and with a work on estimating the expansion coefficients of a geomagnetic field model using first-order derivatives of associated Legendre functions [BW18]. The ADiMat system is documented in an extensive electronic handbook [VW13].

2.3 Releated work

Related work on AD in general are numerous, and on AD for MATLAB in particular we would like to mention the first work on source transformation for MATLAB automatic Differentiation which is implemented by the tool MSAD [KF06], also described in a PhD thesis [Kha12]. `TOMLAB/MAD` is an efficient overloaded implementation of forward mode automatic differentiation in MATLAB [For06]. `CasADi` is a symbolic package for automatic differentiation and optimal control [AÅD12]. `ADiGator` is a toolbox for the algorithmic differentiation of mathematical functions in MATLAB using source transformation via operator overloading [WR17] based on an efficient overloaded method for computing derivatives of mathematical functions in MATLAB [PWR13]. Other work has also been done on automatic differentiation with MATLAB object-oriented programming [Nei10].

2.3.1 Other languages

Nowadays there are AD tools for almost any programming language used in a numerical or technical context, so it is not sensible to list them all here. By our experience, the best and the most predictable performance can be obtained from using AD on low level languages, for example `ADiFor` for Fortran [Bis+92a; Bis+96; BG92], `Tapenade` [HAP05] for Fortran and C [PH08], `ADOL-C` [Gri+99] or `CppAD` [BB08] for C and C++, `TAF` for Fortran and `TAMC` for C [GK03], `OpenAD/F` for Fortran [Utk+08a], or the NAGWare Fortran compiler [NR06]. These tools are also very stable and cover their languages almost completely.

For interpreted languages, which are understandably highly popular due to the much reduced development effort, the maturity and the performance of the available AD tools tends to vary a

lot, from one tool to the other, but also from one problem instance to the next. Even for ADiMat, while there are numerous cases were it was shown to produce correct results efficiently, there are also many others were the runtime was found to be quite large, for example. There resons are often found in inadequate vectorization of the original function, but may also be down to other reasons in ADiMat itself. Moreover, ADiMat does still not completely cover the MATLAB language, in the sense that there are many toolbox functions that are not supported yet. In every such case a detailed investigation will determinine the root cause, so ADiMat is still a work in progress and user feedback and cooperation with the developers is required for further refinement of the software. The situation with regards to other AD tools for high-level languages such as MATLAB, Python, R, or Julia is probably very similar.

As an example, in a study on nonlinear model predictive control using decoupled ab net formulation for carbon capture systems-comparison with algorithmic differentiation approach [MMZ18] discuss two alternative control formulations, one using derivatives and another linear one. In performance results the derivative based approach is slower but arguably more exact and responsive in the control test. With regards to the derivative, a hand code derivative is predictably 3-8 times faster than the derivative evaluated with ADiMat. This would be the factor in computational effort that one has to take into account for the price of saving on programming effort by just using AD to compute the derivative. This factor is in our view not particularly bad at all, but rather surprisingly good.

Recently, a survey of AD tools has been conducted recently with regards to machine learning [Bay+17], which tabulates many languages and the available AD tools, so we will not endeavour to list all the available AD tools here. A new addition to the list is the authors software `R/ADR` which translates the basic design ideas of ADiMat to the language R [Wil20b].

2.3.2 Software that incorporates AD

Many high-level modelling and optimization frameworks use AD internally. The approach is usally that the problem is formulated by the user in some abstract form, like a domain specific language, and the framework will then automatically generate the appropriate code to evaluate the forward function and the derivatives. Examples are AMPL [Gay91; Gay96; Gay15] and the AD Model Builder, which uses automatic differentiation for statistical inference of highly parameterized complex nonlinear models doi:10.1080/10556788.2011.597854. Also, all the available deep learning and AI frameworks such as `TensorFlow` or `PyTorch` incorporate AD since the backpropagation algorithm is a special case of the reverse mode.

2.3.3 ADiMat namesakes

We shall also not fail to mention two further software projects which are also called ADIMAT, but have no relation to automatic differentiation. The first is the finite element software ADIMAT [Bat77], used in experimental and mathematical investigation of response characteristics and aging phenomena in safety fuse elements [Hof87] and in a thermoelastic hydrodynamics analysis of EMP-segments of thrust bearing [Zho96]. The second is a phase diagram assistant software also called ADIMAT [ELN06].

2.4 ADiMat use cases

Over the years, ADiMat has found application in numerous scenarios. One where the author has been involved is the sensitivity analysis of a force and microstructure model for plate rolling [Seu+12; Seu+13], another one is using exact Jacobians in an implicit Newton method for solving multiphase flow in porous media [Büs+14].

When conducting a stiffness analysis of cardiac electrophysiological models, the authors use ADiMat to examine the eigenvalues of the Jacobian of a variety of cardiac electrophysiological models [SD10]. In the work on a generic approach for the solution of nonlinear residual equations, sensitivity computations are conducted using a combination ADiMat with another AD tool called

Diamant [LCD11]. Investigating the question of whether to trust derivatives or differences, the authors use ADiMat to compute the derivatives for their comparisons [MW14]. In a PhD thesis from the University of Dortmund, ADiMat is used in the optimal control of the relativistic Maxwell-Newton-Lorentz equations [Tho15]. ADiMat has been used in an investigation on derivatives in time and frequency domains [BW17]. In a work on gradient- and Hessian-enhanced least square support vector regression(LSSVR) the authors show that the incorporation of Hessian information, obtained with ADiMat and Tapenade, into LSSVR models has great advantages [JZ18]. In the work on preconditioning jacobian systems by superimposing diagonal blocks the authors demonstrate their results with numerical experiments using ADiMat [RB20]. The differentiation of ODEs has been explored using ADiMat [Wil18].

In a benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning [SKF18] conducted, ADiMat completes most of the tests and is even head-to-head with a manual derivative implementation in one problem instance, for the largest problem size.

2.5 Derivative classes

The task of the derivative class is described in [Veh09]. They are the means to facilitate the vector mode in ADiMat, in a so-called hybrid approach of source-transformation and operator-overloaded [Bis+02]. While the generated source code provides scalar mode AD when run with native data types, vector mode is obtained by running the same code with the derivative types being instances of one of the derivative classes. There are two main types of derivative classes. First we have a derivative class based on cell arrays, where internally the multiple derivative directions of `dx` are stored as individual arrays arranged in a cell array, and second, an array-based derivative class where internally the multiple derivative directions of `dx` are stored in a single large array of dimensions `[ndd, sz]`, where `ndd` is the *number of directional derivatives* and `sz` is the size vector of the array in question.

The relative performance of these approaches is of course quite relevant for the overall performance of the code in vector mode. From a small benchmark we obtain the results shown in Figure 1 for the cell based class and those in Figure 2 for the array based class.

The performace of any operation on the cell based class is quite uniform. This is of course due to the fact that any operation boils down to a loop over the internal cell array. The same operations on the array class are sometimes very efficient when they can be performed as a single operation on the internal array. The binary arithmetic operators for example can be handled by the builtin **bsxfun** very effectively. The many well-known vector operations, like **sum**, **prod**, **fft**, etc. and **bsxfun** itself can trivially be handled with a single call processing the entire internal array. While it is not vectorized itself, **conv** can be implemented via **conv2**.

Interestingly, the matrix multiplication is asymmetric in that regard. Depending on the internal layout, whether we use `[ndd, sz]` or `[sz, ndd]` as the dimension of the internal array, a left matrix multiplication can be handled with a single matrix multiplication internally while a right matrix multiplication requires costly data rearrangements with **permute** or **kron**, or vice versa [Veh09]. However, the two operations where the array is reshaped to a vector and multiplied with a matrix from either left or right are both very efficient, which is crucial as these are the staple of the standard FM and RM propagation, where the directional derivatives are multiplied with the the Jacobian matrices of elementary operations.

While the array based derivative class is often more efficient, the implementation is also more complex. For any operation on the derivatives in the derivative code, the array based derivative class must have a dedicated method that handles the case efficiently. In the generated code we often use the method `call` as a wrapper to the method call, so the emitted code might be for example `call(@mean, d_x)`. Then, the derivative classes have the method `call` which applies the handle to each derivative direction in a loop. For the cell based class this is the best that we can to anyway, so the cell based class does not require that we actually implement the method `mean`. This generic method `call` is rather inefficient however in the case of the array based class, even more so than the cell array based derivative class, as the internal array has to processed in

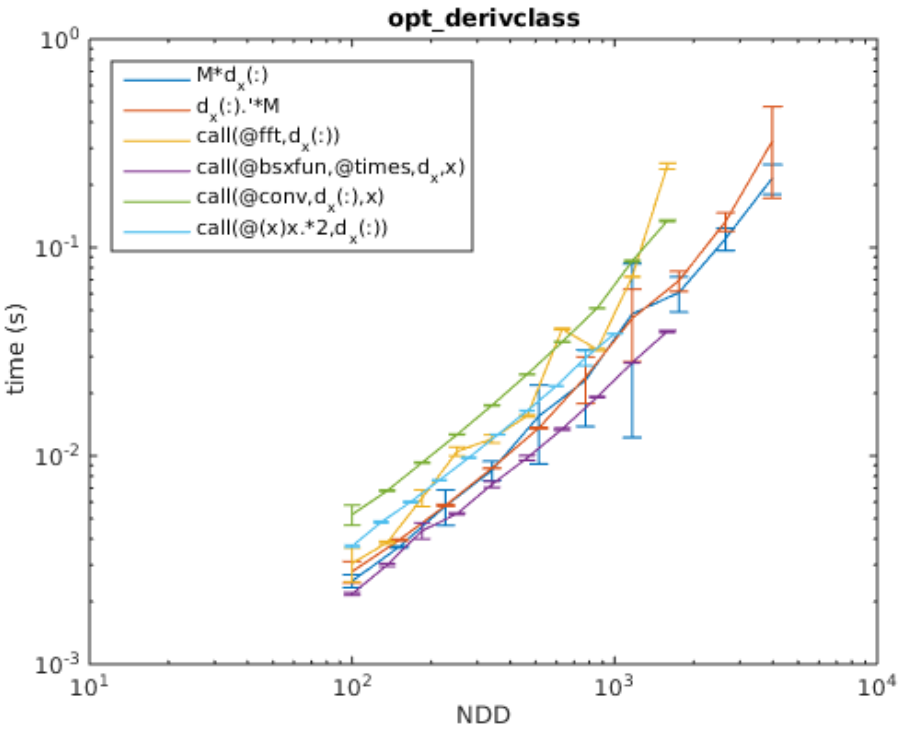

Figure 1: Performance benchmark of the cell based derivative class for selected operations.

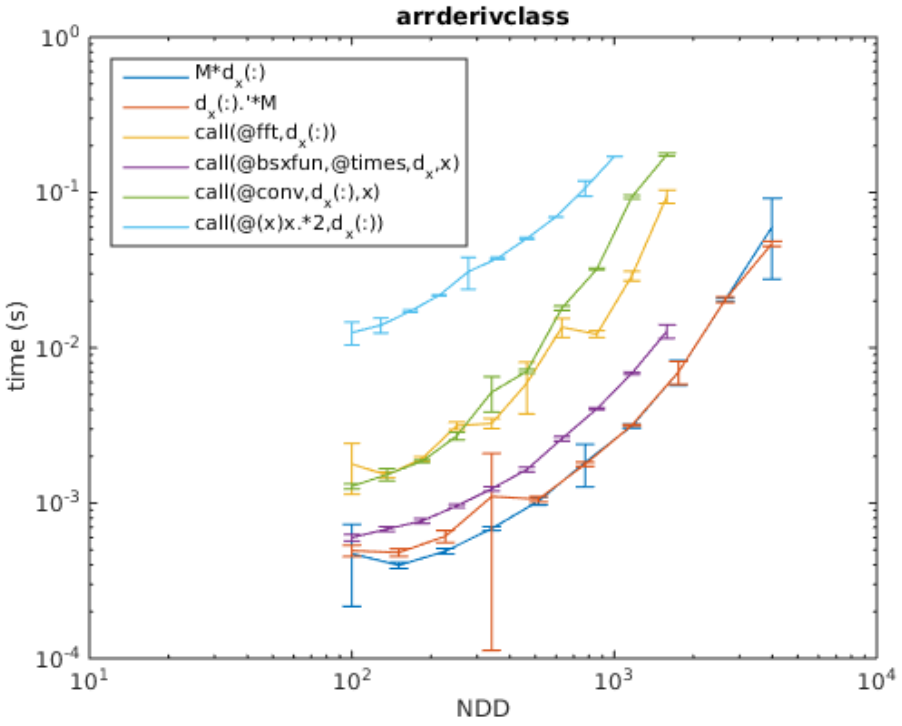

Figure 2: Performance benchmark of the array based derivative class for selected operations.

slices. Hence it is important that for any method we find a way to perform the operation without am explicit loop. The method `call` of the array-based class was constructed to search for and dispatch to a given method that it has. Continuing the example, in the expression `call(@mean, d_x)` the `call` method will automatically delegate to the method `mean` once we add that to the class, and hence this wrapper expression can be used with little penalty in derivative code to keep it generic.

The performance results provide directions for the relative merits of the arithmetic or structural derivative propagation techniques, as discussed later also in Section 5.1. For example in the derivative code the expression `mean(x, k)` might be differentiated structurally to `mean(d_x, k)`. This would then require that a method `mean` is added to the derivative classes, which in the case of the array based class would boil down to evaluate `mean(d_x.derivs, k+1)` on the internal array `d_x.derivs` and can thus be expected to be efficient. When we can construct the partial derivative as a sparse matrix this is the ideal case for the array based class as the reshape-to-vector-and-matrix-multiply operation is by far the fastest of the ones benchmarked.

2.6 Forward mode source transformation

The forward mode source transformation is well described in the dissertation thesis of André Vehreschild [Veh09]. The concept for the generated code is similar to the forward mode code generated for example by ADiFor [Bis+92b] or Tapenade [PH05] for Fortran code.

The transformation uses activity analysis depending on the independent and dependent variables to determine the set of variables that actually require differentiation. Derivative variables are created for those and associated by name [GW08a]. ADiMat typically uses the prefix `g_`.

A novel idea idea is to generate code for the scalar mode only, with the golden rule that a derivative variable `g_x` has the same type and shape as the original program variable `x` it is associated with.

Any work required for the vector mode is then delegated to the derivative classes, described in the previous Section 2.5. Accordingly the interface is clear: a derivative class object `g_x` has to appear as an array of the same size and shape as the associated program variable `x`, and hide the multiple directional derivatives internally. This combination of source transformation and operator overloading in AD is also called a hybrid mode [Bis+02].

For each active function `f` a differentiated version `g_f` is generated, which can then be called as required, either from other differentiated functions or as the top-level function by the user.

The forward code transformation uses a database to store derivative propagation rules for the spported set of builtins, as described in [BBV05]. This macro language also allows users to create custom derivatives by code directives.

Work was also done to handle variable argument lists and the special builtins **varargin**, **nargin**, **varargout**, and **nargout** [BV08].

A new easy-to use interface has been added by the author, facilitating the generation and invocation of the differentiated code [WBB14]. This interface consists of the single driver function **admDiffFor**, which differentiates the source code if necessary given the list of independent function parameters and dependent function results, invokes the differentiated code appropriately given the seed matrix S and returns the resulting Jacobian J matrix or Jacobian product $J \cdot S$ in the form of a matrix.

This high-level driver is not necessarily the right choice when the run of the actual AD code is short and the derivative is needed repeatedly in a tight loop. A typical example is the ODE integration functions like **ode15s**. Here, the kernel function is often rather small and of short runtime. The various sanity checks and the code generation performed by the drivers can be turned of via certain options. While the differentiated code can also be called manually, of course, this requires advanced knowledge of both AD in general and ADiMat in particular.

2.7 Reverse mode source transformation

The reverse mode source transformation is the main subject of this work and hence described in detail in subsequent chapters.

In one slight modification over the forward mode transformation, program and derivative variables may also be of type struct of cell.

On the other hand variable argument lists are not supported and some other semantical structures are also not allowed, the most prominent possibly being the **return** statement. It is currently ignored so it can still be used for handling error conditions which never happen.

Otherwise all control flow statements are supported, these are **if**, **switch**, **for**, **while**, **continue** and **break**. The declarations **global** and **persistent** are also supported.

The **parfor** control flow statement is rudimentarily supported but there are subtle open issues regarding the performance of the adjoint accumulation. Exception handling with **try** and **except** is not supported.

Like in the forward mode code transformation the code generation is performed directly by traversing the AST, not on a basic block representation of the code. This has appearently only two minor drawbacks in quality of the generated code, namely that the **continue** and **break** statements can induce repetitive sections of the transformed loop body code, which could in theory lead to combinatorial explosion with multiple nested loops each containing these statements.

The reverse mode transformation uses the same activity analysis as the forward mode transformation. It does not have a to-be-recorded analysis like Tapenade does [HAP05; Nau02].

The interesting feature of this particular reverse mode code generator is that it is implemented almost entirely in XSLT, with XML as the state or AST representation. The actual transformation by a sequence of XSLT processing steps is described in more detail in a later Chapter 6. The initial XML is obtained from the C++ core of ADiMat after the parsing and initial analysis step. To this end a single overloaded method `outputXML` is added to the classes representing the various AST nodes, which traverses the AST and prints XML markup along the way, thus directly rendering the AST structure in an XML document. Some side information, in particular the variable dependency and the function call graphs, is also included in the XML. The XML dialect used to represent the ADiMat AST in XML is a custom develepment called `AST XML` which is described in section 6.5.

The high level driver function for the reverse mode is called **admDiffRev**, which has the same interface as **admDiffFor**, with the corresponding difference that the product $S \cdot J$ of the seed matrix S and Jacobian J is returned, as is natural for the reverse mode.

2.8 The ADiMat transformation server

When the new high-level user interface was added to ADiMat, the system was also split in two components: the transformation server and the runtime environment. The transformation server is available as a web service at `http://adimat.sc.informatik.tu-darmstadt.de`. The user interface drivers send the user code to the transformation server and the differentiated code is sent back to the user. The differentiated code is stored locally and the user interface drivers automatically check the file times of the users code files to determine whether the differentiated code must be refreshed due to a change in the user code. These checks can be turned off for performance reasons, which is advisable when repeatedly evaluating the same derivative function in a loop.

The separation of the runtime environment and the transformation server has the advantage that most compiled code needs to run on the server only, while the runtime environment consists of MATLAB code only. This greatly facilitates the provision of ADiMat for different operating systems and hardware architectures. The ADiMat server was first set up in 2011, thus making it a pioneer in the buoyant area of *Software as a service* (SaaS) [Wik20f]. Most of the well-known advantages of this concept apply to ADiMat as well. In particular, many updates require a change to the server only, which is then immediately available to all users.

2.9 Stacks for the reverse mode

Basically, there are several variants of the functions **adimat_push** and **adimat_pop**, each one called a *stack*. The default stack uses a persistent variable to store the stack in memory, in particular in the working set of the MATLAB interpreter. Other stacks write the objects to the harddisk, and some of these doing so asynchronously, using multithreading. This is described in more detail in a later Chapter 4.

2.10 Alternative derivative evaluations

For convenience, comparison and reference the two driver functions **admDiffFD** and **admDiffComplex** are provided to evaluate derivatives numerically based on finite differences and the complex step method [LM67], respectively. By providing the same interface as the driver for the forward mode, AD results can easily be verified against alternative methods [WBB14].

2.11 Taylor propagation

Rules for the propagation of truncated Taylor series where established very early in dervelopment of AD already [BWZ70; Ral81], first for source transformation and then also using operator overloading [CR84; BCG93]. Usually univariate Taylor series are propagated since any mixed derivatives can be obtained by intricate interpolations [BCG93; GW08a]. Later works also regarded the application to MATLAB, including the propagation of multivariable Taylor series [Nei10].

ADiMat also provides a very similarly constructed operator overloading Taylor mode class @tseries2, which can propagate truncated univariate Taylor series of arbitrary order. The main purpose is the computation of Hessians in forward-over-reverse mode, as described in the following Section 2.12.

The set of builtins that are supported is somewhat more limited than those that are handled by the source transformations of ADiMat. On the other hand, it is relatively easy for users to add support for a certain builtin, since the Taylor classes are part of the runtime environment of ADiMat, written in MATLAB, and open source. In each case a method has to be added to the Taylor mode class implementing the correct rules.

Currently the Taylor mode class of ADiMat has many methods which implement only the first order derivative propagation and otherwise raise an error. This is the case because the evaluation of Hessians in forward-over-reverse mode requires only the first order derivative to be provided by the Taylor class. For example, although in ADiMat we currently implement only first order derivatives of the builtins **eig** and **eigs**, in any of the differentiation modes, we can obtain Hessians of eigenvalues in forward-over-reverse mode.

Internally there are actually two Taylor mode classes: one for the scalar mode and one for the vector mode. The vector mode version internally uses one of the derivative classes described in Section 2.5 to propagate multiple derivative directions simultaneously.

The generic user frontend for Taylor propagation in ADiMat is the function **admTaylorFor**. The general forward-over-reverse mode is accessible via **admTaylorRev**, where the adjoint code is generated as for **admDiffRev** and then run with @tseries2 objects. This is exactly the same thing as is done for Hessians, the only difference is that the truncation order of the Taylor series can be set arbitrarily >= 1. The adjoint outputs are the derivatives of the Taylor series of the function result variables w.r.t. all the inputs, as described in more detail in the AD literature [GW08a].

2.12 Hessian evaluation

The most effective method of Hessian evaluation is the forward-over-reverse mode, which runs the adjoint code with both the active input arguments and the adjoint input arguments being objects of the Taylor class `@tseries2` limited to first order. As a result we obtain second order derivatives from the adjoint output arguments. The full Hessian requires $O(n)$ in time overhead and the usual

$O(t)$ in space overhead of the RM, by a single run of the adjoint code with the Taylor class with nested derivative class. A Hessian-vector product requires $O(1)$ in time overhead, and again the usual $O(t)$ in space overhead of the RM, by a single run of the adjoint code with the Taylor class in scalar mode. There are also other options, which all boil down to twice differentiating in FM, resulting in $O(n^2)$ time overhead [Wil13a].

In forward-over-reverse mode, Hessians or Hessian-vector-products of multiple function outputs, in the case of a non-scalar function are evaluated using multiple runs of the adjoint code. More precisely, linear combinations of the multiple Hessians or Hessian-vector-products are evaluated as per the rows of the seed matrix U. One run of the adjoint code is required for each row in U.

The set of builtins supported by the forward-over-reverse mode is the intersection of the set of support of the Taylor mode class and that of the adjoint code generator. Furthermore, the Taylor mode class has to support all the calls in the structural reverse mode propagation rules, i.e. the runtime functions invoked by the generated RM code.

The user interface for the forward-over-reverse mode Hessians is **admHessian**. It accepts as the second argument a cell array with three matrixes U, V and W and returns a linear combination with weights u_k, $1 \leq k \leq M$ of the $H_k \cdot W$ for each row $\mathbf{u}_j$, $1 \leq j \leq R$ in U, where H_k is the Hessian of the k-th function output. More precisely, the computational overhead of **admHessian** is $O(R \cdot Q)$, where R is the number of rows in U and Q is the number of columns in W.

2.12.1 Alternative Hessian evaluation modes

For convenience, comparison and reference the two driver functions **admHessFD** and **admHessFor** are provided. Both are based on second order forward derivatives together with the well-known interpolation rules [BCG93], which can also be generalized to arbitrary orders [GUW00; GW08b], **admHessFD** using second order finite differences and **admHessFor** using second order Taylor series. They simultanously compute all the products $V \cdot H_k \cdot W$, where H_k is the Hessian of the k-th function output.

admHessFor2 has the same interface but uses second order forward mode code generated by double application of the forward mode code generator [Veh09].

All second order forward based methods require $O(PQ)$ time and space overhead where P and Q are the outer dimensions of V and W, resp.

2.12.2 Hessian of Lagrangian

The forward-over-reverse mode provides a different form of seeding compared to the second-order forward-mode. This can be used to to evaluate the so-called *Hessian of the Lagrangian*, which is frequently required in optimization routines and solvers using constraints [BGN00; BHN99], such as **fmincon** [Mat13]. While the Lagriangian function has multiple outputs the solvers require a weighted sum of their Hessians. It be computed with a single run of the adjoint code, and hence in $O(n)$ time overhead, for it is a linear combination of the Hessians of the outputs of the Lagrangian function.

Example: When we have the function **lagrange** returning the Lagrangians we can invoke `admHessian(@lagrange, {lambda(:).', 1, 1}, x)` with the vector of weights `lambda`. To the same effect, one can also compute the **sum** of the Lagrangians weighted with `lambda`, and compute the Hessian of the resulting scalar function. This example demonstrates the purpose of the adjoint seed matrix U in the Hessian evaluation, which is set to the vector `lambda` here.

3 Adjoint code generator techniques

For the most parts the adjoint code generated by ADiMat from MATLAB programs is very similar to the established procedures [GW08a], such as has been sucessfully implemented for the Fortran language in the Tapenade AD source transformation [PH05]. Hence we will concentrate on the differences that arise from the particularities of MATLAB in this section.

The adjoint code generator by default will generate a store-all or record-all version of the adjoint code. This means the code is run from start to end in a so called *recording sweep*, followed by the *reverse sweep* of the entire program. However, at any function call the alternative *adjoint* mode may be requested instead, which consists of a regular function invocation in the recording sweep while a recording sweep followed by a reverse sweep is performed at that point in the upper level reverse sweep. This is done by means of a code directive, and is a very effective measure to cut down on the size of the stack, as described in the literature and also implemented by Tapenade.

The *data model* that ADiMat uses in the RM code generator is slightly expanded compared to the FM. In particular, structs and cell arrays can be used as active variables. The corresponding task in the adjoint code is very generally to undo any structural change to the program variables, such that at any point the program variables and adjoint variables have the same data type. For example, when a struct is created in the program, in the adjoint code at this point the adjoint struct is picked apart and the field values are extracted, etc. This is described in more detail in Section 3.1.

In all sorts of operations and builtin functions such structural changes may occur implicitely, such as automatic scalar expansion done by the binary arithmetic operators and some binary builtin functions, described in Section 3.2, and array resizing and reshaping in index expressions, described in Section 3.3. The techniques to handle these implicit changes are described already in [WBB12]. However, with regards to the index expressions, a particularly interesting case is when repeated integer indices occur in the index. In these instances the structural propagation with undoing of implicit reshapes as proposed in [WBB12] is not enough and will fail. A better solution was only invented recently: By setting up the partial derivatives of the operation as a sparse matrix we can easily handle all possible cases and at the same time achieve a much better performance. This is described in more detail in Section 3.3 as well.

In all of these cases there is a considerable difference in the amount of special care that must be taken in the reverse mode compared to the forward mode. The main reason is that in forward mode many operations are handled by structural propagation (cf. Section 5.1.2), and differentiate to themselves, such as **times**, **mtimes**, **fft**, etc. Hence, the same implicit behaviour is automatically applied to the forward derivatives, so very often no special measures are required. In the reverse mode however we have to undo such implicit behaviour when we propagate the adjoints. For example, consider that the **fft** builtin may pad the input with zeros up to a given length or ignore parts of the input data, and return a correspondingly shaped FFT. In the FM the same padding is done on the derivatives, when we just call **fft** on the derivative variable with the same parameters. In the RM however, we have to manually undo the padding on the adjoint variable, or add padding in the case where the input data is used only partially.

The additional code inserted into the adjoint code to undo possible implicit changes comes at a cost in terms of performance. Hence, the adjoint code generator can be instructed by several options to not emit these parts of the code. This is used for the differentiation of the replacement functions where ADiMat uses algorithmic propagation (cf. Section 5.1.3), while the implementation of the options is facilitated by the use of abstract AST elements in the initial form of the adjoint code (cf. Section 6.5.3). These relations are described in more detail in the Section 3.4.

There is one case of implicit data type change that has to be handle differently from those mention above. This is the expansion from real to complex numbers. It can occur when the square root of a negative number is computed or when a real variable is combined with a complex number. In this case it would be wrong to undo the type change by coercing the adjoint to a real value at that point. A simple counterexample is presented in Section 3.5, while the underlying mathematical reasons are discussed in the later Chapter 7.

3.1 Data model and structural manipulations

The *data model* for the legal values of program variables for the purpose of the reverse mode is the set of all double numerical arrays plus struct and cell data structures. The associated adjoint variables will always have the exact same structure, with the numerical arrays being derivative classes in vector mode and double arrays in scalar mode. Accordingly all sorts of data

rearrangement and management such as creation of fields, cells, array reshaping and resizing are also undone on the adjoint variables during the reverse sweep.

In the adjoint code generator we use the term *storage expression* to refer to subtrees of expressions that denote struct, cell array, or array references. Each storage expression has a *principal variable*, which is the variable in the workspace that holds the data structure (cf. Section 6.5.1). Given that the data structure of program and adjoint variables are identical at any time, a storage expression is differentiated by differentiating the principal variable only.

Thus, more generally speaking, storage expressions are handled by structural propagation, and they differentiate to themselves. There is however one important exception to this rule: index expressions with repeated indices require that we use arithmetic propagation for index expressions instead, as described in the following Section 3.3.

Another closely related but mathematically entirely different question is the topic of complex arithmetic and the automatic expasion from real to complex values, which occurs for example when the square root of a negative value is computed. Here undoing the expansion at the point of occurence would be the entirely wrong thing to do, cf. Section 3.5, and the later Chapter 7.

3.2 Binary scalar expansion

In the case of binary scalar expansion (BSX) we need special measures in the RM while the semantics work out automatically in the FM. Basically, these special measures are a runtime function which needs to be invoked on every binary operator. It has to check whether scalar expansion has occured in the operation and when that is the case the adjoint in question has to be summed to a scalar value, as described in [WBB12] and similar to what was done before for the vector operations of Fortran95 [PH05].

3.2.1 Generalized binary scalar expansion

The builtin **bsxfun** is not supported by ADiMat yet. However, it is implemented as a method of the derivative classes already, see Section 2.5, which is useful when handling the derivatives of the **legendre** builtin, see Section 5.2.

3.2.2 Automatic generalized binary scalar expansion in Octave

In recent versions of Octave the expansion rules of **bsxfun** are active for all binary operators automatically, which is called *broadcasting* [oct12]. This is an extension of the semantics of the MATLAB language which is not as of yet followed by MathWorks MATLAB. It is also not supported by ADiMat yet.

3.3 Array selections: indexed expressions and assignments

Array reshaping (AR) may occur during an indexed assignment. Here MATLAB may fill a left hand value with a right hand side of any shape, as long as the number of components is the same. Array reshaping (AR) may also occur when an indexed expression is evaluated. Here, the resulting shape may depend on the shape of the index array used, for example. In both cases this reshape has to be undone, as described in [WBB12].

It was discovered relatively early that the proposed solution was not entirely correct, and would not work in one quite particular instance, namely the repeated occurence of integer index components. This is ultimately due to the fact that the semantics of array selections in the case of repeated indices are somewhat different depending on whether the expression is on the LHS or the RHS.

For example, consider the following statements in MATLAB:

```
x = 5:8
x([1 3 1])
x([1 3 1]) = 1:3
```

```
x =
  5  6  7  8
ans =
  5  7  5
x =
  3  6  2  8
```

Now, when we use structural propagation on index expressions, an index expression will shift from the LHS to the RHS of an assignment in the adjoint code and vice versa, and this will thus fail in the case of repeated indices, due to the different semantics. While permutations and one-to-one selections, including with logical indices, and also array reshapes and array enlargements on the LHS are all covered by the old approach [WBB12], the adjoint code proposed there fails to produce correct results whenever repeated indices occur.

So, again we need to take special measures in the reverse mode, while in the FM we can just follow suit with with the same operations on the derivative, that is, the index operations differentiate to themselves in the FM, even in the case of repeated indices. The new approach that was found later is to set up the local Jacobians of the index operations, which are sparse logical matrices, and multiply the adjoint with them, and so use arithmetic propagation as described in Section 5.1.1. Note that this approach entails a reshape operation per se. Thus, in this case, we can circumvent the peculiarity of having to invert some particular implicit behaviour in the adjoint code by resorting to arithmetic propagration. The previous approach would have amounted to a structural adjoint propagation as described in Section 5.1.2, which is what we usually do in the case of structural manipulations as discussed in Section 3.1 and in particular storage expression as discussed in Section 6.5. Given that arithmetic propagation is also by far the most efficient form of propagation, as shown in Section 2.5, and given that the **subsref** and **subsasgn** methods of the array-based derivative class are by far the most complex and the most involved ones, as shown in Section 2.5, the new approach can be employed with advantage in performance everywhere, that is, in FM as well and not just in the RM because it is unavoidable there. This is currently implemented already in our R/ADR tool for the AD of the R language [Wil20b].

The partial derivatives of both indexed assignments and indexed expressions are computed by setting up *adjuvant objects*, that is clones of the variables in question and performing a mock execution of the operation in question. More precisely, for an indexed expressions `x(i)`, we create an adjuvant array `X` of the same size as `x` and fill it with the integers $1, 2, \ldots, N$, where N is the number of elements in `x`. When we perform the mock evaluation of `X(i)` we see in the result exactly which elements are selected. Similarly, for an indexed assignment `x(i) = y`, we create two adjuvant arrays `X` and `Y` and fill `X` with zeros and `Y` with the integers $1, 2, \ldots, N$, where N is the number of elements in `y`. When we perform the mock evaluation of `X(i) = Y` we see in the result exactly which elements of `X` are overwritten by which elements of `Y`. In both cases this information can be used to set up the sparse partial matrices efficiently using **sparse**. In the case of repeated indices in `i` there will be columns with several non-zero entries or all-zero columns in the resulting partial derivatives. The technique of adjuvant objects is also used in the differentiation of the **kron** builtin in a slightly expanded form (cf. Section 5.3.4).

3.3.1 Multiple pairs of parentheses in expressions

In GNU Octave there is the syntactical extension that both function calls and index selections may be repeated, that is, multiple pairs of parentheses may be appended where there are none yet, as in `x(1,2:3)(:)` or `find(x)(1)`, while in MATLAB proper there can by only one. This is not supported by ADiMat.

3.4 Optimization

The many runtime functions to undo the implicit behaviour of many MATLAB operations are relatively costly, in particular due to the interpreted nature of the MATLAB language. Hence

each of these runtime functions can be omitted from the generated adjoint code. There is a group option parameter `well-behaved`, which suppresses most of the additional code. In this mode the adjoint code will look very similar as in the relevant literature [Gil08; GW08a]. This option is useful for educational purposes but also for the performance. However this comes at the price of the adjoint code being very brittle and allergic against the implicit behaviours mentioned. This mode is used however for the runtime functions that are pre-generated by ADiMat, as described in Section 5.1.3, i.e. the replacement functions are carefully written to be well-behaved.

These options to turn the aditional code on and off are implemented in a separate part of the processing pipeline. This is easily possible due to the adjoint code being emitted in terms of abstract code elements by the core adjoint code generator. Thus, the postprocessing step where the abstract elements are devolved to regular AST elements is the natural point where these options are implemented, as described in Section 6.5.3.

3.5 Complex expansion

There are many situations where MATLAB will implicitely expand the value of variables from real to complex numbers. Contrary to the other situations of data type changes, such as BSX and array selections, it is not correct to cast the adjoints from complex to reals in the corresponding step of the program reversal. This means, the basic principle that the adjoint should mirror the data type of the program variable is not true in this case. This is also contrary to the forward mode in this case.

As a simple example consider a SLC program $c \to \cdots \to s \to x \to \cdots \to f$ computing an analytical function f in a single scalar value c with one specific operation at one point x where the value becomes complex. Necessarily the partial derivative of that step must be complex as well. Hence, in FM the derivative will be complex from that point onward as well. In the RM however, the adjoint may well be real all the way backwards from the end result f up to that point, and only then become complex. It must necessarily be complex in all the steps before that point. Hence, at none of the preceding steps we may legally cast the adjoint to a real value, for example for the false reason that the corresponding program variables is real.

This case is mathematically quite interesting, and hence it will be discussed in more detail in the later Chapter 7.

4 Efficient I/O for the reverse mode

The reverse reading of the stack in the reverse pass of the adjoint code is an interesting challenge for the I/O subsystem of the OS.

The stack of the calculation for medium scale problems can easily reach sizes of several GB of data. When this stack size becomes too large for the available RAM one option is to offload parts of the data to secondary storage. Since the stack data is accessed solely in a LIFO fashion there is ample opportunity to use asynchronous writes and prefetching to reduce I/O wait times.

On the application layer in ADiMat there are several variants of a single runtime function **adimat_store** which each implement one form of stack. The default variant uses a persistent cell array to keep the stack in memory.

Two other variants use asynchronous I/O available via the AIO API [Bha+03] and the MPI-IO library [TGL02] which is part of MPI 2 and implemented in MPICH [GTL99], respectively. The corresponding software layer which abstacts from these I/O layers and manages the prefetching is called RIOS [WBMB15] and published as an open source software library [Wil13c].

With ADiMat we then implement a stack as a serialization layer for the basic MATLAB data structures, and write the bytestream to disk with the RIOS library. For the stack mechanism to work also in the forward-over-reverse Hessian mode the Taylor series objects (cf. Section 2.11) have to be serialized to binary data as well.

In an experiment with a typical medium sized PDE solving the Burger's equation the actual I/O demand from the adjoint evaluation turned out to be not that large after all. The data

rate was easily sustained by reasonably well-equipped I/O system with hard-disks. One obvious thing to avoid are the synchronous writes of data, which are to be many asynchronous. Another issue is appearently that reading a file blockwise backwards occurs a large penalty upon the read operations. Here we devised a simple prefetching strategy that schedules the corresponding reads with a given number of blocks in advance.

Altogether the results were very promising and even impressive. The runtime factor for the computation of the gradient was recorded as well below 30, which is in our view a very good mark considering the interpreted nature of the MATLAB language. The same problem written in Fortran90 and differentiated with Tapenade [PH05] was able to achieve the typically really impressive factors of about seven, when using the RIOS libary for the stack, which is really not far from the theoretical optimum of about three [GW08a].

As for the performance of ADiMat we encounter here a similar effect that is frequently observed with interpreted languages: Depending on the vector length of typical variables the runtime of the code changes. For small vector length the code evaluation time is bound by the number of language statements which each will typically require several thousand CPU cycles [Büs+14]. For larger vector length the time is bound by the floating point performance of the machine. Hence it is essential that problems are formulated in a vectorized fashion [HH16; Mat18].

In the interpreter bound regime the adjoint code of ADiMat performs not that well. The runtime overhead can exceed the number of several hundred. This is reflective of the number of lines of code in the runtime functions invoked by the code. Hence, in theses cases it may be particularly effective to use the so called well-behaved mode of adjoint code, which omits a considerable number of runtime operators, as described in Section 3.4.

However, the important thing to take away from our experiments with RIOS is that for a vectorized numerical code the adjoint computation becomes feasible within the theoretical limits when the vector length or problem size reaches a certain point, with an acceptable overhead due to the code being MATLAB after all. The generated adjoint code is obviously just as vectorized as the input code.

The message is that the performance of AD in MATLAB should always be evaluated at several points with different problem sizes, making sure to obtain some data points in the interpreter bound and some in the compute bound regime.

The overall important message regarding the runtime factor of the reverse mode is that a time overhead in the asymptotic class of $O(1)$, even though the hidden constant of the adjoint code generated by ADiMat for MATLAB is horribly large at first sight, is qualitatively entirely different from a time overhead of $O(N)$, which is unavoidable with any forward method.

The need for an asynchronous stack may become more relative in the future, given that the speed of memory is increasing relative to the speed of CPUs and FPUs. In fact Moore's law appears to have shifted its center of application, so to say, towards the width of the memory busses and network interconnects nowadays, which means that the so called the memory gap is closing very quickly currently [Kru16].

Thus the future development of compute technology looks likely to increase the availability of fast and vast memory, and this means that, in a sense, time is working for the reverse mode. However, the complexity of the compute hardware will probably still require a dedicated software layer to handle the data movements.

Generally, offloading derivative computations to separate threads, is an idea that has been proposed for AD before and remains to be explored to its full potential [Bis91; BGJ91; Ben96; Wal99; BRV08]. Given that more and more interpreted languages support asynchronous operations, and at the same time the so called global interpreter lock is being recognized as a bottleneck, it is probably just a matter of time until a language suitable for high-performance numerical computation is equipped with efficient multithreading to achieve concurrent asynchronous operations on the language level.

4.1 Introduction

The last decade has witnessed a continuous growth in the development and use of mathematical software for the automated evaluation of derivatives. The term *automatic* or *algorithmic differentiation* (AD) [GW08a; Ked80; Ral81] refers to a set of techniques for transforming a given computer program into another computer program capable of computing its derivatives exactly, i.e., up to roundoff error. Software packages implementing these AD techniques are available for various programming languages; see `http://www.autodiff.org` for a list of current AD tools. These tools are used in a rich set of different scientific and engineering disciplines [Ber+96; Bis+08; Büc+05; Cor+02; For+12; GC91].

The two most prominent AD techniques are called forward and reverse mode. By carrying forward derivatives of intermediate variables with respect to input variables, the forward mode (FM) follows the control flow of the original program. In the reverse mode (RM), in contrast, the control flow of the original program is reversed. That is, the RM propagates derivatives of output variables with respect to intermediate variables. FM and RM represent two ends of a broad spectrum of algorithmic techniques [BH96; GW08a].

The focus of this article is on storage issues of the RM, a crucial aspect in most—if not all—actual applications of AD tools to any real-world computer program of reasonable complexity. A program generated by RM requires the storage and retrieval of potentially very large data sets. The sheer size of the data generated when executing such a program may exceed the capacity of the main memory within a runtime of a few seconds. Moreover, batch systems or computing environments such as Matlab may also impose virtual memory limits that are smaller than the actual main memory available. Hence, it is often unavoidable to store the data *out of core*, i.e., on some larger background storage via the file system, usually a hard disk or a large-scale storage system on a compute cluster. However, while such background storage has a much larger capacity than main memory, its access time is much slower than that to main memory. This is the reason why the efficiency of storage and retrieval operations is typically the key factor of the performance of RM-generated programs.

The new contribution of this article is the design, implementation, and evaluation of a software architecture called **R**everse mode **I**/**O** **S**tream (RIOS) that addresses these potential I/O performance problems in the RM. The design goals of RIOS include

1. to provide large capacity background storage like hard disks while at the same time using the full amount of the available main memory for caching,
2. to overlap the data I/O with computations and perform the I/O asynchronously in the background while continuing with computations in the meantime,
3. to rearrange data accesses such that I/O is performed in blocks, and
4. to make available a mechanism allowing for efficient I/O in reverse direction.

The last item in this list is the major design goal. In the RM, we have the peculiarity that data is extensively retrieved in reverse order of being stored. It has already been remarked in [Chr92] that appropriate prefetching techniques could lead to a reduction of the overall RM overhead. There are also techniques for reducing the amount of data that is written by the RM differentiation process. While the basic approach of applying the RM to the whole program by storing all necessary intermediate values is usually called *store-all*, the alternative is to differentiate only a part of the program at a time, or, more generally, to *recompute* intermediate values. This is usually done on the level of function calls [Nau08] or in for loops by a technique called binomial *checkpointing* [GW00] and allows for a trade-off between storage and computation. Thus the amount of data to be stored at any one time can be limited so that it fits into the main memory. However, it can be challenging to achieve the right balance between storage and recomputation that fully exploits the available main memory, especially when the amount of work in the parts of the program is irregular. In these cases RIOS may help to smooth over the instances where data does have to be written to disk with its use of asynchronous I/O. Furthermore, the checkpoints themselves

also constitute data that is usually written to disk, and they are also accessed in reverse order of being written, so they may themselves advantageously be written and read using RIOS. In these respects, RIOS is orthogonal to the existing techniques, and attempts to make the RM of AD more efficient by concentrating on the actual I/O, not withstanding that the user employs the many existing techniques for efficient RM AD. To the best of our knowledge, there is currently no AD tool with a particular emphasis on I/O, although for example the AD tool ADOL–C [GJU96] performs I/O in blocks to write its data to disk, which already goes a long way to mitigate the adverse effects of reading in reverse direction. Therefore, we direct our efforts into facilitating the reading of a sequence of data items from background storage in reverse direction. This requires to circumvent standard buffering techniques present in the C and C++ library and operating system layers because, by default, they assume that sequential reading is carried out in the forward direction. To prefetch data that is speculated to be needed in the near future, operating system kernels and hardware often use read-ahead techniques. For the RM, however, reading in forward direction may be harmful to the performance. It is therefore potentially advantageous to exploit specific operating system features to optimise reading progress in reverse order. One such feature to implement prefetching in reverse direction is the system function posix_fadvise. Since there are many tweaking parameters and many popular file systems, it is reasonable to design RIOS based on efficient large-scale I/O software libraries that already exist.

We accomplish our design goals by means of an abstract layer. More precisely, RIOS introduces a custom stream buffer that can be used to form a standard C++ I/O stream. RIOS enables to write to and read from this stream, possibly setting some options such as the amount of main memory to be used. Experimental results reported at the end of this paper give evidence that RIOS is useful for different AD tools. We also show the generality of RIOS by choosing AD tools that support different flavours of programming languages. In particular, we demonstrate its feasibility for the two AD tools Tapenade [HP04; HP13] and ADiMat [Bis+02]. Tapenade implements the FM and RM via a source transformation approach. It is designed to transform programs written in Fortran77 and Fortran 90 [PH05] as well as in C [PH08]. In contrast to these strongly typed programming languages, ADiMat is an AD tool for Matlab, a dynamically typed scripting language that is usually interpreted by an interactive environment. ADiMat is based on a combination of source transformation and operator overloading and generates FM and RM code [Veh09; WBB12] for programs written in Matlab and its dialect GNU Octave.

RIOS is not only relevant for these two specific AD tools. Its functionality may be beneficial for any AD tool implementing the RM. RIOS might also be relevant for AD tools that implement the FM as long as the implementation is based on program traces, also called tapes. AD tools that fall into this class include ADOL–C [GJU96] and CppAD [BB08]. ADOL–C can write the tape to disk using plain C file I/O in blocks and, thus, could benefit from RIOS's capability of handling I/O asynchronously. CppAD currently keeps the complete tape in main memory and/or virtual memory. To overcome these space limitations, RIOS could here provide access to a larger background storage.

This article is organised as follows. After a short description of related work in Section 4.2, we present an example of a transformation carried out by the RM in Section 4.3. This example illustrates the need for accessing data in reverse order and shows how stack operations come into play. In Section 4.4, we discuss why standard I/O facilities are not adequate for implementing a large-scale stack on background storage to be used for the RM. In Section 4.5, we describe RIOS, a custom stream buffer with a peculiar buffering strategy tailored for the RM. In Section 4.6, we present some performance results for two test cases, an artificially constructed example with a rather typical access pattern for the RM and another example from the solution of a partial differential equation. Concluding remarks are given in Section 4.7.

4.2 Related work on I/O in high-performance computing

The bottleneck of data I/O to and from large capacity storage systems has been a subject of intense research for quite some time. The survey [GVW96] identifies four basic techniques for the solution of I/O performance problems: exploiting caching and data locality, overlapping I/O

with computations, reducing or rearranging data accesses, and exploiting device parallelism. As already indicated in the previous section, RIOS broadly covers the first three categories while it does not directly target the fourth category. However, exploitation of device parallelism could be included to RIOS as well, for example by using a striped RAID file system or by running the AD-generated code in parallel on several machines.

Our work is related to various aspects of I/O in high-performance computing (HPC). In particular, there is a connection to prefetching techniques, for example the work on pre-execution prefetching [Che+08] or on using Markov models for data access prediction [Che09]. The difference to our work is that we have only sequential and, in particular, reverse sequential accesses. Reverse sequential accesses are not uncommon in HPC applications [Pur+95] and there is explicit support for reverse sequential reading in GPFS [SH02] and also in the MPI-IO part of MPI-2 [For09] via the hints mechanism. Specifically, in MPI-IO there is the file hint *access_style*, which can be given the value of *reverse_sequential* [For09]. However, it is difficult to find information on what is actually done to implement these features or what performance they achieve. For instance, using the hints mechanism in MPI-IO, a file can be given only one type of access pattern which seems to force us to close and reopen the file whenever switching between write and (reverse) read phases. One empirical work where reverse sequential reads are explicitly considered is DiskSeen [Din+07]. We believe that the particular use case of I/O for the stacks in the RM might in fact be an interesting test case for these prediction-based approaches or other self-optimizing I/O suites [Wie+13]. The reason is as follows: While, conceptually, the data is read in reverse direction, in practice this consists of several read operations on smaller pieces of data, which are performed by standard read primitives, and thus in forward direction. So, when the size of these reads varies, prediction-based techniques will face certain challenges to determine whether reading goes in forward or reverse direction, probably resulting in the need to set some parameters such as the history window size a priori.

The ADIOS interface [Lof+08] is also related to our work. It allows for abstractions from the particular I/O methods selected via configuration files. This system makes possible to switch from, say, POSIX I/O to HDF5 or MPI-IO without recompiling. It is similar to our approach in that RIOS also provides different stream implementations depending on certain user options. An extension of ADIOS, called CIAO, introduced in the context of energy efficiency [Kun+12] allows to label specific regions of an application program with a particular name. These named regions can be expected to use similar I/O patterns and their behaviour is analysed by the CIAO library. This feature could be relevant for the RM since a code generated by the RM consists of two clearly defined parts. The first part consists of storing data to background storage in forward direction while the second part requires to access data from background storage in reverse direction. This is detailed in the following section.

4.3 The need for accessing data in reverse order

The RM requires the storage and restoration of the intermediate values of program variables and the reversal of the control flow of the program. This is done by generating appropriately augmented source code. This augmented code consists of two parts, the so-called forward sweep and the reverse sweep. Notice that the forward sweep of the RM is different from the FM. The forward sweep of the RM is basically the original source code, but it is augmented by store operations that save any information required during the reverse sweep. Required information includes the values of variables that are needed to evaluate the partial derivatives of elementary operations, but may also include other variable values such as array indices. Determining the minimal set of variable values to be recorded is described in [HNP05].

The reverse sweep contains the so-called adjoint statements that are responsible for accumulating the derivatives in the adjoint variables. For example, consider a statement sequence $y \leftarrow f(x); z \leftarrow g(x, y)$, where x is the input variable, z the result, and f and g are elementary operations of the programming language. Each statement in the original code induces a number of adjoint statements. These are $\overline{x} \leftarrow \overline{x} + \frac{\partial g}{\partial x}\overline{z}; \overline{y} \leftarrow \overline{y} + \frac{\partial g}{\partial y}\overline{z}$ for the second statement and

$\overline{x} \leftarrow \overline{x} + \frac{\partial f}{\partial y}\overline{y}$ for the first, where $\overline{x}$ denotes the adjoint variable associated with x. All adjoint variables are initialised to zero, except the adjoint variable associated with the result which is initialised by $\overline{z} = \frac{\mathrm{d}z}{\mathrm{d}z} = 1$. The adjoint statements must be executed in the reverse order of the original statements. The result of the derivative propagation is then available in the adjoint variable associated with the input variable, in the example $\overline{x} = \frac{\mathrm{d}z}{\mathrm{d}x}$. Hence in the reverse sweep, all control flow structures are reversed as well. In particular, loop iterations have to be run in reverse order, unless the order of loop iterations is arbitrary, i.e., the loop is parallelisable. Also, any branch taken in an if statement has to be remembered. The adjoint statements will also generally require the values of program variables on the right-hand side of the original program statement at the time when this original statement was executed. These values are needed for evaluating the partial derivatives of the elementary operations in the adjoint statement. One solution to this problem is to store the value of all variables before they are overwritten. Since the data items stored during the forward sweep will be required in reverse order during the reverse sweep, the natural data structure to store this information is a stack. Push operations are inserted into the forward sweep code for storing information and pop operations in the reverse sweep retrieve it.

The following example illustrates the program transformation carried out in the RM. Consider the Matlab function shown in Listing 1. This function computes the scalar value

$$z = \sum_{i=0}^{k-1} p_i \, x^i \tag{1}$$

of a polynomial defined by the vector of its coefficients $\mathbf{p} = (p_{k-1}, p_{k-2}, \ldots, p_0)$ at a given scalar point x.

```
1  function z = polynom(x, p)
2    k = length(p);
3    z = p(1) .* x;
4    for i=2:k-1
5      z = (z + p(i)) .* x;
6    end
7    z = z + p(k);
```

Listing 1: A Matlab function evaluating a polynomial using the Horner scheme.

The result of transforming this code by ADiMat in the RM is shown in Listing 2. For the sake of clarity, this adjoint code is a simplified version obtained by setting certain ADiMat options to non-default values. In particular, the wrapper function calls which are necessary for undoing implicit shape changes [WBB12] are omitted here.

Compared to the function signature of the original code given in Listing 1, the function signature of the RM-generated code depicted in Listing 2 is augmented with the adjoint input parameter a_z and the adjoint return parameters a_x and a_p. The function body consists of the forward sweep in lines 2–14, followed by the initialisation of the adjoint variables with **a_zeros** in line 15, and the reverse sweep in lines 16–34. The forward sweep code is basically the original code unchanged, except that **push** operations are inserted and nested expressions are broken up. This is done by a preprocessing step called outlining, which introduces temporary variables with prefix ca or fra. The function **a_zeros** creates zero adjoint objects. In this example, we are interested in differentiating a scalar-valued function, see (1). Here, an adjoint object created by **a_zeros**(p) is a zero array of the same size and shape as p. The for loop in the reverse sweep has an iteration range given by the builtin function **fliplr** applied to the original iteration range. The builtin **fliplr** reverses the order of the items of a row vector and has exactly the desired effect of reversing the order of iterations.

```
1  function [a_x a_p nr_z] = a_polynom(x, p, a_z)
2    ca1 = 0;
3    k = length(p);
4    z = p(1) .* x;
```

```
5     fra1_2 = k - 1;
6     for i=2 : fra1_2
7         push(ca1);
8         ca1 = z + p(i);
9         push(z);
10        z = ca1 .* x;
11    end
12    push(fra1_2, z);
13    z = z + p(k);
14    nr_z = z;
15    [a_ca1 a_x a_p] = a_zeros(ca1, x, p);
16    z = pop;
17    a_p(k) = a_p(k) + a_z;
18    sa1 = a_z;
19    a_z = a_zeros(z);
20    a_z = a_z + sa1;
21    fra1_2 = pop;
22    for i=fliplr(2 : fra1_2)
23        z = pop;
24        a_ca1 = a_ca1 + a_z.*x;
25        a_x = a_x + ca1.*a_z;
26        a_z = a_zeros(z);
27        ca1 = pop;
28        a_z = a_z + a_ca1;
29        a_p(i) = a_p(i) + a_ca1;
30        a_ca1 = a_zeros(ca1);
31    end
32    a_p(1) = a_p(1) + a_z.*x;
33    a_x = a_x + p(1).*a_z;
34 end
```

Listing 2: The Matlab function obtained from transforming Listing 1 by ADiMat in RM.

To evaluate the polynomial (1) with $k = 5$ and coefficient vector $\mathbf{p} = (1, 1, 1, 1, 1)$ at $x = 2$ we execute

```
z = polynom(2, [1 1 1 1 1]);
```

in the Matlab interpreter. The derivatives of (1) with respect to x and $\mathbf{p}$ are given by

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \sum_{i=0}^{k-1} i p_i \, x^{i-1} \qquad \text{and} \qquad \frac{\mathrm{d}z}{\mathrm{d}\mathbf{p}} = (x^{k-1}, x^{k-1}, \ldots, x^0). \tag{2}$$

Invoking the adjoint function from Listing 2 in the Matlab interpreter at the same point x via

```
[a_x a_p z] = a_polynom(2, [1 1 1 1 1], 1);
```

computes these derivatives in the variables a_x and a_p, respectively. Here, the adjoint input parameter, a_z, associated to the function result z is set to a scalar one. In addition to the derivatives, the adjoint function also returns the polynomial value z. The analytic results (2) are easily verified as follows. By observing that $\mathrm{d}z/\mathrm{d}x = \sum_{i=0}^{k-2}(i+1)p_{i+1}\, x^i$ and since the coefficients satisfy $p_i = 1$ for all i, we obtain this derivative by evaluating, at the same point $x = 2$, a polynomial whose degree is reduced by one and whose coefficients are linearly decreasing. That is, the value computed by

```
dzdx = polynom(2, [4 3 2 1]);
```

coincides with the value of a_x computed above by the adjoint code. From (2), the entries of the vector $dz/d\mathbf{p}$ are given by the powers of two which are computed in a_p.

Note that the RM code shown is constructed in the store-all fashion of the reverse mode, that is, all necessary intermediate results are stored on the stack.

4.4 An interface between RIOS and automatic differentiation tools

The code generated by the RM of an AD tool will typically contain many push and pop operations. Depending on the particular strategy of an AD tool, this code can also involve some form of recomputation for certain parts of the code. A clever combination of recomputation and push/pop operations can lead to efficient RM-generated code in practice. However, rather than considering such a combined strategy, the focus of this article is on push/pop operations that typically dominate the overall performance. In this section, we describe the RIOS interfaces of these push/pop operations. We show how these operations can be mapped to write and read operations on files, by virtue of the serialisation of the variable values to be stored. We take codes generated by Tapenade and ADiMat as illustrating examples for multifaceted AD tools. For small data, current AD tools offer various in-memory solutions. However, to achieve scalability for large quantities of data, a new approach is necessary that uses files on some background storage efficiently.

4.4.1 Stack interfaces in Tapenade

The RM code generated by Tapenade contains calls to various functions for storing and retrieving data from a stack. Examples include pushreal8, popreal8, pushreal8array, and popreal8array. The file `adStack.c` of the Tapenade distribution implements these functions in C. All of these functions receive as parameters a pointer to an array of the respective data type and an integer indicating the size of the array being stored or retrieved. They are ultimately mapped to the two functions shown in Listing 3, which requires just a multiplication of the array size by the byte size of the data type, e.g. pushreal8(v, n) calls pushNarray(v, n*8).

```
1 void pushNarray(char *x, unsigned int nbChars);
2 void popNarray(char *x, unsigned int nbChars);
```

Listing 3: The basic push and pop functions used by Tapenade.

The function pushNarray writes nbChars bytes of data, while popNarray reads back the nbChars bytes that were last written, i.e., the functions work according to the usual stack semantics. We shall see in the next section that these two functions ultimately define the RIOS interface to the AD codes, in both Fortran and Matlab. In Tapenade's default implementation found in `adStack.c`, these functions use a doubly linked list of memory blocks of a fixed size, by default 16 kB, to store the data.

4.4.2 Stack interfaces in ADiMat

The runtime environment of ADiMat contains two basic functions for storing and retrieving values on a stack. The function **push** stores its arguments in left-to-right order on the stack. That is, **push**(a, b); is equivalent to **push**(a); **push**(b). Any number of arguments can by given to **push** because it uses Matlab's special formal parameter varargin. The function **pop** retrieves as many items from the stack as there are output arguments in the call. In Matlab, a function can return any number of results. The special formal output parameter varargout is used to specify that this number can be variable. For example, an invocation like a = **pop**() invokes **pop** with one output argument, as does [a] = **pop**(). The statement [b a] = **pop**() invokes **pop** with two output arguments, and so on. Inside the function, the builtin **nargout** is used to determine the number of outputs requested. The items retrieved from the stack are placed in the output arguments in left-to-right order. That is, [b a] = **pop**(); is equivalent to b = **pop**(); a = **pop**().

The functions **push** and **pop** themselves call a function **store** with the two arguments mode and obj. If the integer mode is 0 then the function **store** retrieves one item from the stack. When

mode is 1, then the value obj which can be of any kind is pushed on the stack. There are several different versions of the **store** function, as shown in Figure 3. We call one such implementation of the **store** function a *stack*. The following stacks are available in ADiMat; all but the first are implemented as extension functions written in C/C++, which are called *MEX-functions*.

- NATIVE-CELL: This stack is implemented as an m-file that uses a cell array in a persistent variable to store the items on the stack. Thus, data remains in main memory and inside the workspace of the function store.
- MEM: This stack is implemented as a MEX-function. The MEX-API functions are used to obtain the pushed object's data, which is stored in main memory in a linked list. However, this requires that we take a copy of the object's data.
- SSTREAM: This stack is also implemented as a MEX-function which serialises the data as described below. The resulting byte stream is written to an std::stringstream, and thus still remains in main memory.
- FSTREAM: This stack is identical to SSTREAM except that an std::fstream object is used to store the data on disk.
- ABUFFERED-FILE: These stacks are identical to SSTREAM and FSTREAM, except that an std::iostream object with a custom streambuffer, as described in Section 4.5, is used to store the data on disk. We design several different streambuffers, for example using the standard I/O functions of C (StreambufCFile), those of the Windows API (StreambufWinFile) and finally the one that we mainly present here, StreambufBlocker, which in turn uses various library routines for asynchronous I/O.

Each stack resides in a different directory and the function **adimat_stack** can be used to manipulate the search path in Matlab so that one or the other version is found and used. The last two stacks are capable of writing the data to hard disk. Though all these stacks are used in the numerical experiments, the focus of this article is on the stack ABUFFERED-FILE.

In Matlab variables may not only hold numerical arrays but also struct arrays, cell arrays or other objects. We use a serialisation software layer, shown as class Serialiser in Figure 3, that can convert objects of any of the builtin classes of Matlab, including cell arrays and struct arrays, to a byte stream. Objects of user-defined classes can be serialised if they support conversion from and to structs. The byte data that comes out of the serialisation layer is written and read using the functions pushNarray and popNarray shown in Listing 3. Thus, the serialisation layer provides a mapping from the high-level push and pop functions in Matlab to the byte-wise interface in Listing 3.

The serialisation layer will in general perform several low-level operations when a single item is serialised or deserialised. For example, when a double array is pushed, the serialiser will first push the array data on the low-level stack. Then, the vector of array dimensions is pushed, then the number of dimensions, and finally an ID code indicating the type "double array". The deserialiser will first retrieve the ID code from the low-level stack. Then it knows that there are three more items to fetch. After fetching the second item, the number of dimensions, it knows the length of the third item, the vector of array dimensions. Now it can calculate the length of the actual array data. Note that this already indicates the necessity for an intermediate layer handling IO in blocks to buffer the reads of small amounts of data from the stack. This turns out to be important later to understand why the standard I/O facilities are unsuitable for our needs. One possible approach which is orthogonal to the work proposed in this note is to split the serialisation data into several streams, using one stream for each of the four categories of data items mentioned above, thus creating three control data and one value data stream. This is already implemented in the serialisation layer used by ADiMat and can alleviate the problems created from mixing large and small writes.

All of the C++-based stacks are also available for Octave, as so-called *oct-functions*, which correspond to the MEX-functions of Matlab. Note that everything downward from the std::iostream can be used identically in Octave and Matlab. Basically only the serialisation layers are different.

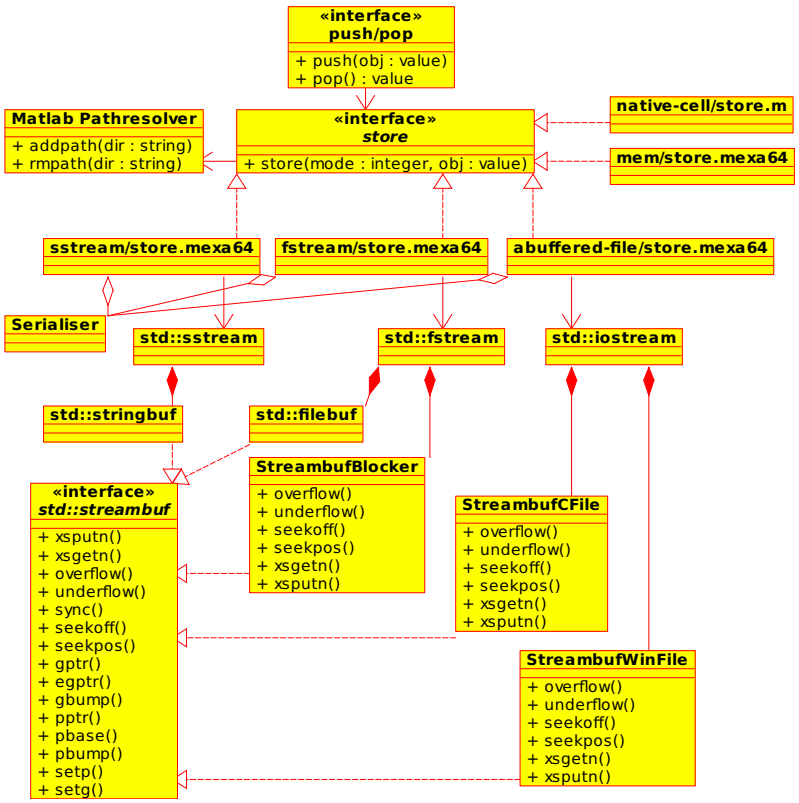

Figure 3: Class diagram of the infrastructure behind ADiMat's push and pop functions.

4.4.3 Common backend for ADiMat and Tapenade stacks

Next, we discuss how to implement the interface of Listing 3 such that the data can be offloaded to background storage. To this end, we consider std::iostream objects from the C++ I/O library. Objects implementing the interface std::iostream have the following methods that we need:

```
1 basic_ostream& write(const char_type* s, std::streamsize count);
2 basic_istream& read(char_type* s, std::streamsize count);
3 basic_istream& seekg(pos_type pos);
```

Listing 4: Methods from the iostream interface.

Calls to pushNarray can be directly mapped to a call to the iostream::write method. This will append the count bytes after the current stream position and also advance the stream position by count bytes. Hence, a following call to pushNarray will append the data after that and so on. A corresponding sequence of popNarray operations will have to retrieve the data in reverse order. The sequence of operations to retrieve a single item of n bytes, which we call a *reverse read* operation, is shown in Listing 5. First the stream position is moved backwards by n bytes, by calling the *seekg* method, then the n bytes of data are read, and finally the stream position is moved back again in order to leave it at the correct location for the following stack operation, be it another pop or a push[1].

```
1 s.seekg(-n, std::ios::cur);
2 s.read(data, n);
```

[1]The std::iostream methods allow for two independent stream position pointers, a get pointer and a put pointer. Here, we assume that they are synchronised (std::fstream) or we synchronise them manually (std::stringstream).

```
3  s.seekg(-n, std::ios::cur);
```

Listing 5: The basic reverse read operation in C++.

The same kind of operation can also be done with the C I/O interface. The functions that are needed here are *fseek* and *fread*, as shown in Listing 6.

```
1  fseek(file, -n, SEEK_CUR);
2  fread(data, n, 1, file);
3  fseek(file, -n, SEEK_CUR);
```

Listing 6: The basic reverse read operation in C.

Using the C++ iostream interface has two advantages: Firstly, we can readily use the two different I/O stream implementations provided by the C++ I/O library. For the stack FSTREAM, we use std::fstream representing a file on disk, while for the stack SSTREAM we use std::stringstream representing a memory buffer. Secondly, it is possible to create I/O streams with customised behaviour, which is what we do for the stack ABUFFERED-SSTREAM and what is explained in more detail in the following sections.

4.5 RIOS: A custom stream buffer for reverse reading

In this section we briefly summarise the facilities for file I/O in both C and C++. Then we analyse the buffering strategies employed by these two different I/O layers, for the case of the GCC compiler and its associated libraries. This analysis will show that the existing file I/O facilities in both C and C++ are unsuitable for backwards reading of large-scale data. In particular, their performance turns out to be low when mixing reverse reads to large and small data. Hence, we propose and implement the novel RIOS software for a special I/O layer adapted to reverse reading to be used as the backend of an RM stack[2].

4.5.1 File I/O facilities in C and C++

Let us first recapitulate the facilities for file I/O available in C and C++. In C, there is the opaque FILE data type with which library functions like *fopen*, *fclose*, *fread*, *fwrite*, *fseek*, *ftell*, and *fflush* are associated.

The C++ I/O library provides the std::iostream object and two derived classes: Objects of type std::stringstream and objects of type std::fstream. Objects of type std::iostream can also be constructed themselves. The corresponding constructor requires as argument a pointer to a so-called stream buffer, as shown in Listing 7. A stream buffer is an object of the class std::streambuf or a derived class.

```
1  std::streambuf *s = ...;
2  std::iostream myStream(s);
```

Listing 7: Constructing std::iostream objects with a particular stream buffer.

The idea behind separating the stream buffers from the I/O stream classes is that all the formatting functionality is kept in the I/O stream class and its overloaded operators, most notably the operator <<. On the other hand, the stream buffer is concerned only about reading or writing character sequences. The underlying stream buffer of std::stringstream is std::stringbuf which keeps data in main memory. The underlying stream buffer of std::fstream is std::filebuf which represents a file on disk. Implementing other stream buffers with a particular behaviour allows for the creation of I/O streams for special purposes as per Listing 7, and we capitalise on this option. In the following section we detail the different I/O stream variants that are part of RIOS. When using the ADiMat stack ABUFFERED-FILE, the desired RIOS stream can be specified using the function **adimat_stack_async_io_type**.

[2]Download RIOS from http://rios.ourproject.org/

4.5.2 Buffering strategies of file I/O in C

File I/O in C employs a memory buffer with a default size in bytes given by the constant BUFSIZ, with a current value of 8192. The function *setvbuf* can be used to set this value to a different size b. The following description of the buffering strategy assumes that the size of the buffer is a power of two.

We can understand the buffering strategy by looking at the system calls (`read` and `lseek`) issued by a program doing a series of reverse reads of a fixed size n using the C library functions, as shown in Listing 6. In our case, we look at the current GNU libc version 2.13. Apparently, the buffer is aligned with file offsets which are multiples of b. Thus we can imagine the file as a sequence of *aligned blocks* of size b. Assume that the file pointer is at some offset p, then a reverse read of n bytes will move it to offset $p' = p - n$. When both p and p' are within the same aligned block, the reverse read of $n < b$ bytes can be satisfied entirely from the buffer, and no system `read` occurs. This situation is depicted in Figures 4 (a) to (c).

Now, consider a reverse read that crosses an aligned block boundary where $p' < i \cdot b < p$ for some i, as depicted in Figures 4 (d) to (f). Upon the first backwards *fseek*, the C library will `lseek` backwards to the next lowest multiple of b, i.e., to offset $(i-1) \cdot b$, and then perform a `read` of b bytes, in order to fill the buffer with the data of the aligned block $i-1$, in which p' lies. Then, the *fread* operation can only be partially satisfied from the buffer, because the part between $i \cdot b$ and p is not in the buffer any more. Hence, a second `read` occurs, to load block i again. The final backwards *fseek* makes the C library `lseek` backwards to offset $(i-1) \cdot b$ again and, once more, `read` block $i-1$ to again fill the buffer with that block.

Now, consider a series of reverse reads of size n that read back an entire file. We discuss the following two extreme cases. If the size of the data being read is much larger than the buffer size, $n \gg b$, then the proportion of data that is read repeatedly and redundantly is small. However, the benefit of using such a small buffer is also small. In the second case where the buffer is large, $n \ll b$, each aligned block i will effectively be read three times: Two times when a series of reverse read crosses the upper boundary $(i+1) \cdot b$ of the block, and once more, when the series of reverse reads crosses the lower boundary $i \cdot b$ of the block. From these two extreme cases, we conclude that it does not help to use a large buffer, unless the programmer actively avoids that reverse reads cross buffer boundaries, while using a small buffer obviously forfeits the advantage of buffering.

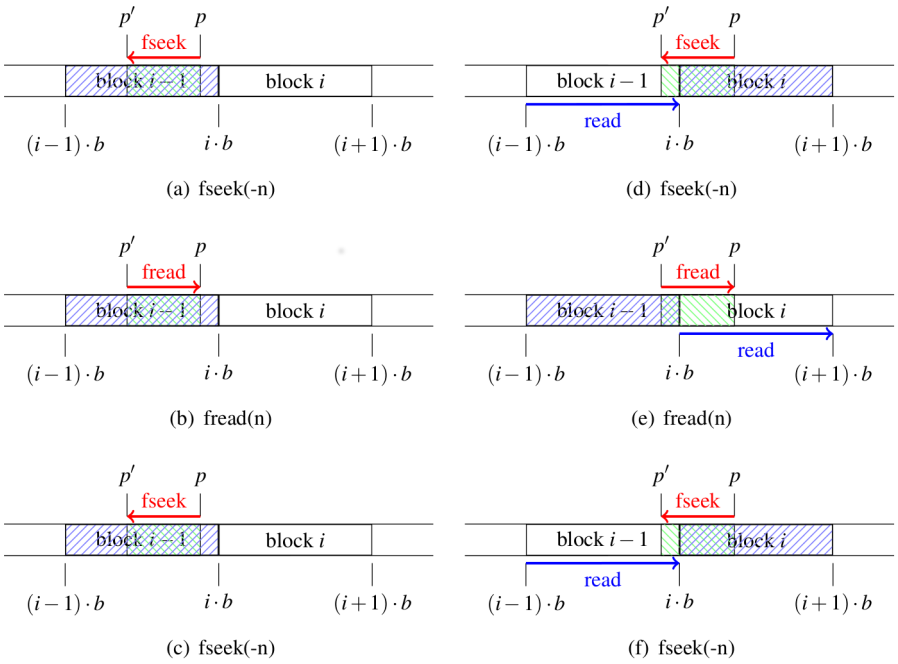

Figure 4: The effect of the buffering strategy of the GNU C library version 2.13 on a reverse read operation, cf. Listing 6. Sub figures (a)–(c): When a reverse read falls between two alignment boundaries, the read can be satisfied from the buffer. Sub figures (d)–(f): When a reverse read crosses an alignment boundary, like $i \cdot b$ here, three system reads occur.

4.5.3 Buffering strategies of file I/O in C++

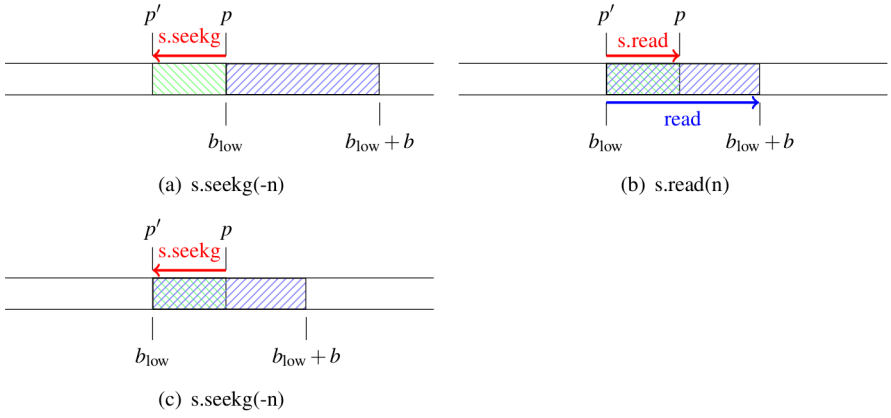

Figure 5: The effect of the buffering strategy of the GNU C++ I/O library on a reverse read operation; cf. Listing 5. The buffer offset b_{low} is always aligned with the read offset p' and hence for every reverse read of n bytes there is a system read of at least b bytes.

File I/O in C++ via the std::fstream and std::filebuf objects is buffered with a memory buffer which also has a size of BUFSIZ bytes by default. It is also possible to set a user defined buffer using the method *pubsetbuf* of std::filebuf. Interestingly, the buffering strategy of std::filebuf in the GCC implementation appears to be fundamentally different from that of C-style file I/O, as we see by looking at the output of the strace utility. When doing a reverse read, the new stream position p' is generally before the lower end b_{low} of the current buffer. This lower end is then realigned

with p' and then the maximum of n and b bytes will be read. This is depicted in Figure 5. Now, as with every reverse read the stream position moves backwards by n, so does the beginning of the buffer b_{low}. So, when using a large buffer with $b \gg n$, each read operation will load b bytes into the buffer, because the I/O stream assumes that data will continue to be read in forward direction. For this reason using a large buffer for reverse reads is completely counterproductive. The buffer will be redundantly filled with—in this case useless—data over and over again. The larger the buffer size, the worse this effect: Each byte of data is read b/n times.

4.5.4 Design and implementation of custom stream buffers

We first give a general overview of our design goals for a custom stream buffer. In Figure 6 we show a UML activity diagram of a write operation. This diagram shows that we can broadly separate the actions into three layers. At the very top of the diagram, the initial action is labelled as "Write n bytes". This is what comes out of the serialisation layer and represents a call to the function pushNarray. First we can decide whether to use a custom buffering strategy or not. This amounts to using the predefined I/O stream std::fstream or maybe a stream buffer which uses the C I/O interface on one hand, or a stream buffer with a custom buffering strategy on the other hand. This is the layer labelled *stream buffer* in Figure 6. The next question is what we do when a buffer is full. Obviously we have to write the b bytes it contains to disk somehow, and return a free buffer to the upper layer, but we could use just one buffer or several. Therefore, there is a second layer that is labelled *block sink/source* in Figure 6. We might also call it "buffer management". Finally there is a third layer that is responsible for performing *asynchronous I/O* operations as they are indicated by the small concurrent section in the right part of Figure 6. Obviously, using multiple buffers does only make sense when asynchronous I/O is used, and vice versa. So, the separation between the second and third layer is not cleanly possible in this diagram. However, the third layer is useful for abstracting from the different sets of asynchronous I/O routines that are available.

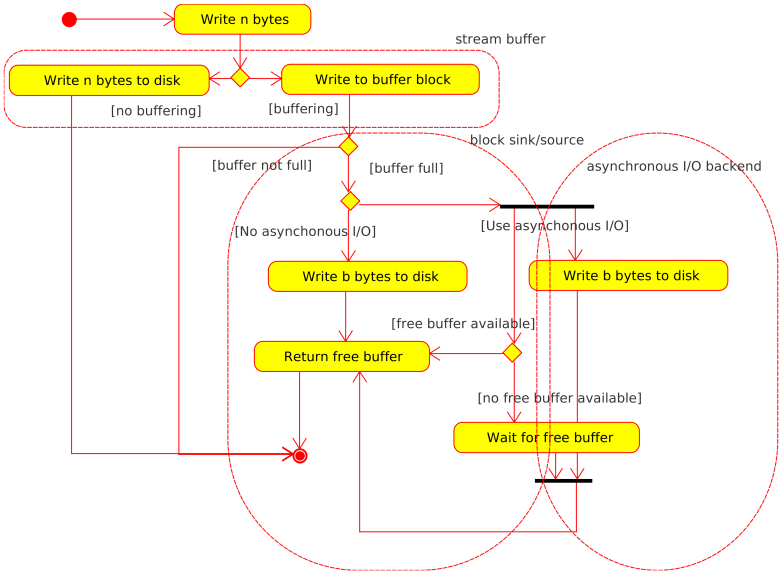

Figure 6: Streambuffer activity diagram.

The separation between I/O streams and stream buffers in C++ exists to provide the concept of a so-called *controlled sequence* that represents a window into the whole data stream, the so-called *associated sequence*. Usually the controlled sequence is a memory buffer. The associated sequence may be yet another std::iostream, or a plain C file, a data base or possibly even a tape reader. The C++ I/O library is a rather complex topic, cf. [LK08], so we briefly introduce only

issues pertinent to our work. In the following, we call the associated sequence the *file* and the controlled sequence the *buffer*.

The std::streambuf class has a set of virtual functions that can be overwritten in order to implement the desired features. The methods that usually are reimplemented in a derived class are shown in Listing 8.

```
1 int_type sync();
2 int_type overflow(int_type);
3 int_type underflow();
4 std::streamsize xsgetn(char *s, std::streamsize n);
5 std::streamsize xsputn(char const * s, std::streamsize n);
6 pos_type seekoff(off_type off, std::ios_base::seekdir way,
7                   std::ios_base::openmode which);
8 pos_type seekpos(pos_type const sp, std::ios_base::openmode);
```

Listing 8: Some important virtual methods of a stream buffer class.

The two methods *xsputn* and *xsgetn* are eventually called by the iostream object in order to read or write some byte data. The method *xsputn* handles the insertion of n bytes at the current position. This function will usually fill the internal buffer with the data, and when the buffer is full, it may call *overflow* in order to flush the buffer to the backend and thus create new space for more data. Likewise, *xsgetn* is the method for reading data. It copies data from the internal buffer to the output pointer and probably calls *underflow* when the buffer is empty to read more data from the backend file. The methods *seekoff* and *seekpos* are used to position the file get and put pointers. These methods are necessary to implement seeking in the iostream via its *seekg* and *seekp* methods, but *seekoff* is also called in order to obtain the current file position.

Note that it is not necessary to overwrite all of these methods. A very simple streambuffer for writing data to a particular backend may, for example, be implemented by just overwriting *overflow*. There is a default *xsputn* which will call *overflow* for every single character. Similarly, when one does not implement *seekpos* and *seekoff*, one may not inquire or change the stream position, and thus one obtains a non-seekable I/O stream, which might be useful for I/O to a network or a data base.

Figure 7 shows a UML class diagram of the stream buffers that we designed. Let us consider two examples occurring in that diagram. As a first example, consider the class StreambufCFile which is simple. It uses a plain file from the C library as a backend. It maps the functions given in Listing 8 to those mentioned in Section 4.5.1 as follows. The function *xsputn* is mapped to *fwrite*, *xsgetn* to *fread*, *sync* to *fflush* and *seekoff* and *seekpos* are implemented via *fseek* and *ftell*. The *overflow* and *underflow* methods are not needed in this case. This streambuffer will not implement buffering by itself. This allows for a direct mapping of operations on an I/O stream to C-style file operations, which is an interesting test case for comparison, and, as we will see, may even perform more efficiently in our usage scenario than a standard fstream from the C++ library.

As the second example, consider the class StreambufWinFile which uses the file I/O functions from the Windows API. It is interesting to note that these are asynchronous by default, but of course we cannot make use of that in the simple wrapper class StreambufWinFile. However, as discussed in the next section, when we use a buffering intermediate layer, then we can exploit these asynchronous functions by using them to implement the third layer, which is shown as the class AsyncIO_Win in Figure 7.

4.5.5 Architecture and buffering strategy of a special-purpose stream buffer for reverse reading

As we mentioned already in the previous section, we separate the design of our custom stream buffer into three layers: stream buffer, block sink/source, and asynchronous I/O. On the first layer, we have the actual stream buffer, which is shown as class StreambufBlocker in Figure 7. It implements what we call a *block forming* mechanism. The idea is that it aggregates the operations on data of variable length n into *data blocks* of a fixed size b. This mechanism reads or writes

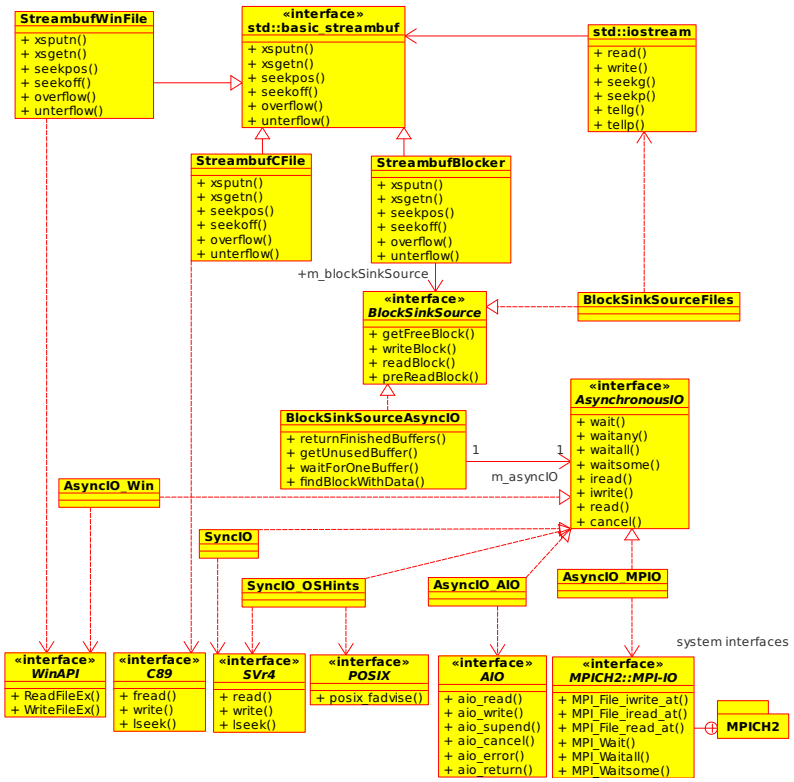

Figure 7: Class diagram of the custom stream buffer.

these blocks from or to the next layer, which is the middle layer, which we therefore call block sink/source. This middle layer is represented by the interface BlockSinkSource in Figure 7. Its main implementation is the class BlockSinkSourceAsyncIO, which is responsible for managing a set of M memory *buffer blocks*. The size of one of these data blocks of the top layer is given by b. Using multiple buffers is what actually allows for the use of asynchronous I/O operations on the bottom layer. At any one time, one buffer block is the active one from which the current read and write requests are satisfied. The others may hold data that is being written or being read asynchronously. The actual I/O operations are on the bottom layer, represented by the interface AsynchronousIO. We implement several variants of this layer for the different I/O functions and libraries that we wish to use. In particular, we will implement this layer by using the POSIX asynchronous I/O (AIO) interface or by using the asynchronous I/O capabilities built into MPI 2, namely those of the MPICH2 implementation. As already mentioned, we can also use the Windows API and, as a test case, even the basic blocking system calls of UNIX.

The buffering strategy we wish to implement is the following: During write phases, full buffer blocks are written away by asynchronous write operations. When all buffer blocks are full, we wait for the first of these operations to complete in order to gain a free buffer block. Then, when a phase of reverse reads begins, the first $M-1$ data blocks that are required will still be loaded in buffer blocks, possibly with pending write operations. To also make use of asynchronous read operations, we define a prefetch amount $P < M$. Upon accessing a data block i, we will issue asynchronous read requests for the data blocks $i-1, i-2, \ldots, i-P$ unless these are already present in a buffer block.

StreambufBlocker, a block forming streambuffer At the top level of a software stack for asynchronous I/O via C++ streams, we implement a block forming stream buffer, in a class called StreambufBlocker. This stream buffer implements the methods listed in Listing 8 and uses a memory buffer of size b for the controlled sequence. At any time the buffer shall be associated

with the i-th aligned data block of the backend file. That is, the b bytes in the buffer correspond to the b bytes of data from $i \cdot b$ to $(i+1) \cdot b - 1$. This is exactly the buffering strategy that the GNU C library also employs, as detailed in Section 4.5.2.

BlockSinkSource, an I/O layer for blocks of data The StreambufBlocker class needs a mechanism to write and read such data blocks when its memory buffer becomes full. For this purpose it uses the BlockSinkSource interface shown in Listing 9:

```
1 int writeBlock(char * buffer, size_t index) = 0;
2 char *readBlock(size_t index) = 0;
3 char *getFreeBlock(size_t index) = 0;
4 size_t bufferSize() const = 0;
5 int sync() = 0;
6 int preReadBlock(size_t index) = 0;
```

Listing 9: The virtual methods of interface BlockSinkSource.

In this interface, the method parameter index corresponds to the block index i. The method bufferSize returns the block size b. The interface BlockSinkSource is used by StreambufBlocker as follows: The method getFreeBlock is called first to obtain a buffer block. The stream buffer method underflow calls readBlock to obtain a buffer block filled with the i-th data block from the file. The StreambufBlocker uses writeBlock to write the data in the current buffer block to disk. Then it has to call getFreeBlock again to obtain a free buffer block again. The crucial point in the design of this interface is that the pointer returned by getFreeBlock need not be the same as before. So, it is possible that the BlockSinkSource internally holds several buffers, and this in turn allows it to use asynchronous I/O. However, just as well, another BlockSinkSource might have just one buffer block and use synchronous I/O. This latter case is implemented by the class BlockSinkSourceFiles (cf. Figure 7), which writes every block to an individual file on disk using the standard std::fstream.

As indicated above, one purpose of StreambufBlocker is to implement a prefetching strategy. The user can set a number P of blocks to be prefetched. The prefetching of the data is actually triggered by StreambufBlocker via calling the method preReadBlock. After any call readBlock(i), StreambufBlocker will also call preReadBlock(i−1), ..., preReadBlock(i−P), as long as $i - P \geq 0$.

We now describe the system for managing multiple buffer blocks. There is an implementation of the interface BlockSinkSource called BlockSinkSourceAsyncIO. This class has $M > 1$ different memory buffers. It maintains a state field, an index field, and a request field for each buffer. It also maintains a field indicating the current buffer, i.e., the index $0 \leq k < M$ of the buffer block that was last returned by getFreeBuffer or readBuffer. The state of a buffer can be either UNUSED, BEING_READ, BEING_WRITTEN, or LOADED. The request field is zero when the state is either UNUSED or LOADED. When the state is BEING_READ or BEING_WRITTEN, the request field holds an integer handle describing the pending I/O request. The index field holds the data block index i with which a buffer block is associated. The state UNUSED represents buffer blocks which are as of yet unused. This is the initial state of all buffer blocks. Buffer blocks are in the state LOADED if they have completed their read or write operation, and hence are in sync with the backend file.

The class BlockSinkSourceAsyncIO uses the Least Recently Used (LRU) strategy to evict data blocks from buffer blocks when a free buffer is needed. The LRU strategy is of course applied only to the set of buffers that are in a committed state, i.e., in state UNUSED or LOADED.

Asynchronous I/O library routines The interface that is used to encapsulate the third layer of our software stack is called AsynchronousIO and shown in Listing 10.

```
1 int iwriteData(char *data, size_t n, size_t offset, int *handle);
2 int ireadData(char *data, size_t n, size_t offset, int *handle);
3 int readData(char *data, size_t n, size_t offset);
```

```
4  int cancel(int *handle);
5  int wait(int *handle);
6  int waitany(int *handles, unsigned n, int *which);
7  int waitsome(int *handles, unsigned n,
8               int *completed, int *numCompleted);
9  int waitall(int *handles, unsigned n);
```

Listing 10: The virtual methods of the AsynchronousIO interface provide an abstraction layer over the POSIX AIO and MPI-IO APIs.

The main idea in this interface is that we subsume the different kinds of pointers and request handles of the various library routines for asynchronous I/O under a common mechanism using integer handles.

For the asynchronous I/O operations that are needed to implement the AsynchronousIO interface shown in Listing 10, we have at least the following three options: The first is the POSIX AIO library, the second is the use of the immediate file operations of the MPI-IO section of the MPI 2 standard, and the third is the use of the I/O routines of the Windows API.

The AIO routines are specified in the POSIX standard. Hence, they are readily available on most Unix systems. The available functions are *aio_read* and *aio_write* for asynchronous read and write requests. The functions receive a pointer to a so-called AIO control block of type struct aiocb containing the necessary information: a pointer to the data, the length of the data, the file offset, and the file descriptor. The aiocb pointer also serves as the request handle. The function *aio_error* can be used to test for the current state of a request. When *aio_error* indicates that a request has completed, *aio_return* can be used to obtain the amount of data that was actually read or written. Thus *aio_return* returns the value that a corresponding read or write system call would have returned. A request can be cancelled using *aio_cancel*. The function *aio_suspend* takes an array of pointers to aiocb structs and returns when at least one of the requests completes.

MPI-IO includes the two functions MPI_File_iread_at and MPI_File_iwrite_at which can read or write a block of data to a file at a specified offset. These functions return immediately, as the prefix i indicates, and return an MPI_Request structure. These are the same as those returned by the immediate send and receive operations, and hence one can use MPI_Wait, MPI_Waitany, MPI_Waitsome, MPI_Waitall, MPI_Cancel, and MPI_Test on them.

We tested three common MPI libraries, namely OpenMPI 1.4.5, IntelMPI 4.0 and MPICH2. We found that apparently OpenMPI and IntelMPI do not support the asynchronous I/O operations of MPI-IO, although they do provide the API functions. However, the read or write operations using MPI_File_iread_at and MPI_File_iwrite_at take a rather long time while the following MPI_Wait returns instantaneously. Only the MPICH2 library showed the desired behaviour, i.e., the read and write operations indeed returned quickly. The MPICH2 library also appears promising because it has quite an elaborate implementation of MPI-IO, contained in a subsystem called ROMIO [TLG04]. This ROMIO library in turn contains an abstraction layer called ADIO with several specialised implementations for common file systems in HPC. Among others, it supports the LUSTRE file system and incorporates a number of techniques that have been shown to result in good performance on this file system [DL09]. While we tested our software only with the LUSTRE file system, we expect that the implementations of ADIO for other common HPC file systems are similar. There are efforts to make the ROMIO software actually check its performance when running in a certain environment, by using formal performance expectations and requirements [Gro+08]. Hence, by using the MPICH2 library for our purpose we hope to obtain good performance on most of the common HPC file systems.

The class AsyncIO_AIO implements the interface AsynchronousIO by using the AIO routines. In this case the methods with name prefix “wait” are all implemented using *aio_suspend*. The method readData reads data synchronously and is implemented using a basic read system call. The class AsyncIO_MPIO implements AsynchronousIO using MPI-IO from MPICH2. In this case the method readData is implemented by calling the MPI-IO (blocking) function MPI_file_read_at. The methods with prefix “wait” are mapped to the MPI routines of the same name. However, we wrote our own version of MPI_Waitsome, as that function of MPICH2 appears to behave

identically to MPI_Waitall, i.e., it always waits rather long, until all pending requests have finished. This is obviously not our intention, instead we rather want it to behave more like MPI_Waitany, but with the capability to wrap up more than one completed I/O operation. The function myMPI_Waitsome is shown in Listing 11 in the appendix.

We also create two further implementations of the AsynchronousIO interface using blocking I/O routines, class SyncIO and class SyncIO_OSHints. This results in a stream buffer version which uses multiple buffer blocks but with blocking I/O. Write requests are performed immediately, but the information of read requests is stored and they are completed using a blocking read when the corresponding wait operation is called. The second variant in addition calls the function posix_fadvise with parameter advice set to POSIX_FADV_WILLNEED upon calls to iread to tell the OS that the data block in question will be required soon. This causes the OS to asynchronously fetch at least part of that data into unused memory pages [POS01]. Note that using multiple buffer blocks eliminates the redundant read problem from Section 4.5.2. Going back to the example situation in Figure 4, if we use two blocks of memory, one for data block i and the other for data block $i-1$, then we have to read block $i-1$ only once, if we keep data block i loaded in the other buffer. For this reason, using these two variants we may gauge the performance that we gain by avoiding the redundant read problem separately. So, this appears to be interesting for comparison.

4.6 Performance results

For quantifying the performance of RIOS, we consider the following two situations. The first situation is an artificially-generated test case incorporating a write and read pattern that is typical for a large-scale RM differentiated code. The focus of this test is on a comparison of the various iostream variants without taking into account too many additional issues of a real-world RM application. In the second situation, we apply the reverse mode of AD to a real-world application arising from fluid mechanics and compare the performance of RIOS in this more practical setting.

In the performance tests, we label the data with the name of the stack that is used, in the case of NATIVE-CELL, MEM, FSTREAM, and SSTREAM. In the case of ABUFFERED-STREAM, we label the data with the name of the underlying RIOS stream:

C–File A simple stream buffer which uses a plain C–style file as the backend, as described in Section 4.5.4.

SBB+Files An I/O stream using StreambufBlocker and BlockSinkSourceFiles, i.e., each block is written to a particular file using blocking I/O.

MB+SyncIO An I/O stream using StreambufBlocker and BlockSinkSourceAsyncIO, and at the bottom the class SyncIO, i.e., the AsynchronousIO implementation that uses blocking I/O (MB stands for the multi buffering done by BlockSinkSourceAsyncIO).

MB+OSHints Like MB+SYNCIO, but with pre-read hints to the OS via *posix_fadvise*.

MB+AIO An I/O stream using StreambufBlocker with BlockSinkSourceAsyncIO and the class AsyncIO_AIO, which is the AsynchronousIO implementation that uses the AIO asynchronous read and write functions.

MB+MPI-IO Like MB+AIO, but using the class AsyncIO_AIO, i.e. the AsynchronousIO implementation using the MPI-IO asynchronous read and write functions of the MPICH2 library.

Configuration parameters, system, and hardware In the case of FSTREAM, we set the buffer size to 5 kB, a rather small size chosen to avoid the problems with the buffering strategy, as detailed in Section 4.5.3. In the case of SSTREAM, we set the buffer size to 1 GB. With all of

the I/O streams with "MB" in the name, the total buffer size is set to 1 GB, split into 16 pieces of 64 MB each. In the case of SBB+Files, we set the buffer size to 64 MB.

The tests are run on the Linux cluster at the Center for Computing and Communication (CCC) of RWTH Aachen University. Computations are carried out on a single node of the Bullx Blade B500 cluster called "MPI-Small" of the CCC. Such a node consists of two Intel Westmere X5675 hex core processors running at 3.06 GHz. The system runs a Linux kernel version 2.6.32 and we use Matlab version 2012b and version 1.5rc1 of the MPICH2 library. Tapenade version 3.6 and ADiMat version 0.5.8 carry out the AD transformations for Fortran and Matlab. The MPICH2 library, the software stack described here, and the MEX functions are compiled with GCC 4.6. The tests are run through batch system requests, where the amount of available main memory is limited to 15 GB and the node is reserved exclusively. When data is written to disk the file is created on a LUSTRE file system attached via a fast network.

4.6.1 Test A: Artificial simulation code

To assess the performance of the different I/O streams we conduct an artificial test. This test simulates the data access pattern produced by the stack in a large-scale RM-differentiated code which does use recomputation for certain parts of the code. In this test, we first write a large amount of data and then read and write data in a zig-zag pattern, reading for example 3 GB and writing 1 GB until the stack is empty. The data reads and writes are done in chunks of 1 MB of data which is obtained from `/dev/urandom`. Before each read or write of such a chunk the test program multiplies a double array of size w MB by a scalar, and then divides it by the same scalar, where w is the work factor. This double array has $w \cdot 2^{17}$ items, and thus $w \cdot 2^{18}$ FLOPS are performed before each 1 MB write or read. These floating point computations constitute a pseudo workload simulating those of a real world RM-differentiated program.

In one test run, the zig-zag I/O pattern is performed with each of the various iostream implementations. The following six timing results are obtained in each case: The total time of the I/O pattern, labelled *total*, the read and write times, labelled *read* and *write*, the read and write calculation times, labelled *r-calc* and *w-calc* and the remaining time, labelled *rest*. The read time includes the r-calc time and likewise the write time includes the w-calc time. The total time is the time from program startup until termination and the rest time is calculated a-posteriori by subtracting the read and write times from the total time. The purpose of this is to assess the amount of program runtime that is not covered by the read and write times. Each test is performed N times and we compute the mean and the corrected sample standard deviation (SSD) σ of the three timings.

The particular tests are configured as follows: In Test *lustre-W2* and *lustre-W8*, we initially write 100 GB, and then alternatingly we read 3 GB and write 1 GB, until the size of the pseudo stack is 4 GB. This remaining data is then read again. This amounts to a total of 148 GB of data that is written and read. The work factor is $w = 2$ in Test lustre-W2 and $w = 8$ in Test lustre-W8. These tests are run $N = 32$ times.

The results are shown in Figure 8 and Figure 9. For each I/O stream implementation, the mean times of the N test runs are shown with error bars with a total length of 2σ. We also compute the corresponding I/O speeds, which are shown in Figure 10 and Figure 11. In our results the r-calc and w-calc are the same for all stacks, which is clearly expected. The read times and, hence, also the total times are fairly large for both the fstream and C-file stacks, but much larger for fstream. The write times of these standard components are fairly good. Moving on to the stacks SBB+Files and MB+OSHints, we see that these cause a significant improvement in the read times but a slight increase in the write times. The MB+OSHints I/O stream performs slightly better than the MB+SyncIO stream, which indicates that the calls to posix_fadvise actually do make a difference for the read times. For the sake of brevity, we do not report the results for the MB+SyncIO stream here, since these are fairly comparable, though slightly worse, than those of MB+OSHints. A further drop in both read and write times results from the stacks which use asynchronous I/O, i.e., MB+AIO and MB+MPIO. The performance of these two stacks is nearly identical. Moreover, for the work factor $w = 8$, the r-calc and w-calc times of these stacks

are almost identical to the read and write times. This indicates that, here, the I/O operations happen during the computations. However, for $w = 2$, the I/O times are still larger than the calculation times. That is, the amount of computation is too small to fully hide the overhead caused by data I/O.

Our conclusion from these tests is that the buffering done by the standard I/O facilities in both C++ and C is inadequate for reverse reading. A large speedup can be gained from suitably tailored buffering strategies. For example, in the Test lustre-W2, using the MB+OSHINTS stream results in a performance increase in the total runtime by a factor of more than 2 over standard C file I/O, with a total time of 1585 s for C-FILE and 626 s for MB+OSHINTS. Using asynchronous I/O on top of that gives us an additional performance increase of about 28 %, with a total time of 487 s for the MB-AIO stream. The advantage gained from using asynchronous I/O is even larger with an increased computational workload. In Test lustre-W8, we measure an additional performance increase of almost 50 % when comparing MB-OSHINTS and MB-AIO. More precisely, the total time of 1108 s for MB-OSHINTS decreases to 741 s for MB-AIO. On the other hand, the I/O speed is larger when there are less computations. In Test lustre-W2, we achieve read and write speeds of 600 MB/s and above, which is fairly close to the best results achieved on LUSTRE systems [TKC13]. Also in this test, the best read speed (and runtime) is actually achieved with MB+OSHINTS. However, when in comes to writing and to larger computational workloads as in Test lustre-W8, the performance is clearly superior for asynchronous I/O.

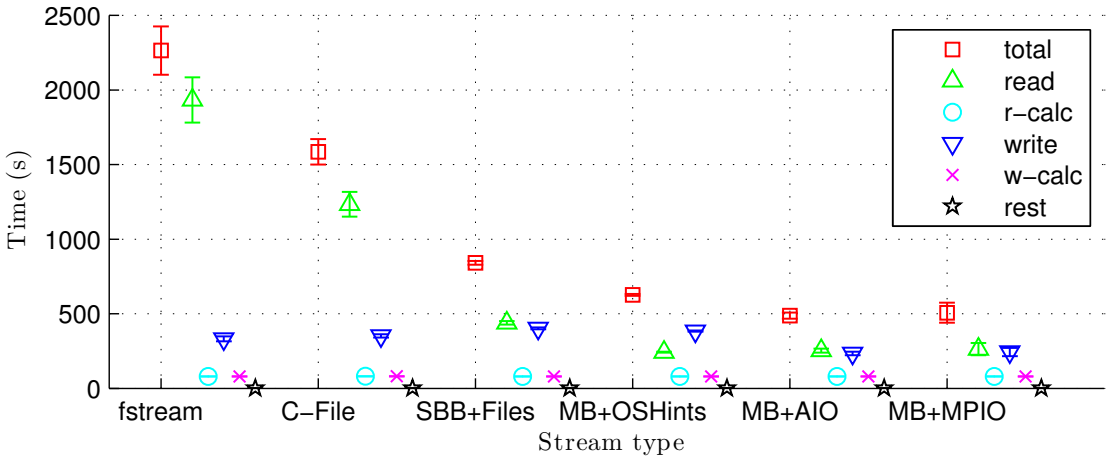

Figure 8: Timing results for Test A with a work factor $w = 2$.

4.6.2 Test B: Solution of Burgers equation

In the second test, we consider a code that solves the Burgers equation in one space dimension with periodic boundary conditions [JX95]. The simulation time is $t_{\text{end}} - t_0 = 3\,\text{s}$, the spatial resolution uses n_x grid points and second-order terms are used. In Figure 12, we show the initial state $u_0 = u(t_0)$ and the final state $u_{end} = u(t_{end})$ of the simulation, computed with $n_x = 200$ grid points. The wave imposed as the initial condition travels towards the right and a sharp shock is developed. We consider a scalar cost function which compares the solution at the simulation end time t_{end} to a given solution. This scalar function is differentiated in reverse mode, computing the gradient of the cost function w.r.t. the initial solution. Thus, the length of that gradient is n_x.

The code for the solution of Burgers equation, originally written in Matlab by Michael Herty of RWTH Aachen University, serves as a test example for shocks in a hyperbolic partial differential equation (PDE). This Matlab code is translated to Fortran90 and C++. To construct an example for RM differentiation, we compute the L_2-norm of the final state u_{end}. The gradient of this

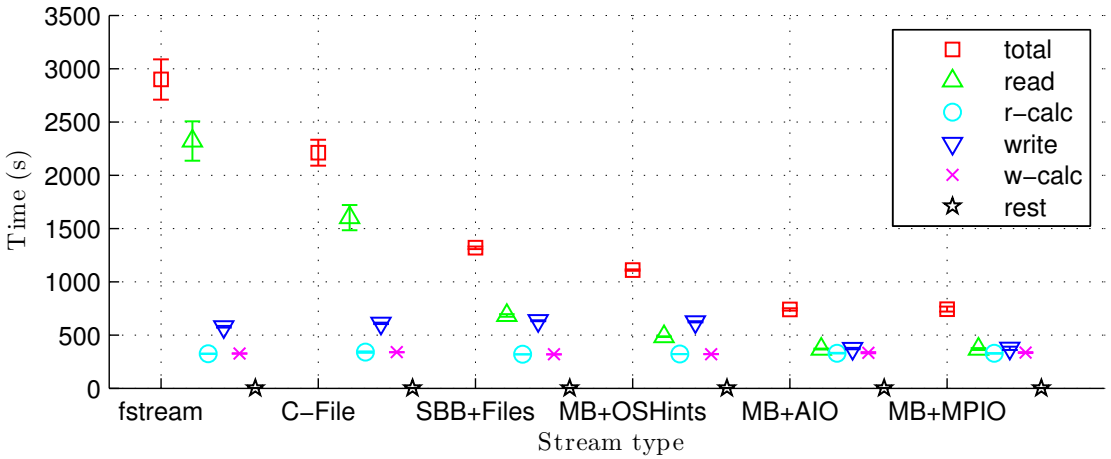

Figure 9: Timing results for Test A with a work factor $w = 8$.

scalar result can be computed in Matlab by RM differentiation with ADiMat, in Fortran90 by RM differentiation with Tapenade. We also differentiate the C++ version with ADOL–C [GJU96] in RM. Table 1 gives some performance values for this test problem. In that table, we show the runtimes and stack sizes for the particular test case with $n_x = 2000$. We present the runtime of the function evaluation t_f in seconds, the runtime of the gradient evaluation t_∇ in seconds, the differentiation overhead factor t_∇/t_f, and the size of the stack or tape produced by the AD tools in MB. We use ADOL–C version 2.3, Tapenade version 3.6, and ADiMat version 0.5.8. For comparison, we also approximate the gradient using central finite differences (FD) in Fortran90. We run the tests with ADiMat using the stacks NATIVE-CELL, MB+SyncIO, MB+MPI-IO and the test with Tapenade using the supplied in-memory stack, and the RIOS stacks MB+SyncIO and MB+MPI-IO. The RIOS stacks are configured to use 1 GB of main memory. The value for t_f in each case is the runtime of the plain non-differentiated program. The ADOL–C results are intended to serve as a reference value. We configure ADOL–C to also use 1 GB of main memory for the tapes, roughly divided in proportion to the total size of each of the four tapes produced, and the time t_∇ in this case includes the taping and the gradient evaluation with a call to the fos_reverse function[3]. All the programs are compiled using the GCC compilers version 4.7.2 with optimisation flags `-O3 -march=native`, and in each case the files go to a LUSTRE file system attached via a fast network. Since ADOL–C also writes and reads its data in blocks, we expect the performance to be comparable to that of MB+SyncIO or MB+OSHints, and the gain that could be achieved by using the asynchronous I/O features of RIOS with ADOL–C to be about 30 % to 50 %, as suggested by the corresponding results in the previous Section 4.6.1.

The results for the runtime to evaluate the function, t_f, show that the fastest language is Fortran90; computing the function in C++ takes more than twice as long and this time is still somewhat longer using Matlab. We now focus on the time t_∇ needed to compute the gradient. As expected, computing the $n_x = 2000$ components of the gradient with FD takes almost exactly $2n_x = 4000$ times as long as the time t_f. Computing the gradient in RM using Tapenade, we achieve a very low runtime and an overhead factor close to the theoretical optimum of 3. During the computation of the gradient, Tapenade produces a stack of almost 1 GB. When this stack is written to disk using RIOS the runtime increases, although the penalty is small for using the MB+MPI-IO. Using ADiMat, the serialised stack size is slightly more than 4.5 GB, and both RIOS stacks perform better than NATIVE-CELL, although in that case the entire stack remains in main memory. ADOL–C produces a tape of more than 20 GB and its runtime is the largest of all tested configurations. While these test runs are spot checks, the numbers reflect the advantages

[3]Download the source code of this test from http://rios.ourproject.org/

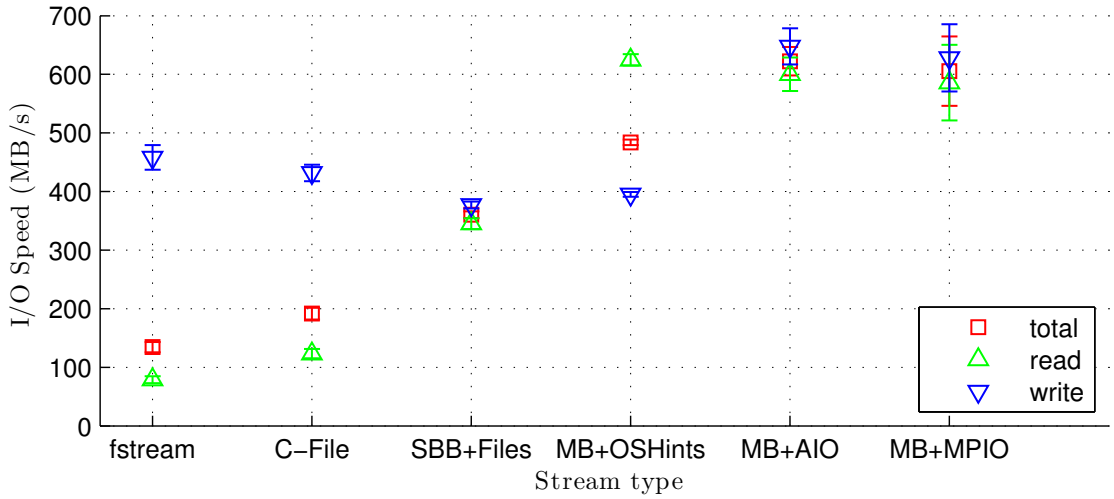

Figure 10: I/O speed results for Test A with a work factor $w = 2$.

Language	AD Tool	t_f(s)	t_∇(s)	t_f/t_∇	Stack (MB)	I/O Speed (MB/s)
C++	ADOL–C 2.3	1.051	108.5	103.2	21605	199.1
Fortran90	Tapenade 3.6	0.517	2.31	4.469	913.6	395.5
Fortran90	+ MB+SyncIO	0.517	4.548	8.797	913.6	200.9
Fortran90	+ MB+MPI-IO	0.517	2.638	5.102	913.6	346.4
Fortran90	FD	0.517	2064	3992	0	0
Matlab	ADiMat 0.5.9	1.277	79.88	62.57	–	–
Matlab	+ MB+SyncIO	1.277	69.52	54.46	4560	57.09
Matlab	+ MB+MPI-IO	1.277	56.71	44.42	4560	80.41

Table 1: Function runtime t_f, gradient runtime t_∇, overhead factor, stack size, and I/O speed for Test B with $n_x = 2000$ using various AD tool configurations and programming languages.

of AD based on source transformation over operator overloading and tracing based techniques and also the advantages of hierarchical AD approaches [BH96; Büc02; Gil08; TFP03] that are prevalent in "higher-level languages" with builtin vectors and vector arithmetic.

Now consider Figure 13 and Figure 14, showing a qualitative impression of the stack history. To produce these plots, we augment the RM source code with special statements that query the current time, number of items on the stack, and the size of stack in bytes. This information is taken upon entering and leaving each of the functions in the adjoint code and, in addition, at the reversal point which is the point in the top-level adjoint function where the forward sweep ends and the reverse sweep starts. At this point the stack is largest. To compare the different stack implementations, we shift the time information so that the reversal point of each run is placed at time $t = 0$. In Figure 13, we plot the number of items on the stack over time of a small test case where $n_x = 1585$. In Figure 14, we plot the size of the stack in MB of a large test case with $n_x = 6310$. In the small test case, the total stack has a size of about 2.9 GB and can be kept in main memory. Hence, we test all available stacks for the small test case. For the two stacks NATIVE-CELL and MEM, we do not have the stack size information. Also, in the large test case, the stack has a size of about 46 GB and, thus, we can test only those stacks that support writing data to the hard disk.

The lines labelled "ideal" indicate an ideal forward sweep which takes time t_f with no additional overhead followed by an ideal reverse sweep which takes time $3t_f$. These lines run from point $(-t_f, 0)$ in the figures to the largest stack at time $t = 0$ and back down to the point $(3t_f, 0)$. We have $t_f = 1.1$ s in the small and $t_f = 12.66$ s in the large case. Also, the data labelled "None" are

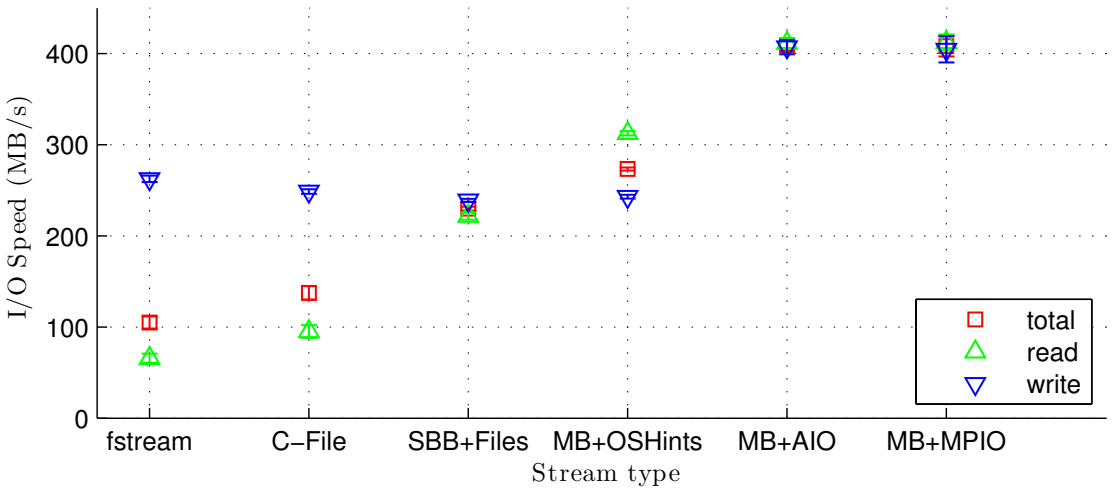

Figure 11: I/O speed results for Test A with a work factor $w = 8$.

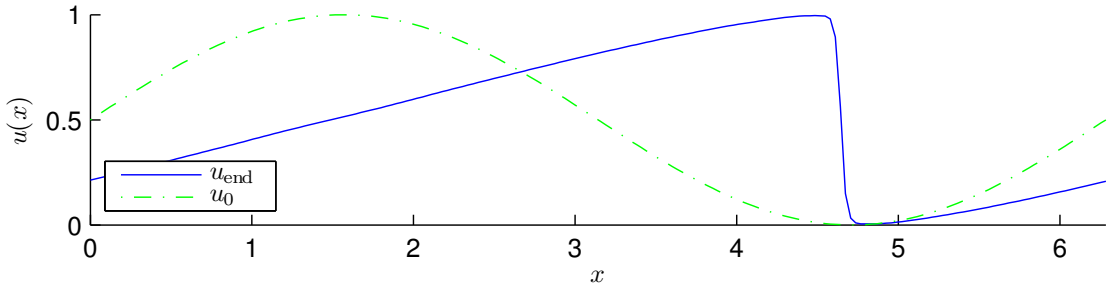

Figure 12: The initial condition u_0 of the Burgers equation and the final state u_{end}.

measurements of a stack similar to NATIVE-CELL but where the **push** function does nothing except to count the number of pushed items. The idea here is that we wish to gauge the performance of the augmented forward sweep code, but excluding the actual stack operations. When using this stack, the reverse sweep cannot be run and so we do not have the data of that. When showing the large case results, the stack "None" is represented by a straight line from the point $(-t_{For,None}, 0)$ to the top of the stack, where $t_{For,None}$ is the time of the forward sweep using that stack, since we do not have the stack sizes in this case.

From the results we see that the reverse sweep generally takes longer than the forward sweep, which is expected. We also see that the stacks implemented as MEX functions, except FSTREAM, are faster than those written in Matlab. Some of the MEX stacks which do save the data are even faster during the forward sweep than "None", which does nothing. The stair-like appearance of the data for the stack FSTREAM is due to the Matlab process hanging in wait for I/O completion. The particular shape of the stairs is different for each test run. These waiting periods are smoothed over by using asynchronous I/O. In the small test case, we see that the stacks using asynchronous I/O to disk are only marginally slower than those MEX stacks that keep the data in memory. In the large test case, the asynchronous IO stacks save about 50 % of the time. Note that with the stack FSTREAM there is a period of very slow I/O towards the end of the reverse sweep, for example at $t = 350\,\text{s}$ the stack still has a size of about $350\,\text{M}B$.

Next, we conduct tests with varying problem size n_x ranging from 10^2 to $10^{4.5}$ in 26 logarithmically equidistant steps. The results are reported in Figure 15 to Figure 22. For each stack

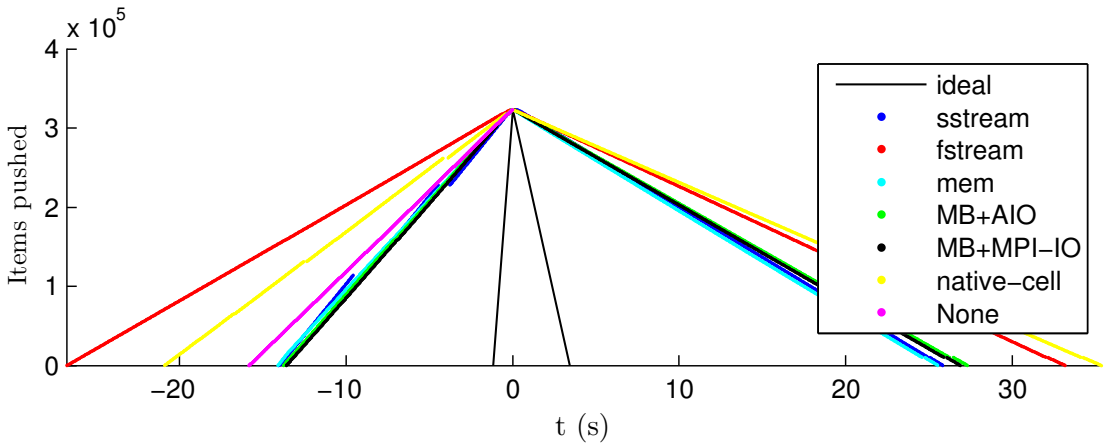

Figure 13: The stack history for Test B of various I/O streams in terms of the number of items for a small test case, $n_x = 1585$.

implementation, we run the tests in ascending order of n_x. When a test run takes longer than a threshold time $t_s = 600\,\mathrm{s}$, the test series is aborted. In Fortran, we limit the virtual memory using the ulimit command to 4 GB. In Matlab, we run the tests with in-memory stacks only up to $n_x = 3000$ to avoid running out of memory. In the case of SSTREAM there appears to be a bug in the stdc++ library shipped with Matlab, which causes a crash when the stack becomes larger than about 2.5 GB. The correct behaviour would be to set the fail-bit of the iostream and possibly raise an exception indicating the exhaustion of memory. Hence, with this stack we run the tests only up to $n_x = 1600$. Each test is run $N = 16$ times to gauge the variance in the test results.

The RM differentiation uses the so-called save-all strategy or split mode which means that the adjoint code consists of one large forward sweep followed by one large reverse sweep of the complete code [GW08a]. In ADiMat this is the default, while in Tapenade a strategy based on recomputation and checkpointing is the default, but we set the save-all strategy for all subroutines via options. It should be noted that in a real-world application of the RM of AD one will probably do make use of recomputation or checkpointing strategies in order to achieve a smaller stack size. However we deliberately use the store-all approach in order to illustrate the capabilities of RIOS.

In Figure 15, the runtimes t_f of the cost function over varying n_x are shown. We show the mean values of the $N = 16$ test runs with error bars of length of 2σ. As can be seen in the figure, the runtime rises from about 0.03 s for $n_x = 100$ to 303.89 s for $n_x = 2.51 \cdot 10^4$. The variance in the timings is rather small, except for the two largest values of n_x. The runtime rises superlinearly because of the adaptive time stepping, which takes smaller time steps with larger spatial resolutions and thus the number of time steps also rises with n_x. Also in Figure 15, we show the bandwidth theoretically required by the forward sweep of the adjoint function, labelled *ideal bandwidth*. This is obtained by dividing the size S of the stack by the average runtime t_f. We see that this appears to flatten off at a couple of GB/s for the larger problem sizes. The data labelled *measured bandwidth* is $2S$ (since the data is written and read) divided by the best gradient runtime, obtained with MB+MPI-IO, which is also shown in the figure.

In Figure 16, the stack sizes are plotted over n_x. The blue line, with the scale on the left, shows the size of the stack in MB, while the green line, where the scale is on the right, shows the number of items that are pushed on the stack.

In Figure 17 the runtimes, t_∇, of the differentiated function are shown. For smaller values of n_x, we see that the in-memory stacks implemented as MEX functions take almost exactly the same time. This shows that the serialisation layer does not produce a large overhead. The stacks NATIVE-CELL, FSTREAM and C-FILE are noticeably slower, the first probably because of

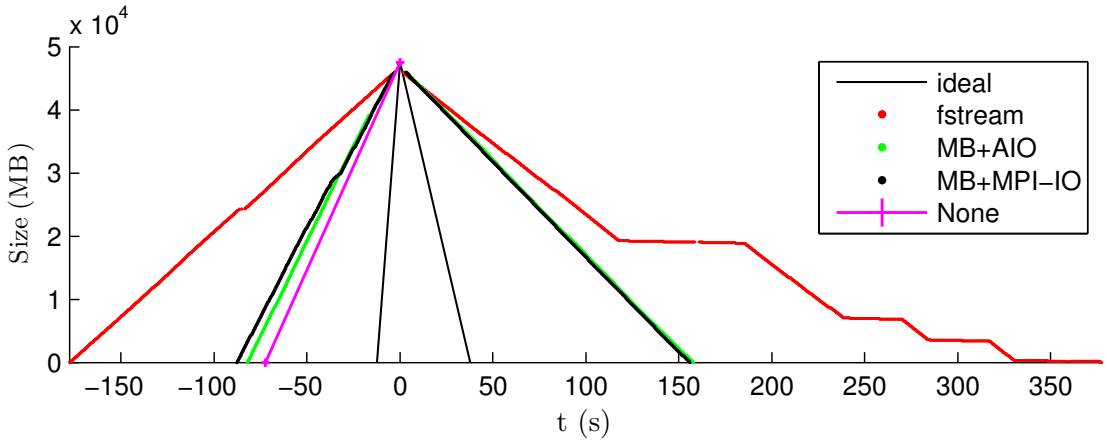

Figure 14: The stack history for Test B of various I/O streams in terms of the stack size for a large test case, $n_x = 6310$.

the interpretation time of the Matlab code and the second because of the file I/O.

In Figure 18 the overhead factor t_∇/t_f of the previous tests are shown. In this case where we run $N = 16$ instances of each test we represent this quantity in the plot as follows. This ratio is computed for each test instance individually, and then we compute the mean and the SSD of these ratios. These results show that for most stack implementations there is a clear downwards trend with increasing n_x. For problem sizes $n_x > 1000$ the mean overhead factor is about 30 or less.

In Figure 19, Figure 20, Figure 21, and Figure 22 we show the corresponding results for Fortran90 differentiated in RM with Tapenade. The timing results show that here the advantage of using RIOS over plain C or C++ I/O is not quite as large, however it is still significant. The reason might be that there are not as many small data items since there is no serialisation control data. Again however, the main performance enhancement seems to stem from the block forming layer, and asynchronous I/O gives a small yet considerable boost on top of that. Also we see again that there is no penalty of using RIOS over in-memory stacks for small to medium problem sizes. The stack sizes and also the bandwidth estimates are much smaller, but also closer together, than in the case of Matlab/ADiMat. This is probably the effect of the to-be-recorded analysis in Tapenade which apparently saves a lot of data from being pushed unnecessarily.

4.7 Conclusion and Future work

We design, implement, and evaluate a layered software architecture for the out-of-core storage of stack data to be used in the reverse mode (RM) of automatic differentiation (AD). This software architecture is called RIOS and provides a standard C++ I/O stream that abstracts from the underlying implementation. A multi buffering strategy and asynchronous I/O operations are used to enable the efficient retrieval of large-scale data from file streams in reverse order. This data access pattern is commonly required in the RM of AD and, more generally, in trace-based AD tools. In particular, we provide for the peculiar data access pattern of a stack, i.e., consecutive write and reverse read operations. To this end, RIOS implements a carefully designed buffering strategy that facilitates the switch from writing to reading. It is currently used to provide efficient stacks for the RM in the AD tool ADiMat for Matlab. However, by additionally carrying out numerical experiments with the AD tool Tapenade for Fortran, we give evidence that RIOS also provides a crucial infrastructural building block in the more general context of RM AD and also of trace-based tools.

RIOS provides several different stack implementations whose performance is evaluated using

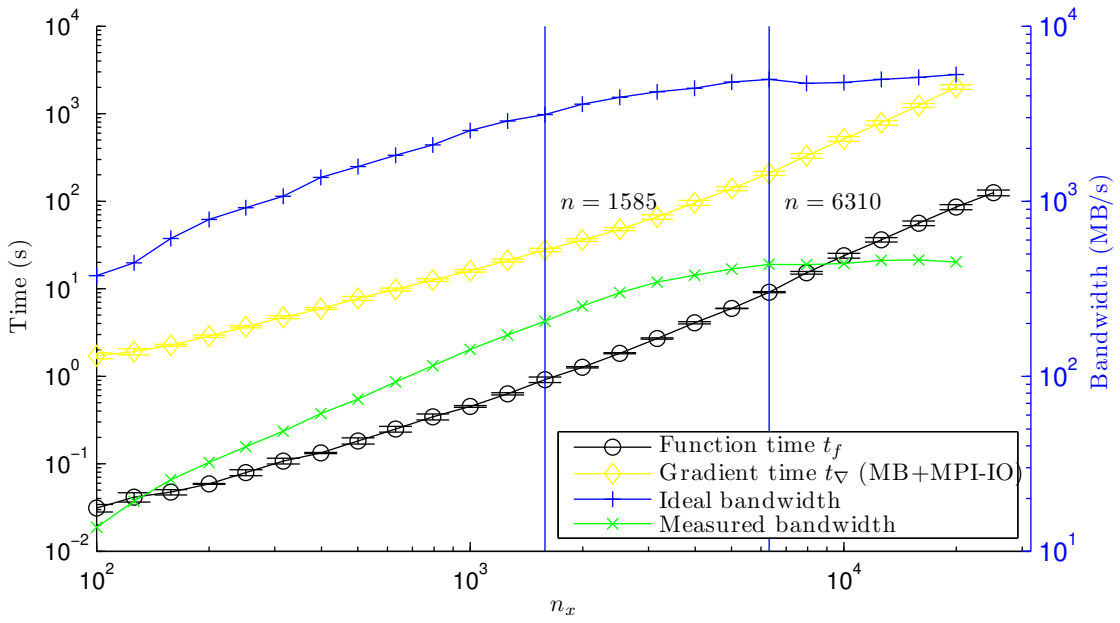

Figure 15: Function and gradient runtimes (left axis) and stack bandwidths (right axis) for Test B and Matlab/ADiMat, $N = 16$.

two test cases. As the first test case, an artificial simulation code is considered that mimics a typical storage access pattern occurring in the RM of AD. The second test case arises from fluid mechanics and consists of a code that solves the Burgers equation. Both cases are investigated using codes written in Matlab as well as Fortran. The AD transformations are carried out using the AD tools ADiMat and Tapenade, respectively. These performance tests demonstrate substantial savings in runtime using RIOS to implement stacks for the RM of AD.

We foresee several potential directions of future research and enhancements. One might split the data stream into different categories as suggested in Section 4.4.2 and also apply suitable compression techniques to the control streams. We would also like to test RIOS with trace-based AD tools such as ADOL–C or CppAD. Furthermore, RIOS is also relevant outside the field of AD; other software that produce large traces of programs could benefit from RIOS, in particular if the traces are read in reverse direction. One could also investigate device parallelism for storing and retrieving the data, for example, distributing the data to the local disks of multiple hosts via the network. This would increase the average I/O bandwidth. In our studies, we also compare two different flavors of asynchronous I/O. More precisely, the POSIX AIO programming API is compared with the corresponding functions provided by the MPI-IO component of MPI 2. In doing so, we use the MPI-IO library on a single host for the sole purpose of asynchronous I/O. While we found only small differences in the performance between these two approaches, using MPI 2 might allow for more interesting usage patterns for high volume data I/O in the future.

4.8 Source code listings

Custom version of MPI function MPI_Waitsome

The function *myMPI_Waitsome* whos source code is shown in Listing 11 has the same semantics as *MPI_Waitany*, but behaves differently from the MPICH2 version; see Section 4.5.5 for the corresponding discussion.

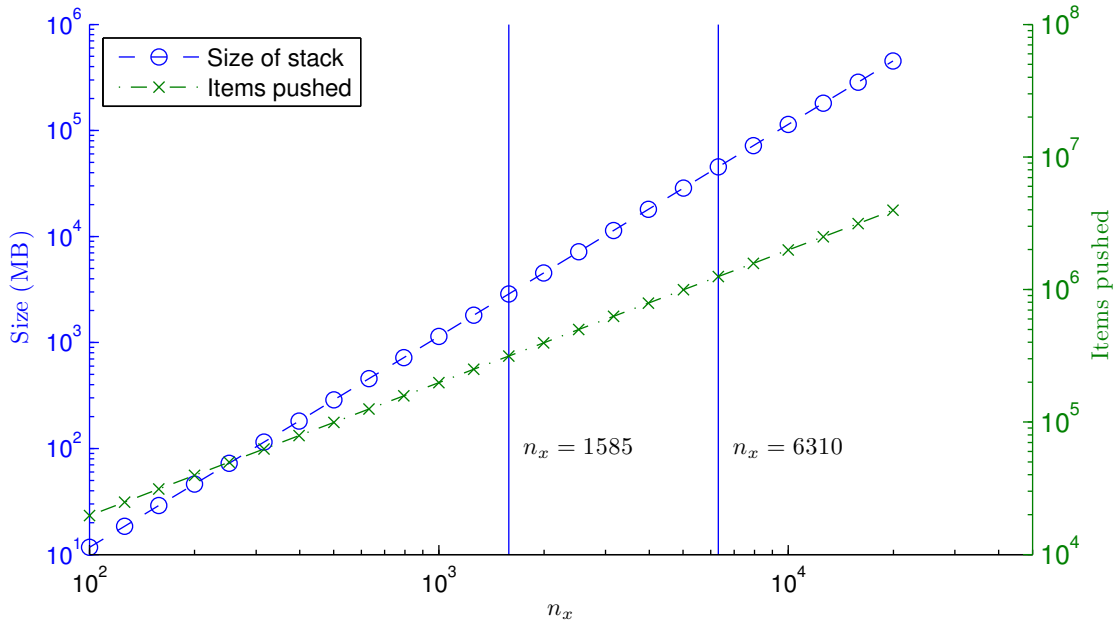

Figure 16: Stack size (left axis) and number of items pushed on the stack (right axis) for Test B and Matlab/ADiMat.

```
1  int myMPI_Waitsome(int cnt, MPI_Request *requests, int *numCompleted,
       int *completed, MPI_Status *statuses) {
2    int res = 0, flag;
3    bool done = true;
4    *numCompleted = MPI_UNDEFINED;
5    for (int i = 0; i < cnt; ++i) {
6      if (requests[i] != MPI_REQUEST_NULL) done = false;
7    }
8    if (done) return res; // all are MPI_REQUEST_NULL -> return
9
10   *numCompleted = 0;
11   do {
12     for (int i = 0; i < cnt; ++i) {
13       if (requests[i] != MPI_REQUEST_NULL) {
14         res = MPI_Test(requests + i, &flag, statuses + *numCompleted);
15         if (flag) {
16           completed[*numCompleted] = i;
17           ++*numCompleted;
18         }
19       }
20     }
21   } while (*numCompleted == 0);
22   return res;
23 }
```

Listing 11: The implementation of MPI_Waitsome we use instead of the one in MPICH2.

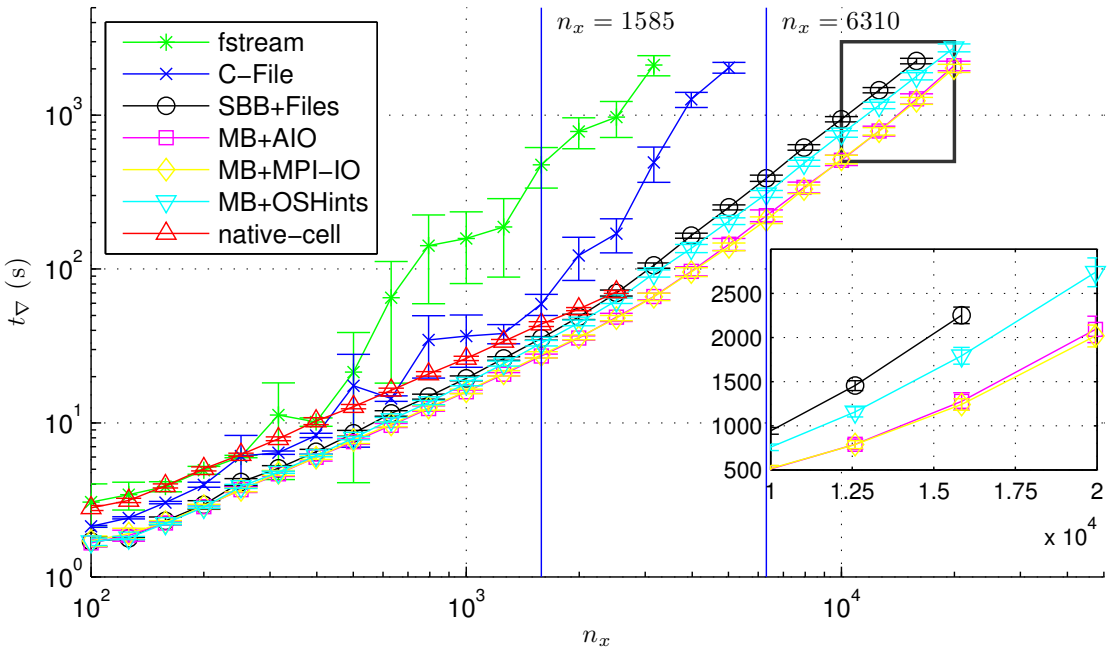

Figure 17: Gradient runtimes using various different stack implementations for Test B and Matlab/ADiMat, $N = 16$.

5 The differentiation of selected MATLAB toolbox functions and builtins

In this sections we want to discuss the derivative propagation steps that are the done for each of the elementary operations in MATLAB. Since we can devolve MATLAB basic block code in principle to a sequence of assignments with nested function calls on the RHS, we can safely equate a builtin function with an elemenary operation in MATLAB. Hence, in ADiMat we have the situation that the language has very many elementary operations, compared to Fortran or C, for example. Accordingly ADiMat still supports only a small subset of all available builtins even though quite a few have been implemented over the years. In this section we want to present some general concepts to the structure of the propagation steps and also present some builtins exemplarily in more detail, showcasing the ideas.

The derivative propagation steps frequently employ common patterns, which we would like to discuss first in Section 5.1. We identify the three generic cases of

arithmetic propagation where a partial derivative exists in the form of a matrix, which is ideally sparse or even diagonal

structural propagation where the derivative can be created by direct manipulation

algorithmic propagation where we apply AD to a devolved form of the builtin

We saw in the introduction section to the derivative classes 2.5 that the reshape-to-vector-and-matrix-multiply operations `d_x(:) * M` or `M * d_x(:).'` are by far the most efficient of the array based derivative class, so arithmetic propagation is clearly preferable. A clear advantage is of course also that the same derivative is used in FM and RM. This technique is discussed in the Section 5.1.1 in more detail.

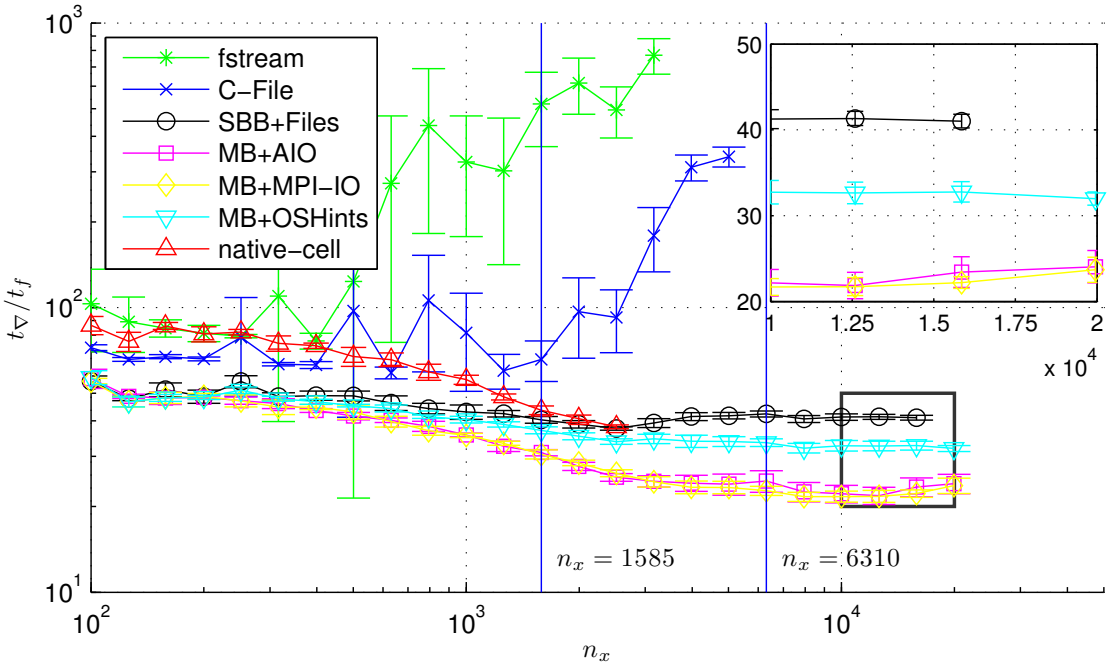

Figure 18: Overhead factors using various different stack implementations for Test B and Matlab/ADiMat, $N = 16$.

Sometimes it may be advantageous however to perform the required operation directly on the derivative, with structural propagation. This technique is discussed in the Section 5.1.2. For example, in the FM, the derivative code for `mean(x,i)` is simply `mean(d_x,i)`. In the case of the vector mode, the derivative class method **mean** will then call `mean(s.m_derivs, i+1)` on the internal array, and thus compute the derivative for all derivative directions with a single call, which is probably as efficient as it gets. In this case and very many others, it is the very same function that is needed in the structural propagation in FM, that is, the functions often differentiate to themselves. For many matrix operations beginning from matrix multiplication **mtimes** to the eigen decompostion **eig** and the singular values decompostion **svd** structural adjoint propagations are fortunately known [Gil08] and the corresponding rules have been incorporated into ADiMat.

Finally, we often have the last option to express the builtin in terms of a devolved expression with only functions and operations that ADiMat already supports. This is used to write a *replacement function* for the builtin in question and then we let the devolved function be differentiated by ADiMat. When the source code of the builtin is available, the better, but usually it is not. We we call this option *algorithmic differentiation*, and it is discussed in more detail the Section 5.1.3.

This option is also available to ADiMat users as an escape hatch when some builtin is not supported by ADiMat. For example, when the builtin **xyz** is called in the code, users may replace this by a call to **myxyz** and write a suitable function **myxyz** that performs the same operation as **xyz** and can be differentiated by ADiMat. The ADiMat developers can also use this method. Interestingly, exactly this strategy, was proposed as the only viable option to differentiate the matrix exponential **expm** in MATLAB [AMH09]. So we literally write the MATLAB code of the suggested Padé approximant as a replacement function of **expm**, and then differentiate it with ADiMat in both FM and RM to then distribute the resulting code with the ADiMat runtime. For certain other matrix operations such as **hess** and **qr** the differentiation of the underlying algorithm may be the only viable option as well. In AD, the differentiation of iterative algorithm is an established technique. In terms of performance the algorithmic method is quite costly. This

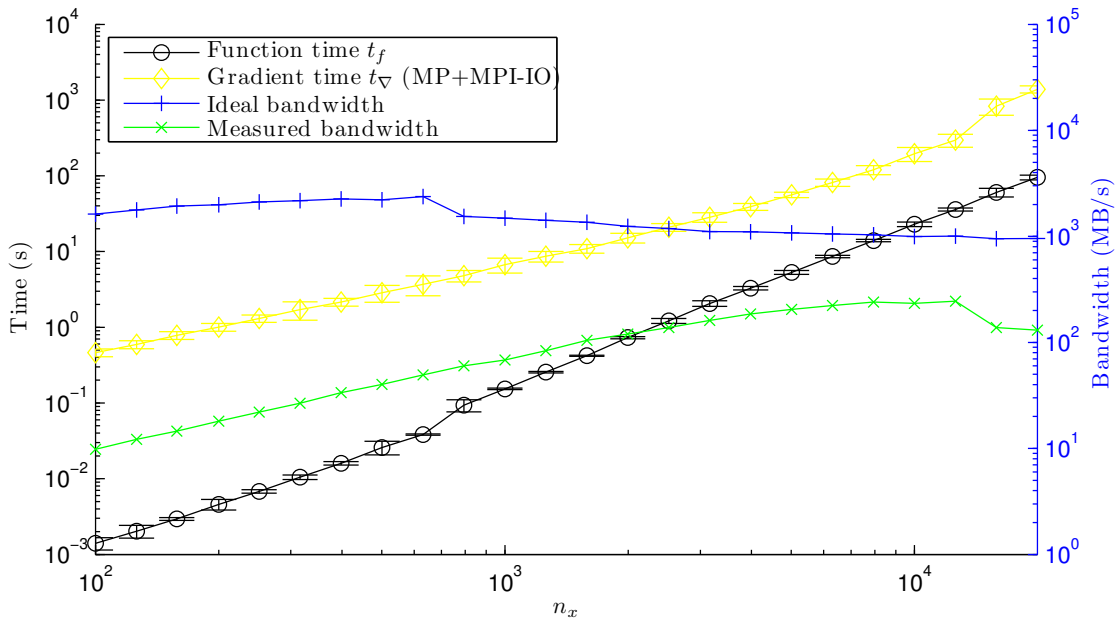

Figure 19: Function and gradient runtimes (left axis) and stack bandwidths (right axis) for Test B and Fortran/Tapenade, $N = 16$.

starts with the development effort for a suitable replacement function, but the computational cost of differentiating an iterative algorithm can also be substantial.

To discuss some practical examples, we have a case study of the Legendre function in Section 5.2. To exhibit some cases that are particularly interesting in the reverse mode, we discuss four the multiplicative operators **times**, **mtimes**, **conv**, and **kron** in Section 5.3.

5.1 Generic approaches to the differentiation of toolbox functions and builtins

In this section we describe the generically available methods for introducing support for a given builtin in ADiMat. Each builtin performs a certain mathematical operation for which the correct propagation rules in forward and reverse have to be specified. A generic method is to construct the local Jacobian and multiply the derivative or adjoint with it, as described in Section 5.1.1. In many cases it is also advantageous to use so called structural propagation, as described in Section 5.1.2, although this routinely results in different rules for the FM and the RM. Finally, one can obviously also apply AD to given function implementation to obtain its derivative, as described in Section 5.1.3. This last approach is relatively costly regarding the performance but in some mathematically challenging cases the only one available.

5.1.1 Arithmetic propagation

By arithmetic propagation we refer to the provision of the partial derivative or local Jacobian of a given operation. The partial derivative is provided as a dense and full, sparse or even diagonal matrix. These cases are implemented in ADiMat by providing a runtime function with name prefix *partial_*, for example `partial_xyz` to return the local Jacobian for a builtin **xyz**. Given the local Jacobian `J` the forward mode rule is roughly speaking `J * g_x(:)` followed by a reshape to output size in FM and `a_x(:).' * J` followed by a reshape to the input size in RM.

This is the classical approach and usually also the preferable. Firstly, the same partial derivatives can be used in FM and RM. Secondly this is the operation that works most efficiently with

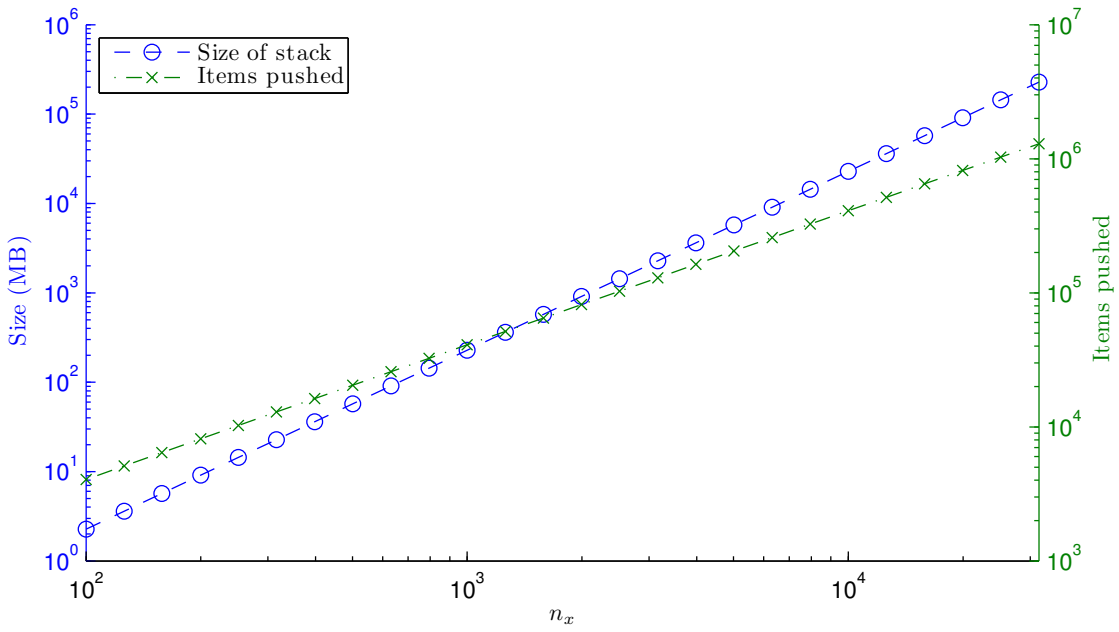

Figure 20: Stack size (left axis) and number of items pushed on the stack (right axis) for Test B and Fortran/Tapenade.

the array based derivative class (Section 2.5). Thirdly, the work to construct the local Jacobian is independent of the number of directional derivatives. As a downside, space is required to store the Jacobian.

Cases where the local Jacobian is actually densely filled are rather seldom, notable examples being the builtins **roots** and **poly**, where obviously each input value influences all the output values.

On the other hand an abundantly occuring case is constituted by the numerous vectorized component-wise builtins in MATLAB, such as **sin**, **sqrt**, **exp**, etc. or the Bessel functions. Here the local Jacobian is diagonal, so the multiplication reduces to a vectorized call to **times**.

These cases are implemented in ADiMat by providing a runtime function with name prefix `dpartial_`, for example `dpartial_xyz` to return the diagonal of the partial for a builtin **xyz**. Given the diagonal as a plain vector `d` the propagation rule is `d .* g_x(:)` in both FM and RM, followed by a reshape.

The local Jacobians of the vectorized reduction operations such as **mean**, **std**, **trapz**, etc. also have a common general structure that can be produced efficiently by a double **kron** product of sparse matrices, given the particular basic block matrix for the reduction operation in question. This is also true for the cumulative reductions like **cumsum** and **cumtrapz**, where the basic block is not a vector but a lower triangular matrix.

With the **legendre** functions discussed in Section 5.2 we saw a special case where each input component yields a vector of results. Thus the Jacobian is a block diagonal of column vectors, which as an operator can be expressed as a **bsxfun** of the **times** operation, when the Jacobian is presented in compressed form.

Finally, for indexed array selections, that is, the **subsref** and **subsasgn** operations, the most viable approach also is to construct the local Jacobians, which is possible to do efficiently. In particular this approach also covers those cases in the RM where a slight difference in semantics regarding the repeated occurence of indices prohibits the structural approach, see Section 3.3.

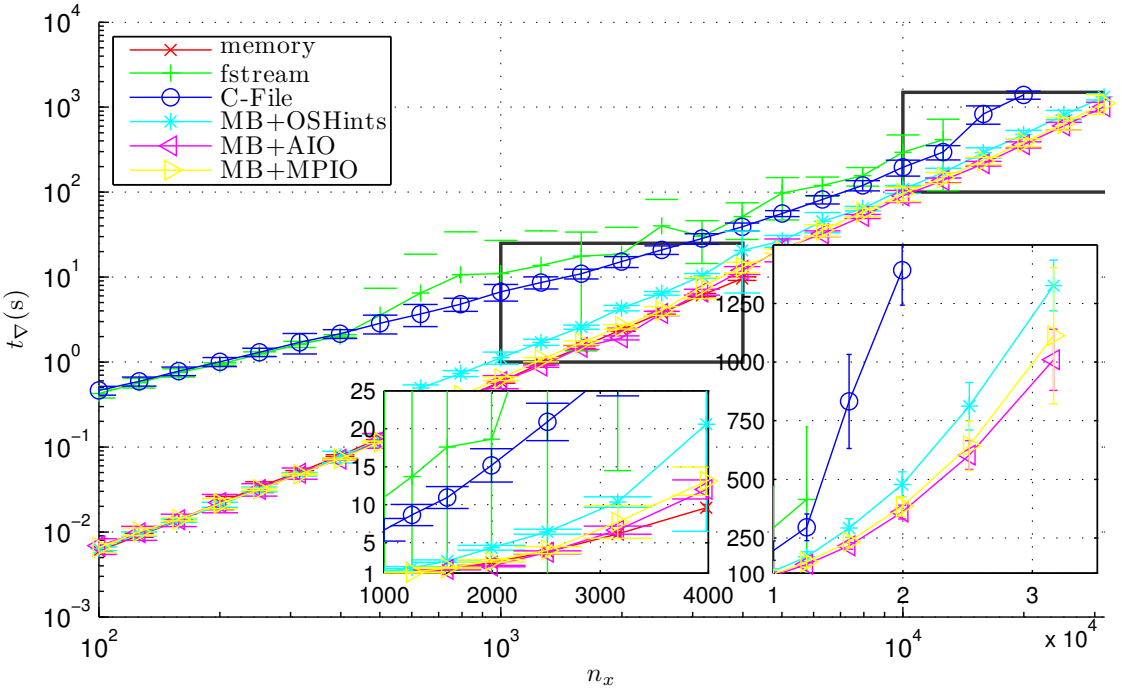

Figure 21: Gradient runtimes using various different stack implementations for Test B and Fortran/Tapenade, $N = 16$.

5.1.2 Structural propagation

By structural propagation we refer to all other cases, where any other kind of MATLAB operation, except multiplication by the local Jacobian, is carried out on the derivatives. For example, ADiMat uses structural propagation for builtins like **sum**, **mean**, **diff**, **fft**, and many data rearrangement functions like **flipdims**, **reshape**, **permute**, **shift**, etc. and strictly speaking also for the basic operations on the data structures of cells and structs that are supported in the RM.

For vectorized functions such as **mean** there usually is an efficient implementation for these methods in the array-based derivative class, since in most cases the vector based function in MATLAB allows us to handle the operation on the derivative object in a single call, see Section 2.5.

In particular when the operation in question is a modification of the data structure, the derivative simply follows suit, as discussed in the Section 3.1 on the data model we use, and in Section 6.5.1 we discussed the storage expressions in AST XML and the corresponding differentiation of such expressions. In such a case, as in many others, a partial derivative in the form of a matrix to multiply with does not exist, so we simply must use structural propagation. A mathematical example for such a case are **real** and **imag**, which are not complex analytic and hence do not have a partial derivative, but which also differentiate to themselves in FM. This looks dangerously like the data manipulation case for structural propagation also applies here, but as discussed in Section 3.5 already this cannot be, and in fact this case is more involved, as discussed in Chapter 7.

For many matrix operations beginning from matrix multiplication **mtimes** to the eigen decompostion **eig** and the singular values decompostion **svd** structural adjoint propagations can be derived using matrix arithmetic with a technique using the trace operator [Gil08]. The results from that work and the corresponding propagation rules have been incorporated into ADiMat.

A downside of structural propagation is that usually the propagation rules for FM and RM

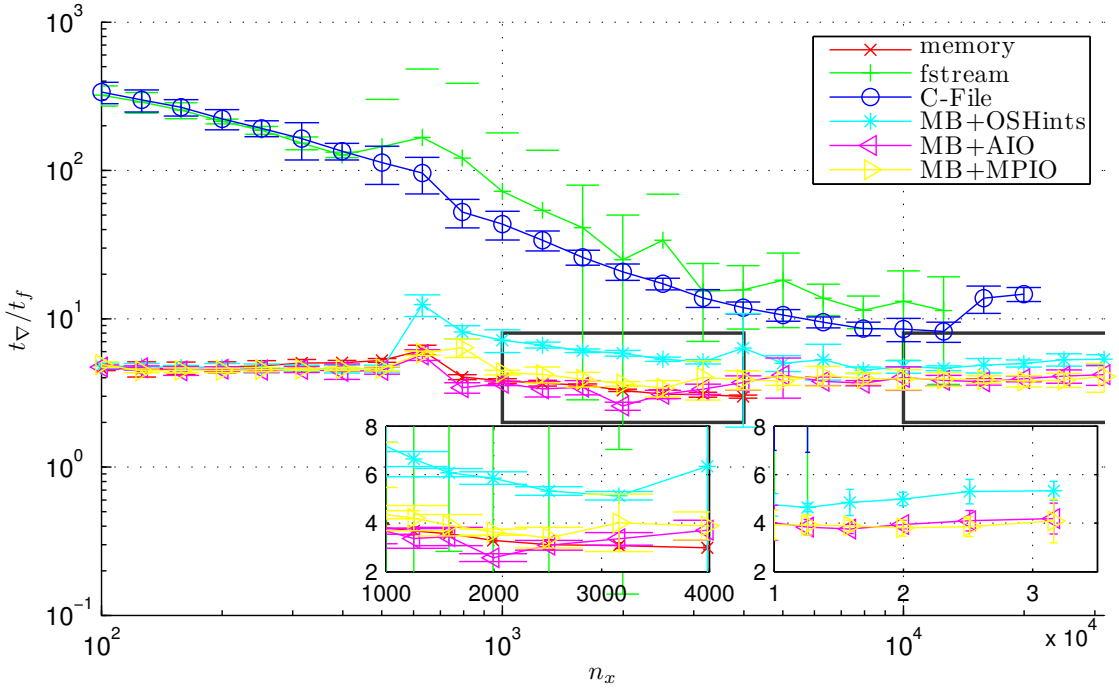

Figure 22: Overhead factors using various different stack implementations for Test B and Fortran/Tapenade, $N = 16$.

are different. In FM mode the progation rule is often the function itself, as is the case for **sum**, **mean**, **diff**, and **fft**, for example. In the RM the adjoint propagation has to undo and invert the operation in question in terms of the data structure, and also perform the correct mathematical computation of course. Another disadvantage is that any operation used must be supported by the derivative classes.

Example: The FM and RM propagation rules for **fft** are simply a call to **fft** itself on the derivative object. This means the derivative classes need to have the method **fft**. In the RM we have to consider, and undo, the data resizing that may occur depending on the further parameters.

The same holds for the multiplicative operators **times**, **mtimes**, **conv**, and **kron**, which are all covered by the Leibniz rule (3) and thus easily handled in FM, while the adjoint propagation is much more involved in each case, as discussed in Section 5.3.

5.1.3 Algorithmic propagation

By algorithmic propagation we refer to the idea to feed the incoming derivatives into an algorithm that propagates them internally and thus computes the output derivatives. Such an algorithm is preferably produced by automatic differentiation and as such a special case of structural propagation.

The original idea for this approach is from the work for the matrix exponential [AMH09], where a state-of-the-art method for computing that mathematical function is basically differentiated line by line in a kind of manual AD in FM.

The generic approach to use in ADiMat is thus obvious. First reverse engineer the function in MATLAB code and then differentiate with ADiMat. Thus we extend for example the case of the matrix exponential also to the RM, simply by generating the adjoint code from the same algorithm source as proposed [AMH09]. The same is done in ADiMat also, for example, for the builtins **qr** and **hess**. By also using algorithmic differentiation for the **prod** buildin in the RM

we exploit the famous Speelpenning result for the adjoint of the product reduction [GW08a].

By using this approach to handle builtins with complex functionality, i.e. many internal branches and special cases, such as **norm**, we ensure that the propagation rules in forward and reverse mode are consistent and thus decrease the software development and maintainence effort.

The respective pre-generated function codes are distributed with ADiMat and then called by the derivative propagation rules of the FM and RM in ADiMat.

The reverse engineered substitute must match the original function exactly, in particular in all fringe cases, which can be challenging. In doubt the substitute function itself should be used for consistency, by replacing the builtin in question with it explicitely in the users code. For this reason the substitution functions, like **adimat_expm**, **adimat_qr**, **adimat_hess**, **adimat_prod**, **adimat_norm**, etc. are also distributed with ADiMat. Obviously the risk for a deviation, i.e. an incomplete or inconsistent reverse engineering of the original function is minimal in a case like **prod** but larger in the cases of the much more involved matrix algorithms.

Another important application of this technique is to fill gaps in the set of builtins supported by ADiMat. For example, when a builtin like **vander** is not implemented in ADiMat one can provide ADiMat with an implementation of **vander** and let it differentiate that.

An obvious extension of that idea is obviously to directly differentiate any MATLAB codes distributed with MATLAB or GNU Octave or certain toolboxes. This can be effected by the user himself by adding the relevant directories to the search path of ADiMat.

5.2 Case study: Legendre functions

The associated Legendre functions are computed by the builtin **legendre** in MATLAB. This function is interesting because it yields a family of the Legendre functions for a single point x. Namely, for a given degree N it returns the values of the Legendre functions of all orders $M, 1 \leq M \leq N$, that means for k input values, the function returns kN outputs. The partial Jacobian is vector-blockwise diagonal and can be trivially compressed. As a result the multiplication collapses to a generalized BSX operation with the **times** operator of the compressed Jacobian function and the derivative, which can be effected with **bsxfun**.

There are also numerous special cases in the partial derivatives to be considered so the addition of this particular builtin to ADiMat can also be cited as one of the cases requiring substantial mathematical considerations [BW18].

In the reverse mode, as usual, a summation undoes the BSX operation. The implementation of the **bsxfun** operator in the array-based derivative classes yields a BSX with an additional expansion along the directional derivative dimension, as mentioned in Section 2.5, so the whole operation reduces to a single call to **bsxfun** even in vector mode, cf. also Section 3.2.

5.3 Case study: the multiplication operators

In this section we discuss four multiplicative operators, namely namely component-wise multiplication $x \odot y$, in Matlab **times**, matrix multiplication $x \cdot y$, in Matlab **mtimes**, discrete convolution $x * y$, in Matlab **conv**, and the Kronecker product $x \otimes y$, in Matlab **kron**. For each of these operators $m(x, y)$ the well known second Leibniz derivative rule holds:

$$\mathrm{d}m(x, y) = m(x, \mathrm{d}y) + m(\mathrm{d}x, y) \tag{3}$$

which spells out as

$$\mathrm{d}x * y = (\mathrm{d}x) * y + x * (\mathrm{d}y)$$

for the convolution, for example.

This means that in forward mode these operators can be easily handled by recurring to themselves in structural propagation according to the Leibniz rule (3). For example, the FM derivative code for `conv(x,y)` is `conv(g_x,y) + conv(x,g_y)`. The only thing left to be taken care of is to add the relevant methods to the derivative classes for the vector mode to work.

In the reverse mode however, there are some special considerations in each case. These are summarized as follows:

times Handle scalar expansion

mtimes Multiply adjoint by transposed matrix, handle scalar expansion

conv Convolve adjoint by reverse operand and select subset

kron Set up partial derivative and multiply by transpose

In the following subsections we discuss the four cases in more detail.

5.3.1 Component-wise multiplication

The adjoint of the **times** builtin can be covered in the same as all the component-wise operators. In principle the partial derivative is also computed component-wise and multiplied with the adjoint. The only special case to watch out for is the implicit binary scalar expansion [WBB12]. The adjoints of $z = x \odot y$ are obtained by

$$\overline{x} = \begin{cases} y \odot \overline{z} & \text{general case} \\ \sum y \odot \overline{z} & x \text{ is scalar} \end{cases}$$

$$\overline{y} = \begin{cases} x \odot \overline{z} & \text{general case} \\ \sum x \odot \overline{z} & y \text{ is scalar} \end{cases}$$

5.3.2 Matrix multiplication

The adjoint of the **mtimes** builtin w.r.t. one of the operands is given by multiplying the adjoint with the transpose of the other operand [Gil08]. The adjoints of $Z = X \cdot Y$ are obtained by

$$\overline{X} = \overline{Z} \cdot Y^T$$
$$\overline{Y} = X^T \cdot \overline{Z}$$

These rules also hold for complex matrices identically, i.e. there is no conjugate involved then, just the transpose. Binary scalar expansion may also happen when one of the operands is a scalar, in which case **mtimes** behaves identical to **times**.

5.3.3 Convolution

For the convolution operator we have the option to use algorithmic differentiation by differentiating through a devolved function that computes the convolution via Fourier transformations, in Matlab **fft**. This solution has the usual advantage that little additional work is required from an implementation point of view.

There is however also a structural propagation. This can be seen by considering the so-called circulant matrices. Given a vector $\mathbf{v} \in R^n$, the circulant matrix $C_\mathbf{v} \in R^{n \times n}$ is given by placing $\mathbf{v}$ in the first row and filling the remaining rows with circular shifts of $\mathbf{v}$:

$$C_\mathbf{v} = \begin{pmatrix} v_1 & v_2 & v_3 & \dots & v_{n-1} & v_n \\ v_n & v_1 & v_2 & \dots & v_{n-2} & v_{n-1} \\ \vdots & & & & \vdots & \\ v_3 & v_4 & v_5 & \dots & v_1 & v_2 \\ v_2 & v_3 & v_4 & \dots & v_n & v_1 \end{pmatrix}$$

Now, a discrete convolution $\mathbf{z} = \mathbf{x} * \mathbf{y} \in \mathbb{R}^{m+n-1}$ of $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^n$ can be replaced by either multiplying $\mathbf{x}$ with $C_{\hat{\mathbf{y}}}[1:m,:]$ or multiplying $\mathbf{y}$ with $C_{\hat{\mathbf{x}}}[1:n,:]$:

$$\mathbf{z} = \mathbf{x} * \mathbf{y} = \mathbf{x}^T \cdot C_{\hat{\mathbf{y}}}[1:m,:] \tag{4}$$
$$\mathbf{z} = \mathbf{x} * \mathbf{y} = \mathbf{y}^T \cdot C_{\hat{\mathbf{x}}}[1:n,:] \tag{5}$$

Here $\hat{\mathbf{y}} \in \mathbb{R}^{m+n-1}$ is obtained from $\mathbf{y}$ by appending $m-1$ zero elements and by $C_{\hat{\mathbf{y}}}[1:m,:]$ we denote the reduced circulant which consists of the first m rows of $C_{\hat{\mathbf{y}}}$. For the case $m = n$ the reduced circulant is

$$C_{\hat{\mathbf{y}}}[1:m,:] = \begin{pmatrix} y_1 & y_2 & \dots & y_{n-1} & y_n & 0 & \dots & 0 & 0 \\ 0 & y_1 & \dots & y_{n-2} & y_{n-1} & y_n & \dots & 0 & 0 \\ & \vdots & & & \vdots & & & & \\ 0 & 0 & \dots & y_1 & y_2 & v_3 & \dots & y_n & 0 \\ 0 & 0 & \dots & 0 & y_1 & v_2 & \dots & y_{n-1} & y_n \end{pmatrix},$$

but in any case $C_{\hat{\mathbf{y}}}[1:m,:]_{1,1:n} = \mathbf{y}$ and $C_{\hat{\mathbf{y}}}[1:m,:]_{m,(m-1):(m+n-1)} = \mathbf{y}$, that is, the first row is $\mathbf{y}$ left aligned, post-padded with $m-1$ zeros and the last, m-th row is $\mathbf{y}$ right aligned, pre-padded with $m-1$ zeros.

From what we know about the adjoint of matrix multiplication we can see that for the adjoint of $\mathbf{x}$ we have to multiply the adjoint of $\mathbf{z}$ by the transpose of $C_{\hat{\mathbf{y}}}[1:m,:]$. When we look at that transpose $C_{\hat{\mathbf{y}}}[1:m,:]^T$ more closely, we observe that it is a subset of the columns of the reduced circulant of the reverse of $\mathbf{y}$, conforming to $\overline{\mathbf{z}}$. We expand the transpose of $C_{\hat{\mathbf{y}}}[1:m,:]$ by $n-1$ columns to the left and to the right which can be filled so as to complete the reduced circulant of the reverse of $\mathbf{y}$ conforming to $\overline{\mathbf{z}}$.

$$\begin{aligned} & C_{\overleftarrow{\hat{\mathbf{y}}}}[1:(m+n-1),:] \\ & \qquad = \left(L | C_{\hat{\mathbf{y}}}[1:m,:]^T | R\right) \\ & = \left(\begin{array}{ccccc|ccccc|ccccc} y_n & y_{n-1} & \dots & y_3 & y_2 & y_1 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & y_n & \dots & y_4 & y_3 & y_2 & y_1 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ & \vdots & & \vdots & & & \vdots & & \vdots & & & \vdots & & \vdots & \\ 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & y_n & y_{n-1} & y_{n-2} & y_{n-1} & \dots & y_1 & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & y_n & y_{n-1} & y_{n-2} & \dots & y_2 & y_1 \end{array}\right) \end{aligned} \tag{6}$$

Note that here $\overleftarrow{\hat{\mathbf{y}}} \in \mathbb{R}^{n+m-1+n-1}$ is the vector $\overleftarrow{\mathbf{y}} \in \mathbb{R}^n$ padded with $m-1+n-1$ zeros and accordingly the reduced circulant $C_{\overleftarrow{\hat{\mathbf{y}}}}[1:(m+n-1),:]$ has $m+n-1$ rows.

From (6) we see that we can replace the multiplication of $\overline{\mathbf{z}}$ with the transposed partial $C_{\hat{\mathbf{y}}}[1:m,:]^T$ by a convolution of $\overline{\mathbf{z}}$ with the reverse $\overleftarrow{\mathbf{y}}$ of $\mathbf{y}$ and then select the subvector from n through $m+n-1$ to obtain the adjoint $\overline{\mathbf{x}}$, and likewise for $\overline{\mathbf{y}}$:

$$\overline{\mathbf{x}} = (\overline{\mathbf{z}} * \overleftarrow{\mathbf{y}})\,[n:(m+n-1)]$$
$$\overline{\mathbf{y}} = (\overleftarrow{\mathbf{x}} * \overline{\mathbf{z}})\,[m:(m+n-1)]$$

This result is apperently used already in the training of convolutional neural networks (CNN), where for the backpropagation of the gradient though a 2D convolutional layer the kernel is rotated by 180 degrees, thus inverting the order of the filter coefficients in both spatial directions [Bou06].

5.3.4 Kronecker product

The Kronecker product **kron** is not just interesting and important in itself, but also in that the **repmat** operation can be devolved to the Kronecker product.

To handle the Kronecker product in the reverse mode we set up the partial derivative, which is a sparse matrix, and perform arithmetic propagation. This can be done as follows. Consider the Kronecker product $X \otimes Y$ of two matrices $X \in \mathbb{R}^{m \times n}$ and $Y \in \mathbb{R}^{p \times q}$. The result is $Z = X \otimes Y \in R^{mp \times nq}$.

The partial $\partial Z / \partial X \in \mathbb{R}^{mnpq \times mn}$ is a sparse matrix with exactly one non-zero entry per row. To set up the partial we have to trace were the values in X end up in the result Z. To this end we create the two adjuvant matrices $\overrightarrow{X} \in \mathbb{R}^{m \times n}$ which is filled with the integers $1, 2, \ldots, mn$ in the canonical order and $\hat{Y} \in \mathbb{R}^{p \times q}$ which is filled with all ones and compute $\tilde{Z} = \overrightarrow{X} \otimes \hat{Y}$. The values $\tilde{\mathbf{z}} \in \mathbb{N}^{mnpq}$ of $\tilde{Z}$ in a vector give us the non-zero column for each row of $\partial Z / \partial X$. That is, we can obtain the sparsity pattern of the partial by using $1, 2, \ldots, nmpq$ as i-indices, $\tilde{\mathbf{z}}$ as j-indices and $1, 1, \ldots$ as the values of a sparse matrix, for example using the **sparse** builtin.

However, the non-zero entries of $\partial Z / \partial X$ are obviously not equal to one in general, but instead the values of Y, so we are not quite done yet. We know that each column of $\partial Z / \partial X$ has pq non-zeros which together hold one set of the values $\mathbf{y}$ of Y, since each element of X is multiplied with Y. To fill the values $\mathbf{y}$ into the partial correctly we sort $\tilde{\mathbf{z}}$, and apply the same permutation to the i-indices. Thus we obtain a reordered sequence of indices of the sparsity pattern were first come the pq elements that go into the first column, then pq elements that go into the second column, etc. Finally we repeat the vector $\mathbf{y}$ by the required amount of mn times and give that as the list of values to the **sparse** builtin, together with the reordered indices.

This procedure efficiently sets up the sparse partial derivative $\partial Z / \partial X$. To produce $\partial Z / \partial Y$ we proceed analogously, with the roles of X and Y swapped. Specifically we fill the adjuvant $\hat{X}$ with all ones and the adjuvant $\overrightarrow{Y}$ with $1, 2, \ldots, pq$ so we can trace where the values of Y are sent to in $Z = X \otimes Y$.

The approach presented here to efficiently set up the partial derivative of **kron** is very similar to the idea employed to handle indexed expressions and assignments, as discussed in Section 3.3.

6 Treeprocessing with XML and XSLT for AD and other structural transformations

The decision to use XML and XSLT for the adjoint code generator was motivated mainly by considering that the task at hand consists of tree processing for the most part. Thus, we resolved to using XML and XSLT because they are able to represent and transform trees directly, translating our actual problem instance to XML in the first place. AD has also been done with XSLT before for the CapeML [Pet12; Pet07]. In that instance the input problem was given in XML already, since CapeML is an XML dialect. In the adjoint code generator for ADiMat we explicitely chose XSLT as the processing language and hence create an XML export in the first place.

To name one feature of XML that is very helpful in our development but is hard to find or emulate in most other langauges, consider the XML namespaces facility, which provides a means to logically separate parts of the tree from others. This is particularly useful to attach compiler messages and analyses information to the tree, which can be done without semantically affecting the core tree when different namespaces are employed. This feature of XML is discussed in Section 6.1 together with a short introduction to XML, while and XPath and XSLT are briefly discussed in Section 6.2.

For compiler construction in general we need a means to represent trees. Here XML documents are an ideal choice, in that the hierarchy of XML elements constitutes an *ordered named tree*. This expressive level is not found in most other programming languages, where in almost all cases one has to resort to pointered structures or other indirect means to represent ordered trees. This is discussed in Section 6.3, where we shall discover for example that the language R is interestingly able to represent ordered named trees directly in its native data structure, the *named list*. This leaves the question of the most suitable programming language for the intended transformations. In Section 6.4 we discuss reasons why XSLT is in our view so ideally suited for any kind of tree processing. This is in our view due to the *rule-based processing model* and to the *literal output*

principle, while the sub-language XPath provides concise methods to evaluate selections of nodes based on filtering and path composition, from a given *context node*. In addition *XSLT pipelines* are in our view a concise method to perform complex tree transformation such as the adjoint code generation, in particular because the *recursive identity transformation* is trivially short in XSLT, yet can be overruled at any point.

Our adjoint code generator is an instance were we first have to create an XML representation of the input in order to be able to use XSLT for the processing. In the particular case of ADiMat this was not particularly dificult, since whenever a structured representation is available in memory, then all we need is a recursive function which traverses the structure and prints XML markup. The resulting markup in this case is called AST XML and presented with some examples in Section 6.5.

In other instances the need to first obtain an XML representation of the input may be a significant entry obstacle. For these scenarios, we are also actively developing solutions. This is discussed in the later Section 6.8, were we present two software packages called P2X and R2X that we have recently developed that both explicitely address this general issue. P2X can parse any structured text to XML and R2X translates between XML and named lists in R directly.

We also discuss techniques for implementing XSLT pipelines. In more general terms, XSLT pipelines are an instance of generative programming. In a scenario where XSLT stylesheets, XSLT pipeline definitions, XML schemas and our actual data are all XML, we can envision complex processing networks were various such components are dynamically generated, transformed and updated, all with XSLT, and all this even in-memory when we use a glue language with XSLT support, such as MATLAB, JavaScript, or R. A sensible structure is essential for such scenarios. In ADiMat we basically use plain XML lists of XSLT stylesheet URLs as pipeline definition, but with a small preprocessing step that statically unrolls several closely related pipelines each from tree-formed definitions. These issues are discussed in more detail in Section 6.9.

We also wish to discuss validation in XML briefly, although it is not used in our adjoint code generator and its development. First of all, we think the crucial advantage of XML over the previous attempts of markup languages is of course that it achieves universal readability by using the concept of named parenthesisation to define well-formedness [Cho02]. This means that a simple generic parsing algorithm exists that can read XML document even when all the element and attribute names are unknown. This means that a XML document can be parsed withour prior knowledge and thus provides a generic method to represent named ordered trees. This strategy is mirrored by the JSON notation on a lower syntactical level. When on top of that, additional constraints are placed on the document, that is, on the names of elements and attributes, this is called in XML terms *validation* against a certain grammar, usually called or *XML schema* or *document type*.

Within a single XSLT pipeline, validation in every single step is most probably superfluous and even futile. Instead, a concept that is very helpful in the controlling of a desired tree structure has in our experience turned out to be the use of normalization operations, that take the form of idempotent filter steps. We would like to discuss the relation of validation to these normalization operations. Basically, for every post condition that such a filter establishes, we can allow our tree to deviate from the standard form, for we can very flexibly re-establish said postcondition whenever that should be required, by applying the filter. To specify a validation procedure, a tree in AST XML could be declared valid if it is valid after performing a list of normalizations, including the pruning of subtrees from other namespaces. Thus, while we do not practically use such a validation procedure, these reasonings reveal the importance of having an arsenal of suitable renormalization operations, as discussed in Section 6.11.

6.1 XML terms and definitions

XML 1.0 is a markup language derived from SGML [Bra+00]. Compared to SGML, the main change decided for in XML was to make XML redable without a knowledge of the document structure, the so called *document type*. The only requirement is that XML documents are *well-formed*, that is, the so called *XML markup* is a balanced named parenthesisations of starting

(`<a>`) and closing tags (`</a>`). For this form of markup there are efficient parsing algorithms. A *document type*, which is not required in XML, basically defines which tag names we can expect to occur, that is, it is basically a grammar for XML documents. The process of *validation* against document types is available in XML as an optional feature [Wik19b].

After parsing, the parser provides an in-memory representation of the XML *document*, which is an ordered tree of XML *nodes*. The *root* node at the top has among its children exactly one XML *element* node, the *document element*. Each XML element has an ordered list of zero or more arbitrary nodes as *children* or *child nodes*. Nodes other than elements are `text()`, `comment()` and `processing-instruction()`. These nodes cannot have children, and hence occur only as leaves in the tree. For our purposes we only need `text()` nodes or *text nodes*.

An XML element has in addition a set of *attributes*, which is a named map of string values. Attributes are considered as nodes, but are usually considered to be set apart of the tree, so e.g. a *subtree*, a list of elements and its descendants, would not normally be meant to include the attributes of the elements. However, any element is always thought of as carrying its set of attributes. This is modelled in XPath by the special axis `attribute::`, with the short-cut `@`, such as in `adm:binary/@op`.

The tree of XML nodes is accessible by a common programming API, the so called *Document Object Model* (DOM) API [W3C04; Kes20]. This provides functions to navigate the tree step by step, and to create, insert and remove individual nodes, together with a set of utility functions for basic searching of nodes.

XML elements and attributes have a *node name*. It can include most unicode characters, including `-` and `:`, but must not start with `xml`, a `-`, a `.` or a digit. The colon `:` in an XML name is special in that it distinguishes a *name prefix*. A name prefix must be bound to an *XML namespace*, which is in turn identified with its so called *namespace URI* [W3C09]. This binding must happen at the element that introduces the prefix via a special XML namespace declaration, which uses special attributes starting with `xmlns`, as in

```
<x:br xmlns:x="http://www.w3.org/1999/xhtml"/>
```

This element shown has the name `x:br`, which has the name prefix `x` and the *local name* `br`. The name is in the namespace `http://www.w3.org/1999/xhtml`. This element is identical to

```
<br xmlns="http://www.w3.org/1999/xhtml"/>
```

which uses the `xmlns` attribute to bind the default namespace to the same namespace URI. In XPath or XSLT we have to refer to the same namespace, but we can use any prefix to bind it to, for example set the attribute `xmlns:xhtml` in an `xsl:stylesheet`

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xhtml="http://www.w3.org/1999/xhtml"
  version="1.0">
```

When we then use `xhtml:br` in the XPath expressions within that stylesheet either of the shown elements would be matched by that.

In the DOM API, the fact that XML namespaces were added later in the specification process of XML is reflected by the fact that many functions will not consider namespaces, and have siblings with the suffix `NS`, which do, for example `getElementsByTagName` and `getElementsByTagNameNS`. In XSLT and XPath however, it is not possible to ignore the namespace bindings, that is, any name in XPath matches only elements or attributes of the same name and namespace. One must however use a name prefix, that is, a name without name prefix in XPath always refers to a name from the default namespace. The same it true for attributes, any attribute without a name prefix are in the default namespace.

We call the tree of elements of an XML document, discarding any other nodes, the *element tree*. We call the XML namespace of the document element the *core namespace*. We call the XML element tree made up of all connected elements from the core namespace the *core tree* of the document. Any remaining substrees will consequently then have an element from some other namespace at their root and are attached as leaves to the core tree.

6.1.1 XML documents with the leaf text property

We think is worth defining a subset of XML documents here, namely those that are of the *leaf-text* form. While text nodes are always leaves, by this definition we refer to documents where all text nodes are attached to elements that are leaves in the element tree. This means that text nodes never occur as siblings to elements, and that they have a element which contains them as their only child. The fact that any text node that we attribute some meaning to is wrapped in its own element means that we can ignore any other text nodes, and this in turn means that we are free to introduce such text nodes, in particular text nodes containing only white space for pretty printing with indenting.

In our experience, this is a very broad definition. Almost any XML that is devised for technical purposes will use a leaf-text form. An example for documents that are not leaf-text in general is HTML, as shown in the example

```
<p>This must be <i>emphasized</i>.</p>
```

There is of course still a simple procedure to make such documents leaf-text, continuing the example that could result in

```
<p><span>This must be </span><i>emphasized</i><span>.</span></p>
```

However, it is still not without consequences to insert indentation whitespace into this leaf-text form of the HTML. For example, any amount of white space between `</i>` and `<span>` would result in a single white space character between the word `emphasized` and the full stop character in HTML rendering.

Pretty-printing an XML document with indenting, each tag in its own line, with indenting according to the depth in the document, is a very simple and effective way to visually represent its structure. However, XSLT processors will often refrain from inserting the additional whitespace that is required for such indented pretty-printing, since they are not generally free to do so, to insert additional text nodes into the output tree. Even with setting the XSLT attribute `xsl:output/@indent` to 'yes', when new elements are generated in the output tree, these are often without any whitespace, several tags on a single line, and hence difficult to read.

Hence, it is helpful when we work with XML documents that are of the leaf-text form, since these can be indented without problems. For pretty-printing and indenting leaf-text XML markup we can provide a very simple procedure:

1. Textually replace in the XML markup text all occurences of `><` with `>\n<`, that is, insert a newline character whereever two XML tags – opening or closing – are adjacent.

2. Pretty-print the resulting XML with indenting by some other means, such as the command `indent-region` in the Emacs editor [Sta15] or using the command `tidy -asxml` [Rag20] in the Unix console

This simple procedure bound to a key macro has proven invaluable and even indispensable when debugging intermediate states of XML pipelines. We have to assume that in the original markup any empty element was written as `<x/>` and not as `<x></x>`, of course, or else our procedure would insert a text node where there is none to x. However, we usually can assume the first form, and hence our procedure will never add whitespace to any leaf element. Where ever there is any text node, there cannot be the sequence `><`.

As an example consider the XML of the AST XML when it is written as compactly as possible, as shown in Listing 12.

```
<function xmlns="http://www.sc.rwth-aachen.de/ns/adimat"><outvars><var><id
    id="0">z</id></var></outvars><id
    id="3">ftimes2</id><param-list><var><id id="1">x</id></var><var><id
    id="2">y</id></var></param-list><statement-list><binary op="="><var><id
```

```
id="0">z</id></var><binary op="*"><var><id id="1">x</id></var><var><id
id="2">y</id></var></binary></binary></statement-list></function>
```

Listing 12: An XML document without any indenting whitespace. The document structure is difficult to discern

Remember that in some XML scenarios it would be wrong to alter this document by indenting it. Indenting adds text nodes, or adds whitespace to text nodes, which may alter the semantics of the XML document. When we know however, that the document is of the leaf text form, then we may safely apply our procedure. After the first step, the document markup is now spread across several lines, one tag or one leaf element per line, as shown in Listing 13. When we run some XML pretty printing program on the result, like the shell filter command `tidy -q -xml -indent` or `xmlstarlet fo` we obtain markup where each tag is preceded by an amount of whitespace according to its depth in the tree. In such a form the structure of even relatively large XML documents is clearly visible, as shown in Listing 14.

```
<function xmlns="http://www.sc.rwth-aachen.de/ns/adimat">
<outvars>
<var>
<id id="0">z</id>
</var>
</outvars>
<id id="3">ftimes2</id>
<param-list>
<var>
<id id="1">x</id>
</var>
<var>
<id id="2">y</id>
</var>
</param-list>
<statement-list>
<binary op="=">
<var>
<id id="0">z</id>
</var>
<binary op="*">
<var>
<id id="1">x</id>
</var>
<var>
<id id="2">y</id>
</var>
</binary>
</binary>
</statement-list>
</function>
```

Listing 13: An XML document without any indenting whitespace, but every tag or leaf element printed on its own line

```
<function xmlns="http://www.sc.rwth-aachen.de/ns/adimat">
 <outvars>
  <var>
   <id id="0">z</id>
  </var>
 </outvars>
 <id id="3">ftimes2</id>
 <param-list>
```

```
  <var>
   <id id="1">x</id>
  </var>
  <var>
   <id id="2">y</id>
  </var>
 </param-list>
 <statement-list>
  <binary op="=">
   <var>
    <id id="0">z</id>
   </var>
   <binary op="*">
    <var>
     <id id="1">x</id>
    </var>
    <var>
     <id id="2">y</id>
    </var>
   </binary>
  </binary>
 </statement-list>
</function>
```

Listing 14: A leaf-text XML document pretty-printed with typical depth-based indenting whitespace. The document structure is immediately appearent

6.2 XPath and XSLT terms and definitions

The query language XPath is use to select values from an XML document, value types being `nodeset`, `string`, `number` and `boolean`. The core operator is the slash '/' which selectes nodes given by the RHS expression for each node in the nodeset that is the LHS expression. Nodesets are always in document order. Nodesets can be converted to a string value if necessary. This means to take the first node in the node set and build its string value. For an element this means to concatenate all text nodes found in the element subtree. Strings and thus also nodesets can also be converted to numbers as necessary as per the IEEE-774 standard. XPath expressions that select a nodeset and do not start with a '' *are called /relative* and are evaluated with respect to a given *context node*.

In XPath we must use a namespace prefix to access whatever element by name, which is in a namespace, and can use a prefixed wildcard to select all elements from some namespace, so for example `xhtml:br` matches a HTML5 `br` element when `xhtml` is bound to the HTML5 namespace, while `br` matches only `br` elements from the default namespace. The wildcard `xhtml:*` matches any element from the HTML5 namespace, while the wildcard `*` matches any element from any namespace.

XSLT is a rule based, turing complete processing language for XML documents, that allows general purpose transformations of XML documents. The first version XSLT 1.0 and the slightly enhanced XSLT 1.1 is probably still the dominant version of XSLT in use. The later revisions XSLT 2.0 and XSLT 3.0 do not appear to have gained widespread use, the Saxon processor being the most prominent processor that implements the later XSLT 2.0 and XSLT 3.0 standards [Kay20]. This lack of practical relevance is in our view due to XSLT 1.0 being a very well designed language which has a very balanced set of functionality that due to its flexibility leaves little to be desired in terms of features, and yet is defined by a relatively concise and readble standard. What is often considered the most important added capability of XSLT 2.0 over XSLT 1.0, namely the reprocessing of output trees, is obviously also possible by chaining multiple XSLT 1.0 stylesheets in a pipeline, which arguably also leads to more readable and flexible XSLT code. Another very interesting and obviously attractive feature of XSLT 2.0 is the posibility to accept also other

structured text other than XML, such as CVS, or relational data bases as input. As to structure text, our software package P2X is also able to translate structured text to XML, cf. Section 6.8.2.

Current development of XSLT 1.0 and XSLT 1.1 processors is largely dormant due to several high quality and feature complete implementations being available from early on, with Apache Xalan [The99; The11; Leu04], libxslt [Vei03] and Saxon [Kay01b; Kay01a] being the most widely used. Some work is being done with regards to improving the performance, for example using early exit when an expression like `following::*[1]` is evaluated [Wik20h]. In our view, further room for improvement might by found in the area of parallel computing. In more complex scenarios parallelisation may also be applied on the higher level of processing networks or transformation servers, but certain large scale XML documents and their transformation may still benefit from parallelisation internal to the XSLT processor. According developments have been taking place in the Saxon project [Kay15]. Using OpenMP 3.0 tasks to appears to be an attractive option to implement parallel processing in the XSLT processors written in C and C++, in our view [Ope08; Pas09; Ayg+09]. Another angle is to reduce the memory requirements by avoiding to load documents entirely and process them with streaming techniques [Kay10]. Some recent research is also done on benchmarking XSLT processors [MHM14].

XSLT processes one XML document against a given set of rules defined by one XSLT stylesheet, producing one XML or plain text document as output. A stylesheet is itself an XML document which basically contains a list of `xsl:template` elements. Stylesheets can be structured into multiple files by using `xsl:import` and `xsl:include`. Templates have a *@name* or a *@match* attribute or both.

Templates are called either by `xsl:apply-templates` according to their *@match* attribute, or by `xsl:call-template` according to their name. When a template is called, any child nodes it contains are emitted into the *output tree*, while child nodes from the `xsl` namespace are processed.

A template is called either by `xsl:apply-templates` by finding matching templates for the nodeset defined by the *@select* expression or by `xsl:call-templates` with the `@name` attribute. The *@select* expression defaults to `*|text()`, so child nodes are selected. It is also possible to select other nodes, and to travese the tree in other directions, of course. When a template is called with `xsl:apply-templates` the *context node* is moved to each selected node in turn, and the template matching best is called. The templates have some simple precedence levels which controls their selection by `xsl:apply-templates`, which is imputed according to their match expressions. Basically XPath expressions with a wildcard `*` and with the nodeset union operator `|` get a malus while XPath expressions with a bracked filter expression are given a bonus on the basic precedence level. From several matching templates of equal precedence, the last one in the stylesheet is called. So, when several templates with a `@match` attribute of the form `name[xyz]` are used, where `name` is identical but the filter expressions are different, then the programmer must see to that the more specific filter expressions are place after the more general ones, for these all have the same precedence. For example:

```
<xsl:template match="myelem[@attr1>0]">
  Case 1: <xsl:value-of select="."/>
</xsl:template>
<xsl:template match="myelem[@attr1>0 and @attr2>0]">
  Case 1+2: <xsl:value-of select="."/>
</xsl:template>
<xsl:template match="myelem[@attr1>0 and @attr2>0 and @attr3>0]">
  Case 1+2+3: <xsl:value-of select="."/>
</xsl:template>
```

In this example the increasing specificity is ease to see, but depending on the actual filter expressions, it may require some thought.

Both `xsl:template` and `xsl:apply-templates` can also have a *@mode* attribute, which is simply a name. A `xsl:apply-templates` with some mode only matches templates with the same mode. Thus, for each mode a separate set of templates is created, empty except for default templates. The default set of templates for any mode used, including the default mode, is to recurse through the element tree and to emit text nodes:

```
<xsl:template mode="mymode" match="text()">
  <xsl:value-of select="."/>
</xsl:template>
<xsl:template mode="mymode" match="*">
  <xsl:apply-templates/>
</xsl:template>
```

For this reason a frequent idiom to introduce a new mode is to disallow this text output by overriding the default template, or else text nodes from the tree will be emitted as soon as some element is not covered by some match expression of the template set.

```
<xsl:template mode="mymode" match="text()"/>
```

Although XSLT basically processes one XML document and produces one XML document as output, any number of further documents are accessible within XSLT via the XPath `document()` function. The resulting side loaded documents can be bound to variables, they can be selected from with XPath and they can processed just as the input document with `xsl:apply-templates`.

XSLT 1.0 however distinguishes between input documents and output nodes that it has generated already. The latter can also be captured in `xsl:variable` or passed as `xsl:param`, for repeated output with `xsl:copy-of` for example, but they cannot be queried with XPath or processed with `xsl:apply-templates`. This is a frequent stumbling block for beginners, who are used to reprocess intermediate results in other languages to achieve some aim. XSLT 2.0 provides reprocessing of output trees. With XSLT 1.1 it is possible to generate multiple output documents by using the added `xsl:document` element, which in some situations is a very useful feature. This together with the `document()` function also provides a loophole to reprocess output trees in XSLT 1.1.

However, the same effect as reprocessing output trees is obviously available by chaining XSLT stylesheetes in a pipeline for consecutive transformations of a document. This requires an external glue language to control the overall XML transformation, but it is a simple task to create a corresponding framework and the step to even more complex XML processing networks is also close, as we describe in Section 6.8.

6.2.1 The literal output principle of XSLT

Another very considerable advantage of XSLT over other languages is in our view and experience the concept of literal output. Very simply, any content that is placed inside an `xsl:template` element is output literally whenever the templates is called. Except for the three characters (`<`, `>`, and `&`) that must be escaped in XML, anything else is just typed in literally.

Regarding text output we have the very sensible rule in XSLT that text nodes in the stylesheet with only whitespace produce no output, which means in particular that the XSLT itself can the freely indented. Text in `xsl:text` elements is however output without modification.

This literal output concept includes specifically any XML elements, including their namespaces. Namespace prefixes in a stylesheet are also defined as per the usual XML rules. During the template processing any XML inside the `xsl:template` elements are also output literally. Any XML tags must obviously be well-formed as otherwise the XSL stylesheet itself would not be well-formed and thus not XML and unreadable. Hence it is almost impossible to produce output with XSLT that is not well-formed XML.

XML elements that are from the XSLT namespace are obviously treated specially in the template processing, namely, they are processed as per their respective semantics. For example `xsl:apply-templates` matches the current set of templates to the selected nodes and that output nodeset is inserted at the point of the `xsl:apply-templates` element.

XSLT can produce either XML or plain text output. In an XSLT processing chain there is usually at most one XSLT that produces non-XML text output, and this is the last. One should briefly mention that certain characters are not part of XML and hence binary data cannot be generated with XSLT directly, so a binary filter program would minimally be required for postprocessing, if binary data shall be generated.

In our experience any kind of text output can be produced with XSLT with relatively little effort. In our view, this is due to the literal output principle. This invertes the semantics layers, if we may call it that way, with respect to all other programming languages. Usually what we type in a certain programming language as code is statements and definitions in that language. With any language we can of course instruct the interpreter or language runtime to output text. However, this normally means that we at the least put our output into a string literal in that language. Consider the famous "Hello World" program. This act already means that we have to escape quote characters in any text, for example `"Hello \"World\""` in C. Then we have to call a suitable function to effect the printing of the output, such as **fprintf**. Of course, a single static string literal is not very flexible, so we usually have to compose the output string within the language from many pieces using many function calls, including those which specifically deal with text formatting, such as **sprintf**.

In XSLT however, this logic is inverted. The primary plane of text content, so to say, is reserved for the user level output. There are no explicit print or write statements that need to be called, output is just what is inside the `xsl:template` elements, be it text or XML elements. Only XML elements from the XSL namespace are processed according to the rules of the language.

6.3 The expressive level of XML compared to other data structures

Table 2: The expressive level of languages and their most common data structures

Language	Data structure	Repeated names	Order preserved	Namespaces	Nesting
C/C++	Array	–	Yes	–	Yes
C++	STD Map	No	No	No	Yes
Python	Dict	No	No	No	Yes
JavaScript	Assoc. Array	No	No	No	Yes
C++	STD Multi-Map	Yes	No	No	Yes
MATLAB	Struct	No	Yes	No	Yes
R	List	Yes	Yes	No	Yes
XML	Elements	Yes	Yes	Yes	Yes
	Attributes	No	No	Yes	No

In Table 2 we detail the expressive level of some common programming languages and their most common data structures. For this list we do not consider composite user level data types, such as classes or certain frameworks, but just the data types that are native to the languages. For example, the staple data structure of many interpreted languages is probably the dictionary or associative array. For these we exemplarily list Python and JavaScript in the table, but other examples are PHP, Perl, the Unix shell scripts, and many others. These dictionaries have in common that the order of elements is not preserved and moreover names may not occur repeatedly, there can be just one entry for each name. With these characteristics they are on the same level with XML attributes, although the latter cannot be nested, of course. The MATLAB language features the **struct** data type, which differs slightly, in that the order of elements is preserved.

In the case of the C language, the apex data structure is probably the plain array as defined for example by `int x[100];`. A C array preserves order but has no names at all, so we list it at the top of the table. A dictionary data type is available in C++ with the `std::map` container from the Standard Template Library, which can be used with strings as map keys. Repeated names or keys are possible with the `std::multimap` container.

A notable exception in this list is the R language. The difference is subtle, but crucial. The R language features a list type where the elements can be assigned names, which may occur repeatedly. This entails a one-to-one equivalence between named lists in R and XML documents, or more precisely between named lists in R and leaf text XML documents.

When we now consider the problem of representing an ordered tree in the different languages, as detailed in Table 3, we must use some level of indirection when that is not possible directly. For example in C or C++ we must devise some dynamic pointered data structure that represents our

Table 3: How an ordered name tree could be represented in different languages, not considering namespaces and attributes

Language	How to represent a named ordered tree
C++	Pointered structure
Python, JavaScript	Dictionary with tuples *name* and list *children*
R	Named list
XML	XML document

tree. When a dictionary data type is available we could create a nested structure of dictionaries with two entries `name` and `children`, where the first always holds a string and the second a list. This is what our P2X parser does to output its tree directly to MATLAB or JSON [Wil13b], as described in Section 6.8.2. With this technique we can even represent text nodes in between elements, not just leaf-text trees. We could also use in either language the DOM API, but this obviously also a level of indirection.

In R we are in a more comfortable position, a simple generic tranformation can be devised between named lists and leaf-text XML documents. A recursive named list structure in R corresponds to the element tree of an XML document. We map the non-list leaf elements to text nodes in XML and can thus identify leaf-text XML documents with named lists in R. On top of this, R has an attribute system that is entirely equivalent to XML attributes. This striking structural equivalence is implemented by our new tool R2X to directly translate between R and XML data structures without the need to go through the DOM API. This is described in more detail the later Subsection 6.8.3.

6.4 The expressive level of XSLT compared to other languages

Table 4: How an ordered tree structure can be accessed or processed in different languages

Language	Access a named ordered tree	Process named ordered tree
C++	Recursion	Recursion
Python, JavaScript	Recursion	Recursion
R	Recursion	Recursion
XML	XPath	XSLT

Once we have a representation of a named ordered tree loaded in memory in our chosen representation, we most obviously work with the tree and transform it as desired, on in simpler settings, extract information from it. With indirect means of accessing our tree, we are bound to use sequences of simple operations that have to be arranged in software structures that will probably have several access and convenience layers for any non-trivial use cases. In C++ for example, suppose we are given the pointer to an assignment node and tasked to find all active variable occurences on the RHS. We will have to define some recursive little algorithm that traverses our subtree to find the variables in question. For such common little search tasks we will probably devise an software abstraction to generically traverse our tree and apply some predicate or operation to each node. However, consider our example further. An active occurence is never inside an index expression, so we need more tools, such as filtering, or different recursive traversals, etc. Then we finally express our actual intention as a programmer through these layers of abstraction. This situation is denoted in Table 4 with the entry Recursion. It is no different in Python or JavaScript, although both have functional programming means to process lists, such as the `map` function.

Even in R, while we can seamlessly represent named ordered trees as named lists, working with these structures is not as seamless. For example, the simple access operators in R will select just the first element of a given name. And even though we have in addition to `lapply`, which is the map function for lists in R, also `rapply` for nested lists, we will still need recursive programming

to transform trees arbitrarily. For example, the function `rapply` can still only produce one output for each leaf input item and also cannot modify the list hierarchy.

When working with XML, we use XPath and XSLT to access and manipulate tree structures directly. XPath is used for selection of nodes [CD+99] and as such also a subset of XSLT. XSLT is a rule based declarative language for XML processing [Cla+99]. Its core features are, in our view, the rule based processing and the concept of literal output. This allows us to specify processing steps which perform the identical transformation very concisely. Then, basically we just have to add those rules needed to effect the intended change.

By arranging XSLT processing steps in a pipeline, we achieve a powerful method for tree transformation in general and for our adjoint code generation in particular.

In our view, XML is only ever easy to handle when using XSLT. For example, the easiest way to implement a way for XML to be available in JavaScript, is in our view an XSLT stylesheet that emits JSON. In other interpreted languages one may also generate a literal expression of the desired structure with XSLT, to be evaluated with the uniquitous **eval** function. When some information is to be extracted from XML, one should use XSLT to compute the information and then, possibly in a second XSLT, emit the answer in JSON format. The only alternative is to step node by node through the document with the DOM API, which should be done only when necessary. The failure to realize this by comparing the expressive level of XML and XSLT and other languages has probably left to the demise of XML in the web applications, and its replacement by JSON. In our view, this is however unjustified, and the future of the web lies in using JavaScript as a glue language to orchestrate XSLT transformations that are applied to the so called DOM tree, which in this context refers to the DOM document of the web page that is shown in the browser, or to other XML input data, to produce HTML subtrees to be inserted into the DOM tree. Instead, wave after wave of JavaScript libraries are written to manage the requried tree transformations. When we see it this way, sooner or later we are sure to see a resurgence of XML and in particular XSLT in the area of web applications. All the foundations are layed, in particular XSLT processing is available via JavaScript APIs in all major browsers. Our XC project, presented briefly in the later section 6.10 is an example for such an approach.

In the following Section 6.5 we also describe in more detail how we use XML for AST representation. In the Sections 6.6 and 6.7 we describe in more detail how we use XSLT and XSLT pipelines for AST transformation. As an outlook we present generic techniques for using XML and XSLT pipelines for problem solveing, in particular two simple tools facilitating the entry into the XML world from other structured data representations, namely P2X and R2X, which we already mentioned, in Section 6.8.

6.5 AST representation in XML

In our adjoint code generator, an XML document represents the abstract syntax tree (AST), in a form that corresponds to the tree structure commonly used in most compilers. This XML document uses a particular format that we call AST XML. The XML is printed from the existing ADiMat software by a recursive printing function that emits the XML markup. Some other information is also provided by already existing analyses in the ADiMat forward mode processor, in particular the activity analysis, and included in the XML output.

In order to benefit as much as possible from the XSLT processing we use a tree-based form to represent the parsed code and also the transformed code, as these can readily be represented in an XML document. Fortunately, this is what ADiMat already does in the forward mode code generator.

We would not particularly recommend working with the also commonly used graph formats for compiler construction like basic block graph, variable dependency graph, function call graph, etc. in an XML context. While it is obviously possible to encode a graph format in XML, for example the often used form of node and adjecency lists, there is not really a benefit to be gained from prepresenting the node and adjacency lists in XML. An alternative to represent graphs in XML would be to use spanning trees, but this also is not ideal. More generally speaking, graphs are a more general data structure than trees, and for this reason one will be working with indirections,

no matter whether one works with an procedural language on the node and adjacency lists of a graph or with a tree processing language like XSLT on node and adjacency lists or spanning trees in XML format. The ideal tools for working with graphs would be a graph data format and a graph processing language. These do exist [PF71; Mal+10; Hon+14; SW13], but do not appear to have gained much maturity or popularity as of yet. Hence, we resolved to use tree structures for our task as much as possible. This is different to other AD projects, were the source code is often represented by a graph of basic blocks, and expressions are also often represented by graphs, such as OpenAD [Utk+08b], which uses the XAIF XML format [HNN02], also for adjoint code generation [Nau+04; Nau+06]. So, while it is understandable to prefer graphs over trees as the fundamental data structure when constructing a compiler framework, given that graphs are a higher-level data structure than trees, we think it is also reasonable to settle for the lower-level choice of trees given that we have means to represent and process trees directly.

XML namespaces are used to separate core tree and annotations: usually algorithms on the core tree when expressed in XSLT will be undisturbed by whatever additional leaves from other namespaces are in the tree. When tree restructuring is done, such additional leaves may be lost quite easily, but this can also be avoided with some consideration. Typical useful annotations are results of code analyses, more generally any precomputed information, but also compiler notes, warnings and error messages, which can simply be inserted into the tree whereever they arise.

Abstractions are XML elements that are normally not part of the AST, but from the core namespace and part of the core tree, may represent entire sub trees of expressions or code. They are often created on an ad-hoc basis, simplifying certain processing steps. Again, XSLT algorithms working on different levels of the tree can easily avoid having to know about these, but nevertheless preserve them. Certain postprocessing optimizations which are concerned with these language elements may become more concise since they can reference them by their name. In the end, either the pretty printer will have to be enabled to handle these elements or we define a devolving postprocessing step that expands them back to common language elements, or do both. Examples for abstractions that we employ are for the frequent idioms of adjoint code, like push and pop instructions, adjoint increments, adjoint initializations, adjoint reductions, etc.

The second, related set of restrictions is concerned with the namespaces of the element names. There are several namespaces that we use in AST XML. One of these is the so called core namespace, while we call the others auxilliary namespaces. All the information required to reconstruct the program code is represented by the core tree, while elements from the auxilliary namespaces may be attached to this tree at any level, but they can easily be ignored, or more precisely, almost any given XSLT algorithm that operates on the core tree will automatically ignore elements from other namespaces, which is a again a very simple yet comprehensive exercise in defensive programming. This is due to the rules for name matching in XPath, which we discussed already. Only using the plain wildcard $*$ this will match also other namespaces. So we have the overall convenient behaviour that the recursive identity transform rules copy all AST tree nodes including elements from other namespaces, but as soon as we start refering to specific nodes, this will only match elements from the desired namespace.

We identify two kinds of restrictions or structural invariants on the XML that we work that with for this compiler construction application. Both are structurally similar. The first regards text nodes, namely, we define our tree to be in the leaf text form. In AST XML the set of elements that contain text nodes is quite small, these are namely *var*, *id* and *literal*. The first two are use for any kind of identifiers and the second for both number and string literals.

Consequently, any existing transformation will almost certainly work unchanged irrespective of what auxilliary elements are attached to the tree. Only if a certain transformation requires the attached information it can refer to it by knowing name and its namespace. Thus any kind of side information can be attached to the tree at any stage to be used at some later stage in the pipeline.

6.5.1 XML AST elements and namespaces

The following elements are used in the upper level of the AST XML:

adimat this is the top level element

master-tree this element is child of **adimat** and contains the actual AST

The AST inside the *master-tree* may be either a list of statements or a list of functions, or both. On the AST level we use the following elements:

function-list list of *function* elements

function defines a function, with fixed list of four or five children: *id*, *param-list*, *outvars*, *statement-list*, possibly followed by *function-list*

statement-list list of expressions, assignments, or control flow elements

for control flow element, with fixed list of three children: *id*, an expression, *statement-list*,

while control flow element, with two children: an expression and *statement-list*

break control flow element

continue control flow element

if control flow element, with fixed list of two or three children: an expression, *statement-list*, and possibly *else* or *elseif*

elseif control flow element, with fixed list of two or three children: an expression, *statement-list*, and possibly *else* or *elseif*

else control flow element, with one *statement-list*

switch control flow element, with list of *case*, possibly followed by *otherwise*

case child of *switch* with an expression and *statement-list*

otherwise child of *switch*, with one *statement-list*

From the expression level downwards we use the following elements in the main namespace:

binary binary operator, with two children, attribute *@op* contains literal operator, such as `=` for assignments, `+` for **plus**, `*` for **mtimes**, etc.

unary operator, with one child expression, attribute *@op* contains literal operator, such as, `+` for **uplus**, `~` for **not**, etc., but *@op* may also be `()`, `[]`, or `{}` to represent a parentheses expression

sum represents addition and subtraction with one or more children *op*

op child of *sum* with attribute *@sign* and a single expression as child

call function call, with *id*, *param-list*

param-list function call arguments, with list of expressions

row-list for matrix literals, as child of `unary` with *@op* set to `[]` for a literal matrix or set to `{}` for a literal cell array, with list of *item-list* as children

item-list child of *row-list*, with list of expressions

literal for number or string literals, with single text node

Then we have *storage expressions* that are in turn a subset of value expressions, and thus a further level of the AST tree. They may occur on the LHS of an assignment and they are made up of

array array access, or *indexed expression*, with two or more children: a storage expression, *param-list*, followed by zero or more *cell-index*

cell-index index in cell array access, with single child *param-list*

struct-ref for a structure reference, with two children, a storage expression and *id*

dyn-struct-ref like *struct-ref* but second child may be a general expression

var variable, either *id* or a single text node

As a utility element we have

id identifier name of a function, for loop variable, function name, or variable, with single text node

param-list As child of *function* represents function arguments and has a list of zero or more *var* elements, as child of *call* or *array* represents function call or index arguments and has list of zero or more expressions

outvars Output variables of a *function*, list of zero or more *var* elements

A few notes regarding the structural arrangement of these elements and how they are used to represent the AST of the MATLAB code are in order.

The *function-list* under *master-tree* may contain one or more *function* elements. These are the functions listed in a MATLAB source file, the first being the one that can be called by the name stem of the source file while the others are so called sub functions that can be called only be called by the first. When multiple MATLAB source files are processed by ADiMat, these are also listed under this *function-list*. The *function-list* under *function* may contain zero or more nested functions.

Most of the elements are provided with attributes *@line* and *@column* and the *function* element in addition has the *@file* attribute for error messages with a reference to the exact place in the source code. In the XSLT pipeline, when an error message is composed we use the closest of each of these attributes found from the location in question searching upwards in the AST.

The representation of branches with nested *if*, *else*, and *elseif* elements is certainly complex and a prime candidate to be canonicalized. We would advertise the form that is also used in XSLT: an *if* element for single branch conditionals and *choose*, with *when* and possibly one *otherwise* children for multi-branch conditionals. However, the reason for using this particular structure is that it comes out of the ADiMat parser in this form and the handling of the control flow in `reverse-ad.xsl` was done in the very early stages of the development, when we did not use the technique of pre- and postprocessing as much as we would do in hindsight. There are basically just two instances were we need to explicitely handle these structures in detail: One is the adjoint code generator in `reverse-ad.xsl` and the other is the source code generator `to-source.xsl`. In each we need to spend a couple of additional template modes to deal with this branch representation that would not be needed if we used a simpler form. This shows among others, one thing: the XSLT code that was developed around this rather complex branch representation is quite stable, so once it was done, there never arose a need to review the case.

Then we are somewhat liberal with the question of what `var` actually looks like, whether it contains an *id* element with the variable identifier or the name as a text node directly. This is tolerable since *id* is itself a leaf element. This continues in the processing pipeline were temporary variables are created by wrapping the *id* of *var* elements in *tmp* elements, and similarly adjoint variables are created by wrapping the *id* of *var* elements in *a* elements, leaving both to be expanded to the usual prefixes at a later stage.

As to the binary operators, we also give ourselves some leeway in that a *binary* element may have just one child element, i.e. operand, or more than two operands. More precisely, the parser in ADiMat merges, for example, multiple adjacent multiplication operators into a single node, with

more than two operands. In the preprocessing stage of our pipeline we use a normaliaztion filter to transform this to a normal form where each *binary* element has exactly two children, which simplifies the further processing. The adjoint code generator may then again produce non-regular *binary* elements, but there is no need for cleaning these up explicitely, because the source code generator is also able to handle them directly, which is a straight-forward exercise in defensive programming that costs no additional effort. This is discussed in more detail with examples in the later Section 6.6.

The representation of addition and subtraction via the *sum* element with one or more *op* children with a *@sign* attribute is in our view cumbersome and we would in hindsight rather represent these AST nodes via a *binary* element. Similarly to the branching with *if* the relevant XSLT code handling theses operations was completetd fairly early in the development and there was no later need to change it. The adjoint code generator does not emit *sum* and *op* however, but uses *binary* with *@op* set to `+` or `-` instead.

Another item to discuss are the assignment operators. They are represented in AST XML in a *binary* element with *@op* set to `=`. In MATLAB an assignment, may only occur at the interface between the control flow tree and the expression tree. This means they occur always on the top level of an expression tree, as children of the *statement-list* elements. The assignment operator cannot be used in a nested expression or in condition expressions in MATLAB. While in the GNU Octave dialect it can, just as in may other languages such like C or R, this is not supported by ADiMat.

On the other hand, the left hand side of the assignement may be made up of a bracketed list of output arguments. These are similar to the matrix literal expressions, but they specify the storage expressions where the output arguments are to be stored. This syntax allows to return multiple output arguments from a function. This means, when *unary* with *@op* set to `[]` occurs on the LSH of an assignment, it contains a *row-list* with just a single *item-list* which may only contain storage expressions or the `~` placeholder. Such a tree structure is obviously somewhat redundant, and thus prime suspects to be streamlined using abstract elements

The storage expressions are insofar on a separate level of the tree as regards their first child, which may be a nested storage expression, for example as in `x.abc{2:4}(1:2)`. The *param-list* and *cell-index* and also the second child of *dyn-struct-ref* may contain general expressions which define the indices or field names of the values accessed. In MATLAB an array index with parentheses may not occur repeatedly in a storage expression, which is why any storage expression with one or more cell array indices possibly followd by a single array index is represented by a single *array* in AST XML. In the GNU Octave dialect the array index may be nested, but this is not supported by ADiMat. Cell array index and structure references may be nested inside each other however, as in `x.abc{1,2}.def{3}(1)`, but a possible parentheses index must still come last. For the AST XML this means, that an *array* element can only have a *var* or a *struct-ref* as the first element. The *struct-ref* in turn can have either a *struct-ref* or a *array* or a *var* as its first child, but when it is an array it will only have *cell-index* but never *param-list* children. While interesting, these structural invariants are not particularly important, at least not important enough to be nailed down with a formal grammer, in our view.

The more interesting structural invariants of storage expressions are probably that when we follow the chain of first children of a nested storage expression we only encounter further storage expressions and always end up with a *var* element. This means that we can clearly define the root element of any storage expression, which we call a *principal storage expression*, and also the *var* element found through the first children on all levels, which we call the *principal variable*.

Storage expressions that are not *var* can be devolved to a function call to either `subsref` of `subsasgn` depending on whether they occur on the right or left hand side of an assignment. MATLAB provides a means to define any index expression by means of a nested structure with certain fields, as described in the **subsref** and **subsasgn** documentation. An assignment with a storage expression on the LHS may be devolved to the plain assignment `x = subasgn(x, ...)`, where `x` is the principal variable. Thus, the principal variable is the one variable in the current scope that has its value changed by the assignment, possibly somewhere down its structure, when it is a struct or a cell array, or only partially when it is an array, according to the storage expression.

According to the data model that we use in ADiMat, as defined in Section 3.1, the derivative of a storage expression is in general the same storage expression with the principal variable replaced by its derivative. Thus, we can simply differentiate

```
x.abc{1,2}.def{3}(1)
```

to

```
g_x.abc{1,2}.def{3}(1)
```

in the FM code and to

```
a_x.abc{1,2}.def{3}(1)
```

in the adjoint code. The only quite subtle yet important exception is when there occur repeated indices in one of the index expressions. Then, unfortunately, the semantic of the expression is different depending on whether it occurs on the LHS or the RHS of an assignment. This means that unfortunately we cannot use the simply rule above in the adjoint code, because any storage expression would have to switch to the other side of the assignment in the associated adjoint statement. This particular challenge and the method we use to solve it was discussed already in Section 3.3.

All the elements mentioned so far reside in the namespace

```
http://www.sc.rwth-aachen.de/ns/adimat
```

which is usually bound to the namespace prefix `adm`. Now we complete the AST XML definition by allowing elements from any other namespace to be attached to these core trees. In particular, in the adjoint code generator we use two further namespaces, one for internal compiler annotations and one for messages. For obvious reasons of simplicity we use the same namespace prefixes throught the entire pipeline, as listed here:

adm http://www.sc.rwth-aachen.de/ns/adimat

adc http://www.sc.rwth-aachen.de/ns/adimat/comments

ada http://www.sc.rwth-aachen.de/ns/adimat/attributes

For the same reason we will in the remainder of this work refer to these three XML namespaces by the name prefixes shown. For example, the AST XML document consists of a core tree in the `adm` namespace, possibly with subtrees from the `ada` or `adc` namespaces, or any other namespace, attached to it.

6.5.2 XML AST examples

In this section we provide some short examples for the AST XML. We give examples for one short input function shown in Listing 15 to be transformed in reverse mode and show the AST XML in graphical form in Figure 23. This graphical form is a very simple and direct representation of the XML sub tree beginning with the *function* element. XML elements are rendered as grey boxes labelled with the element name and text nodes in dark green boxes labeled with the node text. The *binary*, *unary*, and *op* elements are coloured in light green and light blue and are labelled with the *@op* or *@sign* attribute value, respectively. We see for example the parentheses as a light blue box labelled with `()`. Since these nodes only ever have a single child and carry no semantic meaning after the parsing stage, they are in effect redundant and they are removed in one of the first preprocessing steps of our adjoint code generator.

```
function z = fsqb(x, y)
  if abs(y^2 - x) < 1e-7
    z = y;
  else
```

```
    z = fsqb(x, (y + x / y) / 2);
  end
end
```

Listing 15: An example function to demonstrate the use of AST XML

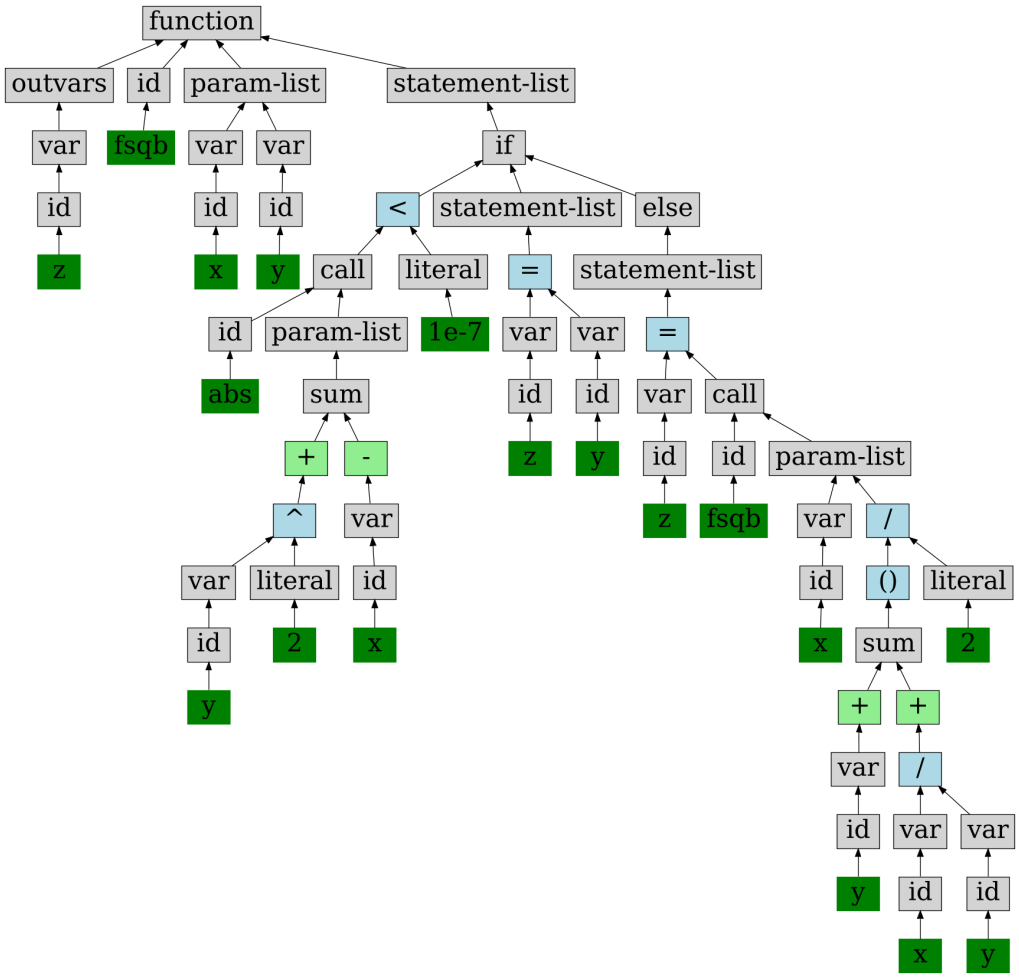

Figure 23: The AST XML document of the example function in Listing 15

```
function [a_x nr_z] = a_fsqb(x, y, a_z)
    tmpca1 = 0;
    tmpca2 = 0;
    tmpca3 = 0;
    z = 0;
    tmpba1 = 0;
    if abs(y^2 - x) < 1e-7
        tmpba1 = 1;
        adimat_push1(z);
        z = y;
    else
        adimat_push1(tmpca3);
        tmpca3 = x / y;
        adimat_push1(tmpca2);
        tmpca2 = y + tmpca3;
```

```
        adimat_push1(tmpca1);
        tmpca1 = tmpca2 / 2;
        adimat_push1(z);
        z = rec_fsqb(x, tmpca1);
    end
    adimat_push1(tmpba1);
    nr_z = z;
    [a_tmpca1 a_tmpca2 a_tmpca3 a_x a_y a_z] = a_zeros(tmpca1, tmpca2,
        tmpca3, x, y, z);
    tmpba1 = adimat_pop1;
    if tmpba1 == 1
        z = adimat_pop1;
        a_y = adimat_adjsum(a_y, a_z);
        a_z = a_zeros1(z);
    else
        [tmpadjc1 tmpadjc2] = ret_fsqb(a_z);
        z = adimat_pop1;
        a_x = adimat_adjsum(a_x, tmpadjc1);
        a_tmpca1 = adimat_adjsum(a_tmpca1, tmpadjc2);
        a_z = a_zeros1(z);
        [tmpadjc1] = adimat_a_mrdividel(tmpca2, 2, a_tmpca1);
        tmpca1 = adimat_pop1;
        a_tmpca2 = adimat_adjsum(a_tmpca2, tmpadjc1);
        a_tmpca1 = a_zeros1(tmpca1);
        tmpca2 = adimat_pop1;
        a_y = adimat_adjsum(a_y, adimat_adjred(y, a_tmpca2));
        a_tmpca3 = adimat_adjsum(a_tmpca3, adimat_adjred(tmpca3, a_tmpca2));
        a_tmpca2 = a_zeros1(tmpca2);
        [tmpadjc1 tmpadjc2] = adimat_a_mrdivide(x, y, a_tmpca3);
        tmpca3 = adimat_pop1;
        a_x = adimat_adjsum(a_x, tmpadjc1);
        a_y = adimat_adjsum(a_y, tmpadjc2);
        a_tmpca3 = a_zeros1(tmpca3);
    end
end
```

Listing 16: The adjoint code generated for the function from Listing 15

Then we show an even smaller example function in Listing 17. Here we also show the XML markup of the *function* subtree in Listing 18. For this second example, we show the graphical AST representation at several stages of the processing, differentiating the function w.r.t. the first parameter `x`:

- At the beginning of the pipeline, shown in Figure 24
- Immediately after the adjoint code generation, shown in Figure 25
- At the end of the pipeline, immediately before the adjoint MATLAB code is generated, shown in Figure 26

The resulting adjoint code is shown in Listing 19.

```
function z = fmtimes(x, y)
  z = x * y;
end
```

Listing 17: A small example function to demonstrate the use of AST XML in the adjoint code generator

```
<function xmlns="http://www.sc.rwth-aachen.de/ns/adimat">
 <outvars>
  <var>
   <id id="0">z</id>
  </var>
 </outvars>
 <id id="3">ftimes2</id>
 <param-list>
  <var>
   <id id="1">x</id>
  </var>
  <var>
   <id id="2">y</id>
  </var>
 </param-list>
 <statement-list>
  <binary op="=">
   <var>
    <id id="0">z</id>
   </var>
   <binary op="*">
    <var>
     <id id="1">x</id>
    </var>
    <var>
     <id id="2">y</id>
    </var>
   </binary>
  </binary>
 </statement-list>
</function>
```

Listing 18: The AST XML document of the example function in Listing 17 at the beginning of the XSLT processing pipeline

In Figure 25 the AST XML document is shown immediately after the adjoint code generation. The function now has two outputs, the adjoint result `a_x` and the function result, which has been renamed from `z` to `nr_z`. The *statement-list* element now has five children:

1. The forward sweep consists of the original statement. No push is required in this case.
2. The function result is available at this point and assigned to the renamed output variable `nr_z` to save it.
3. The adjoint of `x` is initialized using the *initialize-adjoint* abstract element
4. The adjoint of `z` is initialized using the *initialize-adjoint* abstract element
5. The adjoint of `x` is updated with the adjoint of the right hand side of the original assignment w.r.t. `x`, using the abstract elements *adjoint-increment* and *adjoint-left-multiplication*

In Figure 26 the AST XML document is shown immediately before the final MATLAB code is generated. Here the abstract elements have been devolved to regular components of AST XML. The two *initialize-adjoint* elements have been merged into one by a postprocessing step, and so are both handled by a single call to `a_zeros`. The *adjoint-increment* has been devolved to an assignment with a call to `adimat_adjsum` on the RHS and the *adjoint-left-multiplication* has been devolved to a call to the `adimat_adjmultl` runtime function. The resulting adjoint code is shown in Listing 19.

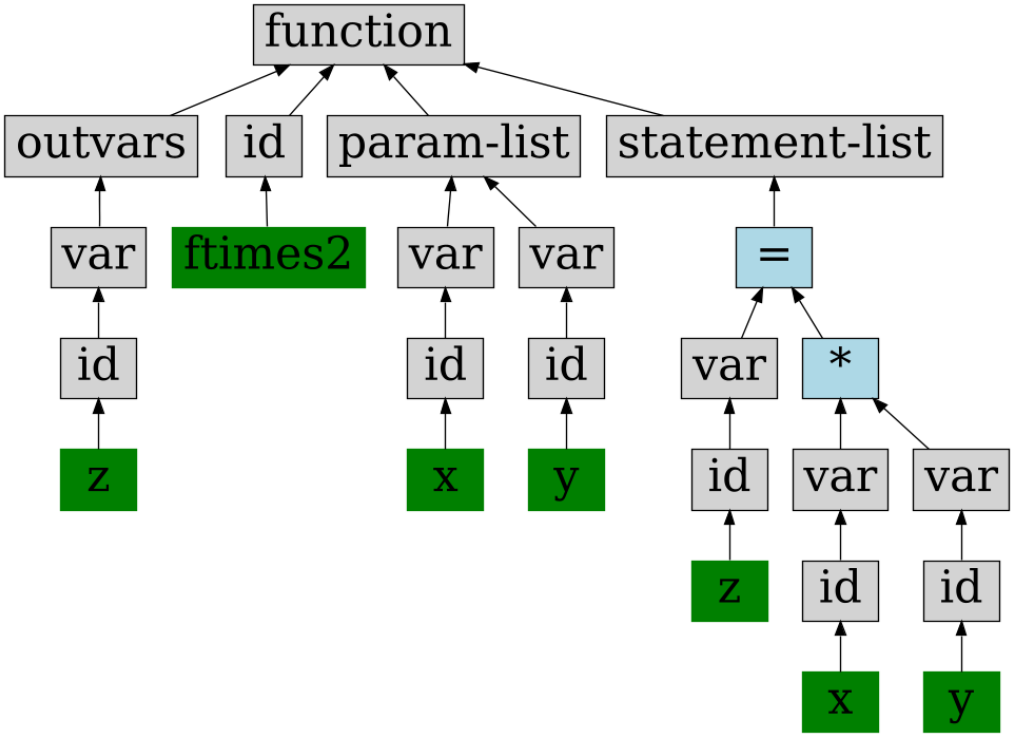

Figure 24: The AST XML document of the example function in Listing 17 at the beginning of the XSLT processing pipeline

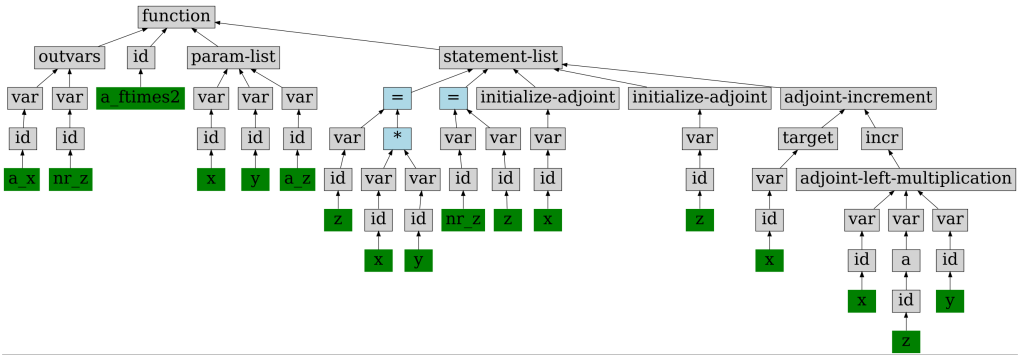

Figure 25: The AST XML document of the example function in Listing 17 immediately after the actual adjoint code generation step

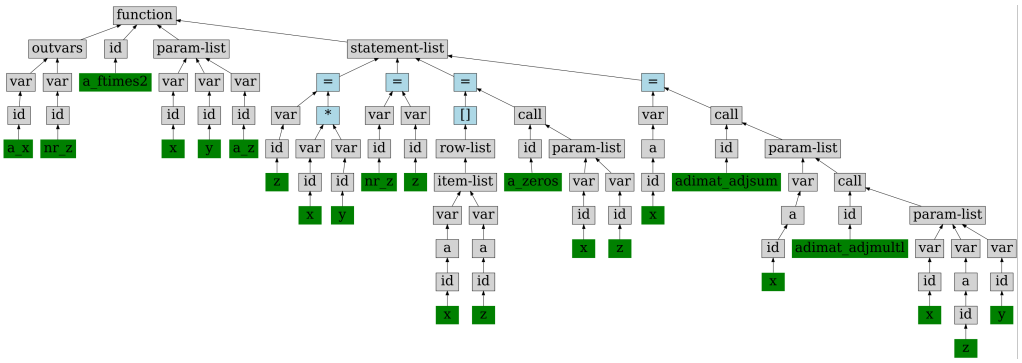

Figure 26: The AST XML document of the example function in Listing 17 end of the XSLT processing pipeline

```
function [a_x nr_z] = a_ftimes2(x, y, a_z)
    z = x * y;
    nr_z = z;
    [a_x a_z] = a_zeros(x, z);
    a_x = adimat_adjsum(a_x, adimat_adjmultl(x, a_z, y));
end
```

Listing 19: The adjoint code generated for the function from Listing 17

6.5.3 Abstract XML AST elements and namespaces

With abstract elements we refer to further elements from the `adm` namespace which carry semantic meaning but are not considered part of the AST XML language. They are often specific to the task of adjoint code generation. They are generated in the adjoint code generator pipeline, mostly by the actual adjoint code generator step. They are devolved to standard AST XML elements by some later step. This allows to simplify the adjoint code generator in many ways.

The abstract elements used in the pipeline are the following:

tmp this wraps the *id* of *var* to denote a temporary variable, with attributes *@kind*, *@name*, etc. Devolves to unique variable names

a this wraps the *id* of *var* to denote an adjoint variable. Devolves to a name prefix, usually `a_`

push-values plain values to push. Devolves to **adimat_push**

push-index indexed values to push. Devolves to **adimat_push_index**

push-field struct fields to push. Devolves to **adimat_push_field**

pop-values plain values to pop. Devolves to **adimat_pop**

pop-index indexed values to pop. Devolves to **adimat_pop_index**

pop-field struct fields to pop. Devolves to **adimat_pop_field**

initialize-adjoint zero an adjoint variable. Devolves to **a_zeros**

adjoint-increment with two elements *target* and *incr*. Devolves to an assignment with **adimat_adjsum** or just `+` on the RHS

adjoint-reduction with two elements *adj* and *value*, check for implicit expansions to reduce. Devolves to **adimat_adjred**

adjoint-reshape with two elements *adj* and *value*, check for implicit reshapes to undo. Devolves to **adimat_adjreshape**

The abstract elements are in our view an important technique to simplify the adjoint code generator. By defining crucial repetitive pieces of the adjoint code in such abstract terms, many tasks can be factored out of the adjoint code generator. For example we can both optimize abstract elements in later steps and also decide how to devolve the abstract elements, either dynamically or through user options. Basically speaking, we are simply inventing new elements in the `adm` namespace, and this allows us to structure our adjoint code first only in broader sketches, and leave the filling in of details for later. This overall approach is discussed in more detail in Section 6.6.

As to optimizing, for example, consecutive elements of *push-values* are merged, thus optimizing the code by reducing the number of statements. The same is done with the *pop-values* and *initialize-adjoint* elements, as can be seen in Figure 26 in Section 6.5.2. To this end, the runtime function **adimat_push** uses the **varargin** special parameter to allow an arbitrary number of pushed values. Then, *push-values* is devolved dynamically to either **adimat_push** or to **adimat_push1**, when it happens to have just one value to push. The runtime function **adimat_push1** does not use the **varargin** special parameter and executes a little bit faster.

The devolution of abstract elements can also be controlled by user options. For example, the option `well-behaved` causes the *adjoint-reduction* and *adjoint-reshape* elements to devolve to just the adjoint, that is, the call to **adimat_adjred** which is basically there for safety reasons [WBB12], is ommited for performance reasons, cf. Section 3.4. As another example, the expansion of *adjoint-increment* can be controlled by a user option to be either a call to **adimat_adjsum**, which recurses through cell arrays and structs, or to a plain addition with `+`. As we shall see later when we consider XSLT code metrics of our pipeline in Section 6.7, the handling of user options is as often a significant chunk of the total amount of code, so again a major part of the code can be factored out in this way.

Another advantage of using abstract elements is that they are often a more concise representation than the devolved AST XML. For example, the content of *incr* of *adjoint-increment* is duplicated in the devolution as it occurs both on the LHS and RHS of the resulting statement, as can be seen in Figure 26 in Section 6.5.2.

6.6 XSLT processing steps for AST XML

A big advantage of XSLT is that the general operation to traverse the AST and thereby faithfully copying all elements, which is also called the *recursive identity transformation* [Wik20c], can easily be made the default behaviour of the XSLT processing. At the same time templates are applied at each node, which means that the user can add special rules for any particular node to effect the intended change to the document. This is achieved by using a very simple XSLT stylesheet called `copy.xsl`, shown in Listing 20.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">

  <xsl:output method="xml"/>

  <xsl:template match="/">
    <xsl:apply-templates select="node()"/>
  </xsl:template>

  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
```

```
</xsl:stylesheet>
```

Listing 20: This XSLT stylesheet provides rules to copy every node while traversing the XML document

Then any intermediate processing step consists of an XSLT stylesheet that has a structure similar to the XSLT stylesheet `remove-nested-statement-lists.xsl` shown in Listing 21. This stylesheet includes the `copy.xsl` stylesheet. This means that we only have to add templates for the operations that we actually want the processing step to perform. For elements not matched by these, the processing will fall back to the rules in `copy.xsl`. The intended operations are two in this case: Firstly, the element *processing* is copied and a new element *step* is appended to its children describing the current step. Presumably, the *processing* element already has a number of *step* elements, which are in effect copied by the `xsl:apply-templates`, which reverts to the default rules in `copy.xsl`. After this output has been emitted the template emits the *step* element that is placed next. Placing this template in corresponding form in every pipeline stylesheet thus produces a list of processing steps in the *processing* element, which is useful for tracing and debugging purposes.

The actual AST transformation is very simple in this case: When a *statement-list* element is immediate child of another *statement-list* element, this is redundant and can be safely removed. Hence the template only recurses into the children with `xsl:apply-templates`, but does not copy the element itself. The statements contained in the nested `statement-list` element are thus inserted into the parent `statement-list` element at that point.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:adm="http://www.sc.rwth-aachen.de/ns/adimat"
  xmlns="http://www.sc.rwth-aachen.de/ns/adimat"
  version="1.0">

  <xsl:include href="copy.xsl"/>

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="adm:processing">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates/>
      <step>remove nested statement lists</step>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="adm:statement-list/adm:statement-list">
    <xsl:apply-templates/>
  </xsl:template>

</xsl:stylesheet>
```

Listing 21: This XSLT stylesheet is a generic example for some intermediate processing step in our XSLT pipeline. Its operation is to remove nested `statement-list` elements, but not its children.

This example showcases some of the principles used in our XSLT pipeline design. By defining the processing step `remove-nested-statement-lists.xsl` we can remove the artifact of a nested *statement-list* at any time in the pipeline. Conversely this means that in all the other processing steps we can proceed agnostically and defensively and wrap any list of statements that we possibly generate in a *statement-list* element, regardless of whether this is actually required or redundant.

The example Listing 21 also shows the use of XML namespaces in our XSLT pipeline: When referring to XML elements which are in a namespace in the XPath expressions inside an XSLT

stylesheet we always have to use a namespace prefix, even if the XML elements does not have a prefix in the document. Thus, even though the element is called just `processing` in the XML document, it actually is in our main AST XML namespace `http://www.sc.rwth-aachen.de/ns/adimat`, because the default namespace is set in the document using the `xmlns` attribute. So in the XSLT stylesheet we bind the prefix `adm` to that namespace and use `adm:processing` to refer to the element in the match expression of first template. In the XSLT we also bind the default namespace to the same main AST XML namespace so that we can output elements from AST XML without a namespace prefix, as in the case of the added `step` element. This approach allows us to work without a namespace prefix for our main AST XML namespace in all of the XML and in the elements that we create via XSLT. Note that both the `processing` and the `step` elements could rather be placed in the AST attributes namespace `http://www.sc.rwth-aachen.de/ns/adimat/attributes` due to their purpose, but since the `processing` element is structurally set apart so well towards the top of the document, this was not deemed necessary.

This simple design for individual filter steps results in a very generic approach which works entirely irrespective of the XML document structure. At each step only the elements that are to be manipulated or whos contents are to be used need to be known by their name, namespace, structure and purpose. For example, both the `ada` and the `adc` namespaces are not used in Listing 21, but should the XML document contain elements from these namespaces, or indeed any further namespaces, they would still be copied and thus be avaílable to steps further down the pipeline.

Thus, these XSLT language properties together with the simple scheme for defining transformation steps, naturally induces a programming paradigm where complex tasks are split into multiple steps. In particular a compiler construction approach is induced which results in a relatively large number of *passes*, i.e. traversals of the AST, because each XSLT filter step is a full AST pass in its own right.

The starting point for most XSLT filter steps is the recursive identity transformation, which is a particularly useful form of the identity transformation, which is of crucial importance in itself in the field of data transformation. In XSLT the recursive identity transformation takes a particularly simple form. The recursive identity transformation has the important property that desired transformations can be effected at any level [Wik20c], by simply adding a template matching the desired node. While many and even complex structural transformations can be expressend in XSLT templates relatively concisely, we have to consider that the situation that probably occurs most often in software development is that we have an existing piece of software and want to achieve some improvement in the data flow, which may require several change at different places. The most obvious danger is always to break anything else, so the transformations we wish to add will often be nearly identity transformations except the for specific changes that we have judged to be required.

For this reason it is of great value that the identity transformation in XSLT is extremely concise even in the recursive form. Further desirable properties of individual XSLT filter steps are the following:

- idempotency
- order interchangeable with or *orthogonal* to other filter steps
- well defined operation
- shortness
- valuable conditions established by the filter step
- half-identity transformations

These properties increase the chance that a filter step can be reused in a similar XML/XSLT processing task, such as AD for some other source language. They also increase the maintainability of the software by offering flexibility regarding the arrangement of the filter steps.

For example, being idempotent makes such a processing step a candidate for being repeated whenever the information it provides or the post condition it enforces might have become invalid. The `remove-nested-statement-lists.xsl` is idempotent for example. When there is no nested *statement-list* it performs the identity transformation on the AST. Being orthogonal to a number of other steps enables the rearrangement of a certain step, moving it past the others in the chain, should it turn out that some order w.r.t. yet another step is required. Such properties are relatively easy to ascertain from the source code of XSLT. For example, we don't have to insert a step of `remove-nested-statement-lists.xsl` after every other step in our pipeline. Not only since do many steps not produce `statement-list` elements, but also because many steps will perform their intended function regardless of the presence of nested *statement-list* elements. Thus, steps that do produce nested `statement-list` elements are still orthogonal to many other steps following it, even though the AST XML with nested *statement-list* elements is technically invalid. Also the `remove-nested-statement-lists.xsl` is othogonal to many other steps so it does not matter if we remove these artifacts now or later in the pipeline. Thus, in our adjoint code generator pipeline, we use `remove-nested-statement-lists.xsl` just in two places, immediately before those two steps that are actually confused by these artifacts: the actual adjoint code generation in `reverse-ad.xsl` and the output source code generation in `to-source.xsl`.

More generally, we think is is appearent that it is often quite natural, and that it requires almost no particular effort, to adhere to a so-called defensive programming style in XSLT. The filter steps can often work on a strict need-to-know basis, they only need to know the names of those elements that they are actually concerned with, but on the other hand can easily be constructed in such a way that regarding any other language elements or annotations the tree is copied as faithful as possible.

In the Listing 21 we also see an example of the literal output principle (cf. Section 6.2.1). To emit a new XML element *step* we simply write that XML element inside the XSLT template handling the *processing* element.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:adm="http://www.sc.rwth-aachen.de/ns/adimat"
  xmlns="http://www.sc.rwth-aachen.de/ns/adimat"
  version="1.0">

  <xsl:include href="copy.xsl"/>

  <xsl:output method="xml"/>

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="adm:processing">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates/>
      <step>rebin: make binary elements truly binary</step>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="adm:*" mode="rebin">
    <xsl:choose>
      <xsl:when test="count(preceding-sibling::adm:*) = 0">
        <xsl:apply-templates select="."/>
      </xsl:when>
      <xsl:otherwise>
        <binary op="{../@op}">
          <xsl:apply-templates select="preceding-sibling::adm:*[1]"
              mode="rebin"/>
```

```
          <xsl:apply-templates select="."/>
        </binary>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

  <xsl:template match="adm:binary">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates select="adm:*[last()-1]" mode="rebin"/>
      <xsl:apply-templates select="adm:*[last()]"/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

Listing 22: This XSLT stylesheet normalizes left-associative *binary* elements. When a *binary* has more then two children, additional *binary* elements are inserted

Another example for an important idempotent operation used to normalize the tree is `rebin.xsl`, shown in Listing 22. This filter normalizes *binary* elements so they have exactly two children. This filter works correctly for left associative binary operators. Again we see the literal output principle at the point where additional *binary* elements are created. The attribute *@op* is set to a computed value using the so called *attribute value templates* of XSLT. We also see the mode feature of XSLT in action in this example. Basically the idea is that the default mode is reserved for the copy-and-traverse semantics while any differing behaviour is implemented in additional modes, in this case the single mode `rebin` is used to traverse the child list from the back to the front. When we reach the first `adm` element, we revert to the default mode, and hence copy the element. Otherwise, a *binary* element is created. The first child of the new element is created by recursively applying mode `rebin` to the preceding element. This will produce either a futher *binary* element or a copy of the preceding element. The current element is copied by reverting to the default mode and hence becomes the second child of the new *binary* element.

As we saw in the introduction to AST XML (cf. Section 6.5) the post condition or `rebin.xsl` is not true for the AST as it comes out of the parser, already. When it comes to differentiation, however, we are better off when we can rely on the fact that there are exactly two children in *binary*. So we apply the filter before entering the adjoint code generation process. Then again the condition is not necessarily true for the result tree after differentiation, as inactive factors in a multiplication may produce no result term, for example, thus producing an addition with a single term. Again, we could simply re-apply the normalization with `rebin.xsl` to remedy the situation.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:adm="http://www.sc.rwth-aachen.de/ns/adimat"
  version="1.0">

  <xsl:output method="text" encoding="utf-8"/>

  <!-- further definitions omitted -->

  <xsl:template mode="to-source" match="adm:binary">
    <xsl:for-each select="adm:*">
      <xsl:if test="position() &gt; 1">
        <xsl:text> </xsl:text>
        <xsl:value-of select="@op"/>
        <xsl:text> </xsl:text>
      </xsl:if>
      <xsl:apply-templates mode="to-source-operand" select="."/>
    </xsl:for-each>
  </xsl:template>
```

```
</xsl:stylesheet>
```

Listing 23: This is an excerpt from the XSLT stylesheet that prints MATLAB code from AST XML, showing the template handling the *binary* elements

Now let us also briefly discuss the final stylesheet in the pipeline, which outputs the MATLAB code corresponding to the AST XML. We continue the discussion of the *binary* elements and show in Listing 23 just the template handling these. At first sight a more obvious implementation would be to rely on the fact that *binary* is supposed to have exactly two children:

```
<xsl:template mode="to-source" match="adm:binary">
  <xsl:apply-templates mode="to-source-operand" select="adm:*[1]"/>
  <xsl:text> </xsl:text>
  <xsl:value-of select="@op"/>
  <xsl:text> </xsl:text>
  <xsl:apply-templates mode="to-source-operand" select="adm:*[2]"/>
</xsl:template>
```

However, it costs us nothing to formulate this template such that it also works correctly with more than two `adm` child elements. Arguably, it even works correctly in the presence of a just single `adm` child element, at least in the cases of `+` and `*` the resulting behaviour is sensible. This is a simple example of the kind of defensive programming that one frequently employ in XSLT pipelines. Also note that either form shown is robust against elements from any further namespaces, due to the prefixed wildcards that we use to select the children. The mode `to-source-operand` is not shown in the excerpt. It has the task of printing the operand either with or without a parenthesis, according to need. To this end it relies on the attribute *@precedence*. This attribute is required as a prerequisite and it is refreshed by a dedicated postprocessing step, which sideloads a table of operator precedences and updates the *@precedence* attribute of all *unary* and *binary* elements according to the *@op* attribute.

A further important class of transformations is what we would call half-identity transformations. These always exist in pairs, and when applied one after the other, produce the identity transformation, so the second is the inverse of the first. In practical terms, we often want to transform the AST, which is in a form A, in some way to arrange it into a more advantageous form B. So we go ahead and define that change as a transformation `a-to-b.xsl`, but we also define the related transformation `b-to-a.xsl` that undoes the change. This allows us to temporarily put the tree in the desired form whenever we wish. Of course, the transformation by `a-to-b.xsl` is probably set up in the first place because if can simplify a subsequent transformation `transform-b.xsl`. In this situation the hope is of course that we can still apply `b-to-a.xsl` without harm after applying `transform-b.xsl`. Without harm meaning that the result is valid in the form A and that the semantical change we achieved with `transform-b.xsl` is also still present in the result. A typical half-identity transformation as we envision it is rather short and concise, and consequently it should usually not be difficult to verify that, by considering the transformations in question.

As an exmple for a pair of half-identity transformations in the adjoint code generator consider the unifications mentioned in Section 6.7. In the preprocessing we devolve `[a,b]`, which is represented by *unary* with *@op* set to `[]` in AST XML, to a function call to **horzcat**. This transformation is not entirely trivial since such an expression may also result in a nested call of **horzcat** and **vertcat**, and we must pay a little bit of attention to avoid code bloat, for example devolve `[a;b]` to `vertcat(a,b)` and not more blindly to `vertcat(horzcat(a),horzcat(b))`. This costs us a couple of modes in the transformation template, but again it is a clear and well defined task that is factored out of the adjoint code generator. In the `reverse-ad.xsl` transformation we now only have to handle function calls to **vertcat** and **horzcat**. In the resulting adjoint code, the so preprocessed code comes out in two versions: in the forward sweep we will find the unified calls, while in the reverse sweep we find the corresponding adjoint statements.

When we now apply the inverse of the unification we will transform the unified calls back to bracketed expressions in the forward sweep, while not changing the adjoint code. Since **vertcat**

and **horzcat** may also occur in the original code we use some attribute to distinguish the unified ones.

In this particular case, the inverse transformation is not strictly required, we could just leave the unified form and emit that. However, the inverse transformation can revert the code back to the original form in the forward sweep code which makes it more readable and recognizable to the user.

Finally we would like to present a small example for a mode that works across several different elements. Lets consider the task of generating the Leibniz differential of an arithmetic expression of `+` and `*`. In XSLT we write this very naturally with a single mode `diff` as shown in Listing 24.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:adm="http://www.sc.rwth-aachen.de/ns/adimat"
  xmlns="http://www.sc.rwth-aachen.de/ns/adimat"
  version="1.0">

  <xsl:include href="copy.xsl"/>

  <xsl:output method="xml"/>

  <xsl:template match="adm:*" mode="diff">
    <xsl:message terminate="yes">Cannot differentiate element <xsl:value-of
        select="name()"/></xsl:message>
  </xsl:template>

  <xsl:template match="adm:binary" mode="diff">
    <xsl:message terminate="yes">Cannot differentiate element binary with
        @op=<xsl:value-of select="@op"/></xsl:message>
  </xsl:template>

  <xsl:template match="adm:var" mode="diff">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <id>d_<xsl:value-of select="."/></id>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="adm:binary[@op='+']">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates mode="diff"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="adm:binary[@op='*']">
    <binary op="+">
      <xsl:copy>
        <xsl:copy-of select="@*"/>
        <xsl:apply-templates mode="diff" select="adm:*[1]"/>
        <xsl:apply-templates             select="adm:*[2]"/>
      </xsl:copy>
      <xsl:copy>
        <xsl:copy-of select="@*"/>
        <xsl:apply-templates             select="adm:*[1]"/>
        <xsl:apply-templates mode="diff" select="adm:*[2]"/>
      </xsl:copy>
    </binary>
  </xsl:template>
```

```
  <xsl:template match="adm:statement-list">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates mode="diff"/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

Listing 24: A simple XSLT stylesheet implementing the differentiation rules for the `+` and `*` operators

In this stylesheet we again use the default mode as the one that copies everything. The mode `diff` is implemented with templates matching certain *binary* elements and *var* elements. For others an error message is produced.

Now we need a trigger, some point we initially use `xsl:apply-templates` with mode `diff` and thus enter the mode. We can choose the *statement-list* element, and differentiate every item. So our stylesheet will accept AST XML where all statement level elements are expressions composed of `+` and `*`.

Then the template of mode `diff` matching *var* elements copies the *var* element and creates a new identifier in an *id* element prefixed with `d_`. The template of mode `diff` matching *binary* elements with *@op* set to `+` also copies the element, so creates a `+` node and recurses into the children with mode `diff`, implementing the rule $\mathrm{d}(a+b) \to \mathrm{d}a + \mathrm{d}b$.

The template of mode `diff` matching *binary* elements with *@op* set to `*` creates creates a new *binary* with *@op* set to `+` node and then creates two copies of the multiplication node inside. Inside the first we recurse with mode `diff` into the first operand while we copy the second, and in the second vice versa. This is in effect writing the Leibniz rule (3) in XSLT. This example pertains to the forward mode of AD of course, and yet it is still simplified. In an AD application we would check at the point of the recursion whether the relative operand is active, for example. However, we think this simple example demonstrates that differentiation is something which is straightforward to implement in XSLT.

The actual adjoint code generation for a nested expression has been described already [Wil10]. Here we compose the adjoint expression inside out while recursing downwards through the tree. That is, we begin with the adjoint of the variable on the LHS of the statement. This output subtree is passed through the template calls via a `xsl:param`. In each template we create a new adjoint expression node which has among its operands the adjoint expression, and pass the result on via a parameter. As explained in Section 6.2, this is on the border of what is possible in XSLT 1.0: The output tree of the adjoint expression constructed so far is passed as a parameter and on each level of the expression a new output tree is created where the existing one is inject in with `xsl:copy-of` at some point.

6.7 The suspension bridge design model for the adjoint code generator

The general software design model we use for the adjoint code generation can be compared to a suspension bridge. There are two rather complex transformation steps and these are also ones that structurally change the code. One is the outlining preprocessing step and the second is the actual adjoint code generator. In each case we can often note the opportunity to simplify these complex software componenents by factoring out certain tasks. These sub tasks are then handled by one or more preprocessing or postprocessing steps, or both, which usually tend to be relatively simple, and as such can be arrange with relative liberty. For example certain annotations need to be refreshed after the outlining preprocessing step.

When the core AST is modified in the sense that its nodes are rearranged such that the pretty printer would yield a different code, we call a transformation structural. When this is not the case a transformation could be called informational, annotational, or incremental.

Structural transformations that are done in preprocessing steps in the adjoint code generator are:

Basic rearrangement Anything we do not like about the XML structure as it comes out of the C++ parser, which is to be kept as simple as possible, is rearranged. After these preliminary steps we call the format AST XML, for which the grammar is given in the appendix.

Index and multiple assignment elimination *Indexed* assignments and *multiple* assignments are both special cases. Eliminating cases of assignments that are both indexed and multiple simplifies the adjoint code generator

Reflexive assignment elimination Reflexive assignments are also a well known special case in adjoint code. These can also be eliminated

Outlining The well-known *outlining* transformation that restricts the height of the expressions and thus inhibits combinatorial explosion in the differentiation stage

Unification Certain operations which have several equivalent names or language idioms are mapped to a unified form, preferably to function call. Two examples: `x+y` is mapped to `plus(x,y)` and `[x,y]` to `horzcat(x,y)`. Generally MATLAB code can be mapped to a function-call-only form, except possibly assignments

Incremental transformations in the preprocessing are:

Activity Set an attribute `active` on any relevant element depending on the activity of the variables

Principality Set an attribute `principal` on any element which represents a storage location, such as `x(k)`, `s.abc(k)`, `c{1}.gef(k)`, etc.

Im postprocessing stage we can highlight the following steps:

Abstraction optimizations The abstractions (cf. Section 6.5) for frequent idioms like push or pop statements, adjoint increments, etc. are rearranged to effect optimizations

Abstraction devolution Towards the end, abstractions are finally expanded into regular AST nodes

Restore unifications Unifications are undone to present the program code in the original form, where it occurs in the adjoint code

Cleanup Certain artefacts from the adjoint code such as empty branches, adjoint increments with 0, etc. are removed. While we do not wantonly produce such artefacts it can reduce complexity to let them occur and clean them up afterwards

Refresh precedence The `precedence` attributes on all unary and binary operators is updated, to ensure correct parenthesisation by the pretty printer. This is more precisely a preprocessing step for the pretty printer

While we do not want do list and discuss all the processing steps in the adjoint code generator in detail, we established some code metrics of the XSLT stylesheets of the processing pipeline. Some meaningful code metrics for XSLT are the following:

Number of Templates Total count of *xsl:template* elements in the stylesheet

Number of Elements Total count of XML elements in the stylesheet, counting both XSLT elements and literal output elements

Number of Modes Number of distinct modes used in the stylesheet

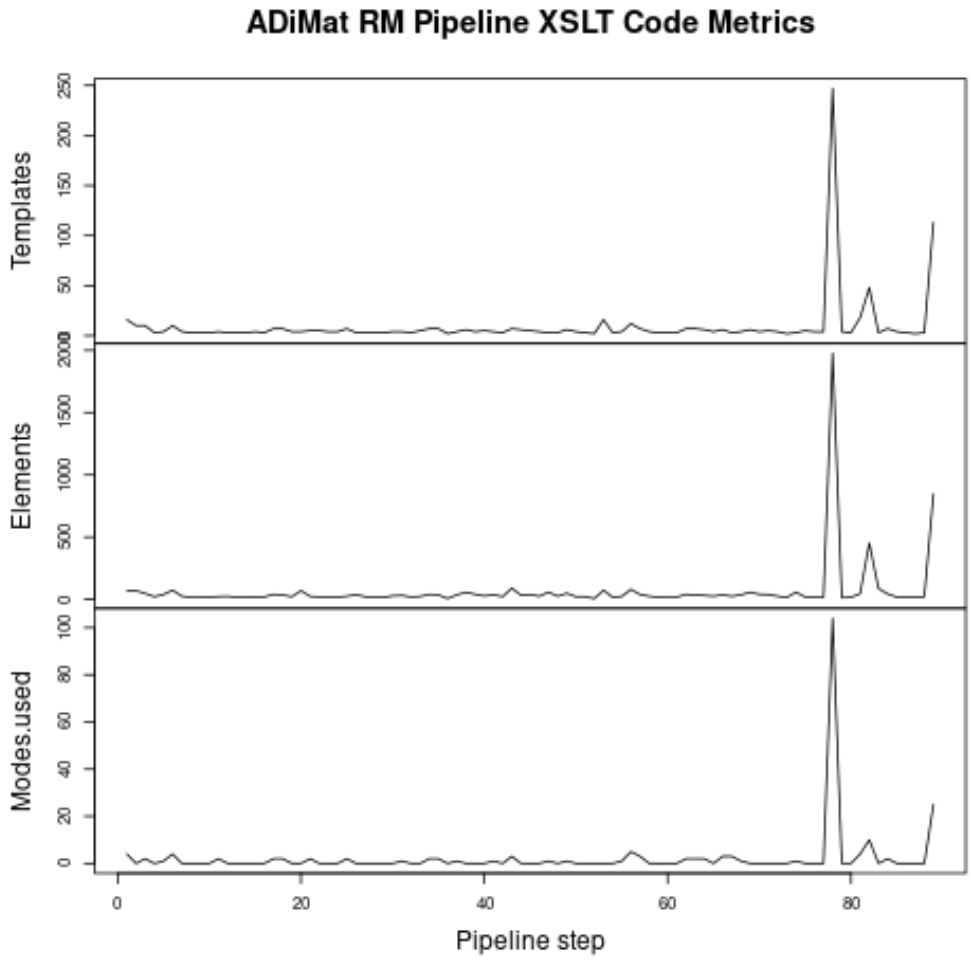

Figure 27: Code metrics for the XSLT stylesheets used by the adjoint code generator over that compiler pass number

While the first and the second metric provide a plain measure for the amount of code in the stylesheet, the number of modes gives the number of different sets of rules and thus the number of different internal states that are needed to perform the desired task, which gives an indication of the semantical complexity of the transformation.

When we establish these metrics on the XSL stylesheets in the adjoint code generator pipeline and plot the result over the step number, the result is as shown in Figure 27.

In the result plot we see that the metrics proposed are quite strongly correlated, in particular the first of the two. Three peaks clearly stand out from among the steps:

reverse-ad.xsl at step 78 the actual adjoint code is generated

renice.xsl a few steps later, the abstract elements are devolved to plain AST XML

to-source.xsl at the very end of the pipeline the MATLAB code is emitted

The peaks of the latter two steps are noticably less pronounced when in the plot of the third metric. This coincides with our qualitative observation that `renice.xsl` is lengthy but conceptually quite simple, since each of the abstract elements is devolved in an entirely local transformation. The high score in the metrics is probably due to the fact that here most of the options are implemented: Should a call to `adimat_adjsum` be generated or adjoints just added with plus, for example. This is done and decided in the template handling *adjoint-increment*, in that stylesheet etc.

The `to-source.xsl` step similarly is relatively lengthy yet conceptually not that involved. What is a challenge here is mostly to produce source code that is not only valid but also properly formatted and indented, which requires quite a number of additional template modes. One intricate task is to properly determine the number of indentation units on each level, but this is of course not critical. More important is to properly place parentheses depending on the precedence of operators, since mistakes here can result in invalid code, which is also not entirely trivial. There are also several options being handled here.

This lets us conclude that the third of the proposed metrics, counting the number of different modes in XSLT, is the most indicative of the actual code complexitiy in semantical terms. As another indication, this metric lets step number 56, which is `canonicalize.xsl`, stand out from the rest, since it has five distinct modes. This is a stylesheet that is of a relatively common size with twelve templates and 81 elements, and hence does not stand out from the crowd in metric 1 and 2. However, it performs the outlining of nested expressions, and hence a major structural transformation of the AST. Two sub tasks are factored out into preceding steps, namely we first detect and and decide which element represent subtrees to be outlined, and mark them with an attribute. This in itself is spread out on two different filter steps. The other subtask is to number the marked subtrees in each statement, which is done in a single filter step. Now the filter `canonicalize.xsl` can decompose the nested expressions into assignments to temporary variable, reusing the same names `tmp1`, `tmp2`, etc. in each statement.

To show how simple most of the processing steps are, we provide a histogram of the third code metric in Figure 28. Here we see that the vast majority of XSL steps uses no modes at all, that is, just the default template set, and twelve of them use a single mode, that is, two template rule sets. Next we have four steps using two modes, three steps using three modes and one step using four modes. On the upper end, we have one stylesheet using five modes (`canonicalize.xsl`), one using ten modes (`renice.xsl`), one with 25 modes (`to-source.xsl`) and one with 104 modes (`reverse-ad.xsl`).

6.8 Facilitating XML and XSLT processing for problem solving

When we use XML and XSLT processing we first of all need some overall control of the structure, some program that implements the sequence of transformations of the input XML, for example by working through a list of file names of XSLT stylesheets. Fortunately there are several options in multiple languages that are all relatively easy to use. This is discussed in Section 6.8.1.

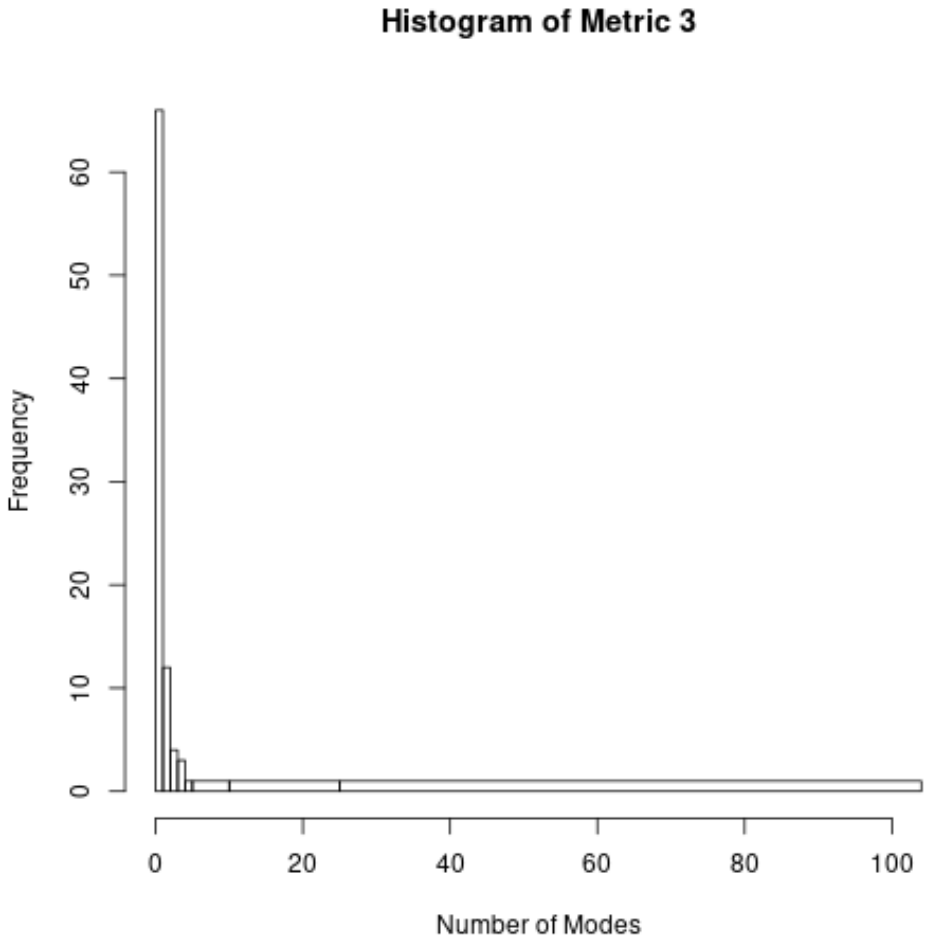

Figure 28: Histogram of XSL code metric 3 for the XSLT stylesheets used by the adjoint code generator with bins 0,1,2,3,4,5,...,10,...,25,...,104

While manipulating XML with XSLT is in our view a highly efficient approach to solve structural processing tasks, we are also very clear in our view that XML is not necessarily a good starting point when a structural problem description should be provided by a human operator.

Many programming languages or other text based formats like CSV, YAML, or JSON provide a considerably more concise and editable representation of such structured problem instances. In such situations we may need methods to translate the problem description to XML first. For a programming language, for example, this would mean that a parser is to be constructed for that language with the sole task of emitting the resulting AST in XML format.

In the following subsections we present two software packages that we have developed specifically to address this problem. The first is P2X, which is a generic recursive descent parser that is configurable to output well-structured XML for almost any structured text format, including many common programming languages. This is described in more detail in Section 6.8.2.

When we are inside a certain programming language, XML datasets can generally be created very easily using just text output to print the XML markup. For example, it is easy to write a shell script that prints out certain file or directory information as XML, and not other tools are required.

The other way is much more difficult. Consider for example accessing information from some XML document from within a plain Unix shell script. The easiest way in that situation is often the generation of shell code with XSLT, to be evaluated by the shell for the desired effect, or the generation of some other text representation which we can handle in shell scripts, such as CSV.

So, the second software example that we present is is called R2X, a R package that can convert between named lists in R and XML documents and vice versa, and so provides a bridge between R and XML. The exploits the structural equivalence between named lists in R and XML documents, which we discussed in the introduction to this chapter. The XML markup is printed from a named list by a simple recursive function. Conversely, a simple XSLT stylesheet generates R code from a given XML document, to be evaluated by the R interpreter to produce the equivalent named list. This tool is described in more detail in Section 6.8.3.

6.8.1 Setting up XSLT pipelines

Setting up XML processing pipelines is appearently so easy that no particular software solution has really established itself so far, neither in the actual XML format nor in the implementation, except possibly that the pipeline definition should of course be in XML [Wik20g]. Most notable instances are the X3C recommendation `XProc` [WMT10] while one of the oldest use of XML pipelines in software was in the Apache Cocoon project [The13].

In our case we require only XSLT filters als elementary pipeline steps, so we also refer to the concept as a *XSLT pipeline*. Inspite of its simplicity XSLT pipelines are a very powerful tool, in our view. One reason could be the that, strictly speaking, it constitutes a form of generative programming, cf Section 6.9. From a formal definition, in the most simple case a list file names or URLs of XSLT stylesheets, we generate the actual program to be run, be it a Unix shell script, and execute it.

To generate a Unix shell code that processes the listed steps one by one, from whatever XML that lists the steps is a very simple and effective solution. This script could call the `xsltproc` program that is the command line interface to the `libxslt` [Vei03] XSLT 1.1 processor for each step. Other XSLT processors like Xerces and Saxon also have command line facilities. Another very interesting option is the `xmlstarlet` [Gru02] program which also provides XPath, validation, pretty-printing, and other useful XML utilities on the command line.

We have ourselfs also once generated so called Makefiles to be processed by the `make` program, to generate a directory structure defined in an XML document. Generating a Makefile which lists the individual processing steps and their dependencies immediately allows the efficient reuse of processed results in the output, and a partial reprocessing depending on which parts of the input have changed.

When we want to do without intermediate files on disk, we have to work within a single process. An immediate and very interesting option here appears to be the tool `xmlsh` [Lee08]. A script

very similar to the Unix shell script proposed above could here do the same processing without intermediate files.

Python has two different interfaces to the `libxslt` XSLT 1.1 processor libraries, one shipped with `libxslt` directly [Vei03] and one in the form of the `lxml` package [Fou20], and the R language has the `xslt` package [Oom17] which also interfaces `libxslt`. MATLAB has the **xslt** function, which interfaces to the Saxon 6.5.5 processor [Kay01b] since at least version R2006a [Mat06]. Both languages have the **eval** function, so we can generate the relevant script from a pipeline description, with XSLT, and execute it to perform the pipeline processing.

XSLT 1.0 is also supported by all major webbrowsers, and available via a JavaScript API. To provide XSLT pipelines in web browsers we have created the *XML Hierarchical Linear Pipelines* (XHLP) project [Wil20d]. XHLP provides a uniform interface to XSLT processing accross the major web browsers together with a set of utility functions on top the XMLHttpRequest (XHR) to facilitate downloading of stylesheets and XML documents, with or without caching, and form submission by POST requests. XHLP pipelines are defined by a lists of steps in JavaScript, which can be created either directly in JavaScript or imported from a simple XML doctype. Each step is either the URL of an XSLT stylesheet, a parsed stylesheet document, the name of a JavaScript function, or the URL of another XHLP pipeline in XML format, hence the name hierarchical. Input documents are either parsed XML documents or XML markup strings which are then parsed to documents on the fly. This facilitates working with document subtrees that are obtained easily via the `innerHTML` or `outerHTML` properties of DOM element nodes, which are in fact available on element nodes of any kind of XML document and crucially also provide well-formed XML [W3C20], while setting these properties is the most simple gateway to inserting new subtrees into the DOM tree [MDN20].

ADiMat uses for the adjoint code generator a custom C++ application processing a simple linear XML pipeline definition together with a parameter facility and some rudimentary branching, which uses the `libxslt` library for individual steps. This allows the caching and reused of compiled stylesheets, which is interesting because of the repeated XSLT processing steps in the pipeline but even more so for the ADiMat transformation server, which can thus avoid hard disk access.

Whatever option is chosen, the programming effort is little to minimal, even for in-process pipelines without intermediate files. An obvious and indespensible enhancement for the latter is the option of writing all intermediate XML documents to the disk for debugging.

For advanced applications it may be required to create iterative processes such as fixpoint iterations based on some basic elementary step. The elementary step, which may in itself be an XSLT pipeline, would perform some operation that after a finite number of steps attains a certain condition on the output. Another XSLT filter or pipeline would be tasked with determining the stop condition, i.e. return a string of either `0` or `1` indicating if the iteration shall be aborted or continued. Using such a feature could be used to implement the AD activity analyis on the AST XML, for example. However, in the case of ADiMat the activity analysis is already implemented in the C++ part and its results are made available in the XML parser output.

6.8.2 P2X

The frequently arising task of parsing structure text formats is solved generically by our P2X parser software [Wil13b], which outputs an XML document for any textual input. P2X is basically a generic recursive descent parser written in C++11. The parse tree is structured according to parenthesisation and unary and binary operators with precedence which are defined by simple configuration files. Thus, the intermediate step of generating and compiling a parser as is required when using parser generators like Bison/YACC [DS00] or ANTLR [PQ95] is avoided.

In our experience, many common language elements as commonly expressed in classical EBNF grammars can also modelled by the comparatively simple rules of P2X. For example, to parse the C language, one might define the common arithmetic operators and also `;` and `,` as binary operators with relatively lower precedence and the parenthesisations `()`, `[]`, and `{}`. The parenthesised items are then themselves defined as postfix unary operators. With these rules the common C language elements like function definitions or function calls result in specific structures in the parse tree

that can then be handled accordingly. In particular we recommend to first use XSLT to translate the P2X output to a more concise XML representation of the C language and then define further transformations for effecting the actual goal, which possibly includes pretty printing the result back to C language code, on that custom XML dialect.

As such P2X has the potential to make the XML and XSLT processing approach applicable to a wide array of tasks involving structural transformations with inputs originating from structured text formats, including many programming languages. Given that input from non-XML structured sources is an important part of XSLT 2.0, cf. Section 6.2, and the software support for this standard is rather limited, P2X appears ideally suited to act as a drop-in replacement to emulate that feature.

The XML output has the additional property that it contains the original input entirely, as tokenized by the lexer. This is even true for input tokens that are handled as irrelevant to the structure, such as source code comments. More precisely, the output is a tree with the leaf-text property and the concatenation of all the leaf elements that carry text nodes in document order reproduces the input identically. This property is quite useful when only a local transformation is required, as large parts of the input may remain unstructured then. However, the number of possible applications is propably rather limited. It is also difficult to maintain this property across non-trivial transformation steps. That is, as soon as we translate the P2X output into some more abstract form we will propably also have to provide an explicit unparsing transformation for that structure, if eventual conversion into some text based format is desired.

An second implementation of P2X in JavaScript is also available as part of the package, thus enabling the use of structured text other than XML directly in the web browser, as input to XML pipelines.

A critical question for P2X is of course the underlying tokenizer or lexer. Ideally this should be freely configurable as well, since tokenization differs subtly from one language to the other. For example, in XML, XPath and XSLT the text `x-y` is a single identifier token while in C it would be three separate tokens. The R language has many syntactic elements not commonly found in others such as the operators `<-`, `<--`, `@`, `$`, and double bracket `[[]]` for element extraction. The C++ version of P2X currently provides several different compiled-in lexers build with Flex [Das07; Lev09] and the more efficient RE2C [Tro20] to choose from. In the JavaScript version of P2X we use a custom lexer that can be configured more flexibly using regular expressions.

6.8.3 R2X

As another example of software to facilitate a smooth and seemless transition to the world of XML documents is R2X [Wil20a], which provides a direct mapping between named lists in R and XML documents, which is possible due to a structural equivalence between the two, cf. Section 6.

The R programming language, which is at the same time procedural and functional, has a quite complex and heterogenous type system, but arguably the central and core data type is the list. List items can be assigned names by setting the `names` attribute of a list. Contrary to the associative maps or dictionaries that are found in most other interpreted languages, such as Python or JavaScript, item names or keys can be used repeatedly and they do not affect the order of the list elements. This establishes the structural equivalence of named lists in R and XML documents, or more precisely, between named lists and XML element trees, and since leaf list items in R can carry a text string, the equivalence extends to leaf-text XML documents, cf. Section 6.1.1. The attribute system in R completes the picture, as it appears as the natural choice for representing the XML attributes. There are also almost no restrictions on the names themselves, given the special quoting available in R. The names do not have to adhere to the conventional rules of programming language identifiers. In particular, XML names which often contain dash characters `-` or with namespace prefixes and thus containing colon characters `:` can be used unchanged. XML Namespaces are not directly supported in R, but we can of course associate the names in namespaces with their prefixed name while in R. XML comments and processing instructions also cannot be mapped by R2X.

There are at least two other packages in R for working with XML, firstly `xml2` [WHO15],

which is an interface to the `libxml2` library [Vei04], and `xslt` [Oom17], which is an interface to the `libxslt` library [Vei03]. The `xml2` packages provides function for reading, parsing and writing XML documents and the DOM interface to navigate XML documents, and the `xslt` package basically provides just a single function, called `xml_xslt`.

The structural equivalence of R named lists and XML documents is exploited in the R2X package directly by providing a bijective map made up by two function `r2x` and `x2r`. The `r2x` function converts a named list in R to an XML document by generating the corresponding XML markup text directly. The function `x2r` is for the main part implemented in a single XSLT stylesheet, which generates the R code of the named list corresponding to the input XML document. This function will ignore text nodes of non-leaf elements, thus being restricted to leaf-text XML documents as input.

By this bijective map the R2X package provides not only a quick and simple method to serialize R lists to XML, to access information in existing XML documents using native R data types and operators, but it also enables the transformation of named list structures with XSLT. Also, it is in effect an alternative to the DOM API, although somewhat incomplete. Interestingly, in Python there is the ElementTree package which uses a similar approach, providing a means to read and write XML documents directly to and from a custom data structure other than the DOM API [Lun07]. The R2X package is however much simpler in that it can use a data structure native to R directly as the target and in that it does not provide any tree searching or manipulation functionality, leaving that also to the means built into~R.

In summaray, R2X provides a shortcut to using structural transformations with XSLT from R, accessing information in XML documents, or exporting structured information in XML format. We could take things even a step further, and not only represent the data in R but also the XSLT stylesheets, although it seems unlikely that this would result in more understandable code.

As a short example of a non-trivial XML document, we show how the ubiquitous `copy.xsl` stylesheet from Listing 20 could be represented in R via R2X, using the overload of the **deparse** function for that R2X adds for XML documents:

```
library(xml2)
library(xslt)
library(r2x)
writeLines(deparse(read_xml('copy.xsl')))
```

This results in the following ouput, which is valid R code representing the structure. The code uses the helper function **element** to more concisely represent values with attributes, as a replacement to the function **structure** that R normally uses to deparse such values. This helper function allows to list the attributes first within each element, so the code resembles more that of the XML markup text.

```
`xsl:stylesheet` <-
    element(`version` = '1.0',
            list(
                `xsl:output` = element(`method` = 'xml'),
                `xsl:template` = element(
                    `match` = '/',
                    list(
                        `xsl:apply-templates` = element(`select` =
                            'node()'))),
                `xsl:template` = element(
                    `match` = '@*|node()',
                    list(
                        `xsl:copy` = list(
                            `xsl:apply-templates` = element(`select` =
                                '@*|node()')
                        )))))
```

With `r2x` we can transform the named list stylesheet back to a proper XSLT document and run a complete id transformation cycle with another named list in R as the input document.

```
library(xslt)
l1 <- list(a=1, b=list(a=2), a=3)
indoc <- read_xml(r2x(l1))
namespaces <- list(xsl = 'http://www.w3.org/1999/XSL/Transform')
xsldoc <- read_xml(r2x('xsl:stylesheet',
                       name = 'xsl:stylesheet',
                       namespaces = namespaces))
l2 <- x2r(xml_xslt(doc = indoc, stylesheet = xsldoc))
l2
```

The last value `l2` is identical to `l1` so the output printed is:

```
$a
[1] 1

$b
$b$a
[1] 2


$a
[1] 3
```

Thus, the R2X package paves the way to define XSLT transformations and possibly even entire XSLT pipelines entirely in R source code. However, in our view this is probably not very practical after all, and the R2X package is more useful in that it provides the ability to seamlessly export and import structured information to and from XML documents.

For example, when a structural problem description is required for a certain task, this structural problem description can be specified directly in the R language in the form of a named list, and transformed to XML whenever necessary, using R2X. This in-language specification of structured problem descriptions is often seen as a desirable feature. It has the potential to, in many settings, eliminate the need to come up with declarative domain specific language, plus the required accessories such as a syntax, a grammar, a lexer, a parser, unparser, documentation, examples, etc. The ability to freely transform between different representations ultimately very much facilitates a completely syntax-agnostic approach to format specification. This means that we only specify the structure of some informational entity or problem description, for example as a EBNF grammar and leave the actual representation, and methods of creation, representation, serialization, storage, etc. up to the user.

6.9 Generative programming with XSLT

Generative programming is a relatively old yet interesting topic of computer programming. It has long been a staple of programming since generating the appropriate code of a computer programm that performs some task is a simple idea yet often also surprisingly simple solution [ACK03; CP03; SA09] to a wide range of problems. More broadly speaking, all kinds of compilers are instances of generative programming.

In our view XML processing with XSLT lends itself particularly well to generative programming. We want to distinguish different types of generative programming in the context of XML pipelines:

1. Generating XSLT with XSLT
2. Generating an XML pipeline definition with XSLT
3. Transforming XML with XSLT
4. The generation of document types, schemas or grammars with XSLT

With 1. we might refer to what could also be called true second order programming, generating a program from another program written in the same language, such as generating a C program code with a C program. In XSLT this situation for doing that is very comfortable, of course, since XSLT stylesheets are XML documents. The only obvious difficulty is how to separate between XSLT elements that are meant to be processed and those that are meant to be output. This is provided for in XSLT by the `xsl:namespace-alias` element, which rebinds some namespace in the output tree to another one immediately after the processing is complete. So, second order XSLT is of course a very interesting option. This technique can be used to implement XPath evaluation of dynamic strings, for example, in a two step process. One could also envision to compose XSLT stylesheets dynamically. A very useful, interesting and early XSL software that makes use of this technique is Graphotron [Nic01], which generates graphs from XML documents according to user specified rules. The basic structure of Graphotron is shown in Figure 29. This software is used to produce the graphs of the AST XML examples in Figures 23 to 26 in Section 6.5.2. A related topic is the transformation of XSLT into other langauges [Béz+03]. However, in our practical work it is not the most commonly used technique, and we do not use it in the adjoint code generator.

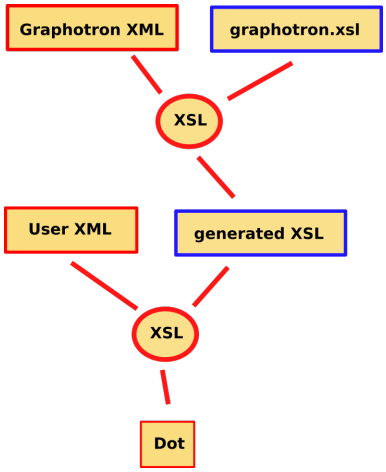

Figure 29: The basic structure of the Graphotron tool. In addition Graphotron can produce output not only in Dot but in three other formats as well

With 3. we refer to the basic process of transformating a single XML document with several XSLT stylesheets applied one after the other in a linear pipeline. This may be seen as generative programming already, in our view, since at each step we can may express information in some XML language dialect and then transform that in the consecutive steps, possibly in some other language dialect, possibly inventing new elements and attributes along the way, to be later processed further, thus establishing their semantic meaning. This what we have discussed at length by presenting our AST XML in Section 6.5, the transformations that make up the adjoint code generator in Section 6.6, and concepts that we use to arrange theses steps in Section 6.7.

With 2. we then refer to more complex networks of XML transformations, to be imagined in a more evolved form as a graph for example, where the nodes are stages in the transformation, defined by certain document types, and the edges are pipelines of XSLT stylesheets. An XML definition of such a structure might also be generated with XSLT or a pipeline of XSLT stylesheets. This might also be done dynamically. This is discussed with respect to the pipeline definitions of our adjoint code generator in Section 6.9.1.

With 4. we refer to the fact that the modern grammars for XML are themselves represented in XML and can thus also be generated with XSLT [CH03]. Also interesting and related is the code generation from document type definitions with XSLT, such as to automatically generate APIs in different languages for a given document type. UML is also a suitable basis for such a

task [Obj17].

6.9.1 Generating XML pipeline definitions

The simple concept of a linear pipeline, in 3. is immediately extendable to tree or graph-like process network, conditional and repeated execution, in 2. The XSLT like solution would be to implement such features by some preprocessing to the pipeline definition, that is, some additional XSLT processing on the pipeline definition itself. In ADiMat for example we use XML documents to define tree-like processing structures, that take one input to multiple possible outputs. From these we extract the linear path from the root to any one of the outputs, in a kind of static branch unrolling. This method allows us to define multiple closely related pipelines in a single document. For example, the adjoint code being generated and emitted in AST XML or in MATLAB source code are two pipelines which differ only in the last step.

From this practical example we observe again that we gain a truly huge amount of expressional power by reverting to second order programming. And that is easily possible, all within our own devices, with XML. We define some ad-hoc XML language P for a linear list of processing steps, another, possibly similar, ad-hoc XML language T that defines a processing tree and a XSLT that performs the path extraction, thereby converting from our langauge T to P. In this particular case the advantage is that we do not have to add branching or other forms of conditional computation to the language P directly, and hence an existing processor for the language P can be used unchanged.

To provide further practical example on the considerations involved with XML processing, let us continue to discuss P and T. In the particular case of T and P both the transformation and the specification of T to P turn out to be not that trivial after all. The reason lies in the fact that, generally speaking, we have to transform a child-parent relation to a sibling relation when transforming from T to P. This itself already is not trivial, but something easily done with XSLT. The real problem comes with a practical consideration: Starting off with existing pipelines in the P format of length 50 or more, we cannot let a newly defined language T have documents with a nesting depth of 50, or else the documents would be completely unwieldy. So what we actually need is a kind of fused tree where the branches are only at those places where an actual branching occurs between the related pipelines, with plain lists of filter steps attached. Then the list of processing steps for a particular output node is the concatenation of the filter step lists found on the path from the root to the output node. Thereby we obtain trees in the T language that are not too deeply nested, at the expense of a somewhat more complex language T and a somewhat more complex transformation between T and P.

In the case of the adjoint code generator all of this is internal to the code generator. That is, in a kind of rapid prototyping approach we start off with some form of T and some kind of transformation. Then, as soon as we manage to extract the desired P pipelines from this process we are done and can stop the development.

6.10 Case study: The XC electronic document system

The *XC* electronic document system has been designed by the author [Wil20c] and is currently being made available via the author's newly founded company *AI & IT* (cf. `http://ai-and-it.de`). This system stores the relevant information in XML documents, which provides a kind of *document as code* technique similar to the configuration as code [Wik20a] and infrastructure as code [Wik20d] techniques that have been devised in recent years in the context of the so-called *DevOps* family of software development and deployment strategies [Wik20b]. The main advantage of such an approach, where the end result is uniquely determined by a set of line based text files, is in our view that the usual versioning control systems such as `Git` can be used to track and manage change sets.

The XC system has a web interface that is also kept as simple as possible, practically only providing the user management via the well-known web development framework `Django`. Django is however used in a rather limited way, basically only providing the Python code of actions on the server, the so called views. The object relational mapper (ORM) layer of Django is used for

the builtin user and session management only and otherwise replaced by working directly with XML documents. These are stored in a common file hierarchy that can be managed via a set of server actions via the web interface, including searching for file names and contents. Hence the organization and layout of the system of documents can be flexibly arranged and may be decided entirely by the user.

The Django views on the server are designed to simply compile the required information in XML format and send it to the web browser without further processing. Any kind of processing and visualization is done in the browser using XSLT processing choreographed by a custom JavaScript framework. Thus, template rendering layer of Django is only very minimally employed in XC, by providing just two templates, one for a HTML page that bootstraps the system for any given entry link, such as `/login/index` or `/main/view?path=/adrs/meyer.xml`, and another template that generically sends an XML document with the response to any of the underlying AJAX end points, such as `/login/ajax_index` or `/main/ajax_view?path=/adrs/meyer.xml`. In the second example the XML response in particular contains the XML document from the relative location `adrs/meyer.xml` in the tree as a subtree. Transformed documents may be send back to the server for storage.

This system provides for a highly efficient multi-user document server that requires only minimal hardware resources by shifting most processing work to the web browser, thus harnessing the compute power available at the end device of the users, be it a PC, a tablet or a smart phone. Of course the relevant XSLT stylesheets must also be transferred to the user.

Any change to the document markup on disk is automatically committed to a Git repository thus providing all the associated features such as integrity, revisability, traceability, backup, including remote transmission of backups, error recovery, be it from system or user errors, and the ability to build decentralized networks of coordinated XC systems. This includes the actual workflows of the deployed system as well, which are implemented as XSLT filters and XML pipeline definitions, that also reside as XML documents in the system and are thus editable and also for all other puprposes treated just as any other document in the XC system.

The visualization of documents in the web browser is done in HTML and SVG, in particular providing a kind of web-font feature using the `opentype.js` JavaScript library [DB17]. This allows to use any OpenType or TrueType font file that is available on the server to render text in SVG vector shapes or *paths* on the end device.

In the currently envisaged application labelled as *XCrm* the documents to be handled are typical business documents suchs as bills, invoices, and offers together with a simple database of addresses. This provides a simple CRM system that produces business documents which can be freely and precisely designed in SVG. Invoices can be extracted from other software, which is trivial in the case of `GNUCash` [Gnu], which uses XML to store its books. In the case of a database-based systems such as `Fakturama` [Fak], the process is slightly more complex. One could either directly access the underlying database and piece together the required information manually or one might provide the software with an XML export filter, similar to the approach we used with the existing ADiMat C++ core, see Section 2.7.

The SVG layout for documents can be freely created by the user, only having to obey very simple rules to specify the places where the document texts shall appear, thus providing a template for appearance of the various document types. XC then extracts the design parameters such as location, dimension and font, font size, etc. from the template and typesets the document texts accordingly at the correct locations. The typesetting algorithm based on `opentype.js` is implemented in JavaScript not XSLT. The resulting SVG is injected into the SVG template, thus producing the result document.

In a last step the SVG form of the documents may be converted to PDFs layed out exactly in the desired paper format. While this last step must currently be performed on the server, or by the user on his PC, using the Apache Batik SVG renderer, or `rasterizer` [The20], the resulting procuments are not only visually appealing but also legally safe since the text rendered in vector graphics cannot be manipulated easily.

The JavaScript framework in XC is designed to generically handle any kind of documents, and to automatically show the available transformation filters and pipelines pertinent to a given

document. This means that the XC document systems can be easily adapted to almost any kind of workflow, by simply adding template XML documents for new document types and XSLT stylesheets and pipelines to transform them to the system, which is possibly via the web interface, just as for any other document. Transformation pipelines can be specified in very simple XML documents listing the steps. A pipeline step may also be another pipeline, allowing for hierarchical organization and reuse of pipeline parts. These transformations are implemented using the XML Hierarchical Linear Pipeline (XHLP) JavaScript library by the author, cf. Section 6.8.1. The association of document types to filters and pipelines in XC is simply by name prefixes. Further specially prefixed pipelines are used for rendering documents to HTML and SVG, and thus provide the visualisation in the browser. As a result the visualizations can also easily be saved as documents on the server, or locally, just as any other transformation result, or send to customers by email.

The XSLT processing is available in all web browsers via a JavaScript API. Minor differences in the API across browsers are abstracted away by the XHLP layer. For extraction of subtrees given by an XPath expression, which is a frequent subtask in XC, we use the two-stage generative programming technique discussed in Secion 6.9.

Also, the XPath function `document()` works as expected which means that additional XML documents can be requested from the server directly from XSLT. The common situation is that a template for a given result type is loaded via `document()` and initially traversed with copy-all rules, only intercepting certain elements where information from the actual input document is to be inserted. Such three-way processing steps that are ideal for producing output according to user-specified templates are much more common in this scenario than in the adjoint code generator for ADiMat, which is basically a very long linear pipeline with some information side-loaded via `document()` in a few instances.

In XC we use the URL specification features of Django to define a generic directory-like download end point such as `/main/getf/<url:path>`. This is defined to perform a pattern search for the path name `*/<url:path>` with the `find` Unix utitlity and diretly return the first file found in the tree. So we may refer to a XSLT stylesheet with an URL like `http://example.com/main/getf/xsl/stylesheet.xsl` we obtain the first matching file named `stylesheet.xsl` that resides in a directory `xsl` anywhere in the tree. When in that stylesheet we use the `xsl:include` element with `href` set to the relative URL `include.xsl` the stylesheet will automatically be downloaded from the same pseudo directory. Likewise we may use XPath expressions like

```
document('/main/getf/company/info.xml')
```

to load some further information into XSLT stylesheets which is still a relative URL and hence will also be downloaded from the correct host `http://example.com`.

6.10.1 Production use of XC system at fionec GmbH

The XC techique is also used sucessfully at the german company *fionec GmbH* (https://www.fionec.de) were the author is also currently employed as a software and hardware architect. Here the XC system is used in two applications: the first is a glass fiber re-spooling station, which is essential to the fiber distribution service provided by *fionec GmbH* for glass fiber products of the US company Corning. The hardware has been implemented using two stepper motors with integrated controllers from *Nanotec GmbH*. These are controlled by a Python application that receives the user parameters such as the desired fiber length and spooling speed, and the fiber type and the spool type, which are required for their geometric dimensions to spool the fiber in the correct length. The XC system is used to implement a simple web interface to this hardware, and the document features are used here to obtain a simple record of the spooling transactions. The maximum amount of fiber that is generally to be respooled is 12.5 km, half the length of a full 25 km standard spool of glass fiber. Respooling this length with 500 RPM on another empty standard spool takes about 40 min, which is why a visual feedback of the progress is deemed essential, while the user management and authentication feature provided by XC is also much welcomed.

The other XC-based application at *fionec GmbH* is the so called environment module. This is an energy efficient micro computer equipped with a sensor module for environment conditions. The XC system again provides a simple web interface for starting and stopping the recording, setting parameters, configure automatic recording schedules. The XC documents in this case are the configurations and of course the actual recorded sensor output. This is stored per measurement in CVS text format, and also send directly and unmodified to the web browser were it is converted to XML and then visualized with XSLT.

6.11 XML document types, schemas and validation

In the beginning of XML development *document type definitions* (DTDs) were used to define XML grammars or *schemas*. A DTD is a text based format that is adapted from the older SGML language. There are also two XML based formats for XML schemas, XML Schema Definition (XSD) [Tho+04; BMC+04] and RelaxNG [CM01], where the latter also has an equivalent text notation similar to classical EBNF, called RelaxNG Compact Notation [Cla02b]. While XML validation is not at all a trivial subject [Mur+05], mature tools and techniques do exist, such as the validator `jing` which can validate XML documents against RelaxNG grammars [Cla01; Cla02a].

There are certain shortcomings to the XML schemas, which in our view limits their usefulness. First of all, it is hard work to create a suitable grammar in the first place. Secondly, the value gained from checking for validity is in itself somewhat dubious, in our view, mostly because it is a binary piece of information. Simply speaking, we either have to create a very loose and generic grammar or we will have very many invalid documents.

And thirdly, from a purely software technical point of view, validity is of interest when using XML across multiple applications, but not so much within a single XML processing pipeline. The reason is of course that within a pipeline it is only the consecutive steps that are the clients of XML that is produced in some step. So we could simply define validity by what consecutive filter steps can handle correctly. Given that the entire pipeline is written by the same developer, one is basically free to do as one pleases, and in particular due to the ease with which one can spontaneously create whatever new element at some stage and the in our view remarkable robustness of the average XSLT filter step against such entirely new elements that appear out of the blue, it is probably even a distinguishing feature of XSLT pipelines that the deviation from validity is the norm for any intermediate stage.

A recurrent and as of yet not satisfactorily solved problem which also affects our AST XML is that of multi-namespace documents. A real world example is a HTML document with interspersed MathML or SVG subtrees. This document type is entirely sensible, has obvious applications and is supported by many browsers [MDN20b; MDN20a]. It is easily defined in words, but accurate grammars for such hybrid document types are difficult to come by. To formally specify the document type a DTD driver was created for XHTML+SVG+MathML [W3C02], but this has limitations such as fixed namespace prefixes that have to be chosen in advance. In HTML5 both *math* and *svg* elements are now incorporated directly. Obviously, neither approach scales well, or would easily allow the addition of yet further document types.

In RelaxNG we have the *Modular Namespaces* (MNS) recommendation [Cla03]. A related work is the *Namespace-based Validation Dispatching Language* (NVDL) [SF16; NK07]. In the Modular Namespaces we find an interesting feature that is called pruning. This means that before validation against a certain document type any elements from a foreign namespace are removed before the actual validation. This would mean in our terms that only the core tree is validated.

This idea of preprocessing steps before validation can be expanded, in our view, to a more generic concept of validation. To the basic pruning operation we can also add our normalization filter steps. So we could define validity for AST XML documents semantically by the following steps:

- Remove elements and attributes not from the `adm` namespace
- Apply the `remove-nested-stmt-lists.xsl` filter

- Apply the `rebin.xsl` filter
- Apply the `renice.xsl` filter
- Remove text nodes that are not children of leaf elements
- Validate against the RelaxNG schema of AST XML

This is just a concept and not currently used in practice. As a concept however it is important since the fact that we have available the individual filters means that we have our AST under control even though it may feature the denormalizations or the abstract elements that the filters remove. We can flexibliy insert one or more of these filters in the pipeline whenever some following filter step requires the corresponding post condition.

In order to give an example of the heterodox relation that a certain filter step in our adjoint code generator pipeline can have to the issue of validaty of the input document, let us consider the pretty printer `to-source.xsl`. As we showed in one of our XSLT examples, in Listing 23 in Section 6.6, we implemented the rule for emitting binary operators defensively, so they may even be de-normalized. So in this respect `to-source.xsl` can handle invalid AST XML. One the other hand, the *@precedence* attribute must be set correctly on each *binary* element, a fact that is not covered by validation. This could be explained by the pretty printer consisting of a combined transformation that is made up of two stylesheets, were the first one refreshes the *@precedence* attributes. Then we discussed that we remove all elements from other namespaces before validating the core tree alone. While we indeed remove all compiler annotations, the elements from the `ada` namespace, in one of the last filter steps, the added elements from the `adc` namespace are kept since they constitute compiler messages. And of course the proper presentation of the errors and warnings from the adjoint code generator is probably to be considered an important part of the pretty printer.

7 Complex arithmetic

In this chapter we want to focus on the complex numbers and complex arithmetic and how they are handled in the AD processes of ADiMat: Let us consider a function $f(c) : \mathbb{C} \to \mathbb{C}$ and its derivatives. The complex number $c \in \mathbb{C}$ is made up of two real components, $c = (x, y)$ where $x \in \mathbb{R}$ and $y \in \mathbb{R}$, called the real part $x = \Re c$ and the imaginary part $y = \Im c$. When i is the imaginary unit s.t. $\mathrm{i}^2 = -1$, so we also have $c = x + \mathrm{i}y$. The function f can also be thought of as two real parts which we call u and v in the following: $f(c) := (u(x, y), v(x, y))$.

When considering a derivative $\mathrm{d}f$ of f, we will naturally want it to be in the complex domain: $\mathrm{d}f \in \mathbb{C}$. On the other hand we may also consider the underlying real components u, v, x, and y, which results in a 2×2 Jacobian matrix. More formally, we may consider a vector function $\hat{f}(r) : \mathbb{R}^2 \to \mathbb{R}^2$ that wraps f:

$$\mathrm{d}\hat{f}(r) = (\Re f(r_1 + \mathrm{i}r_2), \Im f(r_1 + \mathrm{i}r_2)) = (u(r_1, r_2), v(r_1, r_2)). \tag{7}$$

Then the 2×2 Jacobian of $\hat{f}$ would be the canonical form of the derivative of f in terms of real arithmetic:

$$\frac{\mathrm{d}f(c)}{\mathrm{d}c} \cong \frac{\mathrm{d}\hat{f}}{\mathrm{d}r} = \begin{pmatrix} \frac{\mathrm{d}u}{\mathrm{d}x} & \frac{\mathrm{d}u}{\mathrm{d}y} \\ \frac{\mathrm{d}v}{\mathrm{d}x} & \frac{\mathrm{d}v}{\mathrm{d}y} \end{pmatrix} \tag{8}$$

The relation between the real and the complex domain regarding differentiation is given by the famous Cauchy-Riemann equations. They establish two conditions on the complex derivatives, and these conditions are expressed in terms of the derivatives of the real components of the complex

values f and c:

$$\frac{\mathrm{d}\Re f}{\mathrm{d}\Re c} = \frac{\mathrm{d}\Im f}{\mathrm{d}\Im c} \qquad \equiv \frac{\mathrm{d}u}{\mathrm{d}x} = \frac{\mathrm{d}v}{\mathrm{d}y} \tag{9}$$

$$\frac{\mathrm{d}\Im f}{\mathrm{d}\Re c} = -\frac{\mathrm{d}\Re f}{\mathrm{d}\Im c} \qquad \equiv \frac{\mathrm{d}v}{\mathrm{d}x} = -\frac{\mathrm{d}u}{\mathrm{d}y}. \tag{10}$$

These equations hold iff the function f is *complex analytic*, that is, it has a derivative $\mathrm{d}f/\mathrm{d}c \in \mathbb{C}$. This situation is depicted in Figure 30.

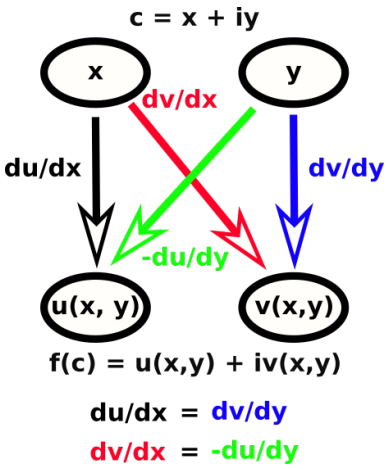

Figure 30: The Cauchy-Riemann equations: The derivative of a complex-valued function $f(c) \in \mathbb{C}$ in a complex parameter $c \in \mathbb{C}$ has four different derivative paths. When the derivative $\mathrm{d}f/\mathrm{d}c$ shall exist as a complex number, the two pairwise relations shown must be true.

The Cauchy-Riemann equations arise from the following consideration: we want to carry the Leibniz rule

$$\mathrm{d}f(c) \to \tfrac{\partial f}{\partial c} \cdot \mathrm{d}c \tag{11}$$

over to the complex domain, given the complex multiplication operation $\cdot$ and the complex values defined as $\mathrm{d}f := (\mathrm{d}\Re f, \mathrm{d}\Im f) = (\mathrm{d}u, \mathrm{d}v)$ and $\mathrm{d}c := (\mathrm{d}\Re c, \mathrm{d}\Im c) = (\mathrm{d}x, \mathrm{d}y)$ and the partial derivative $\frac{\partial f}{\partial c} := (\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x})$. At the same time, the Leibniz rule must also hold for the component-wise operations on the underlying real numbers, i.e. considering $\hat{f}$ instead of f again:

$$\mathrm{d}\Re f = \frac{\partial u}{\partial x}\mathrm{d}x + \frac{\partial u}{\partial y}\mathrm{d}y,$$

and likewise

$$\mathrm{d}\Im f = \frac{\partial v}{\partial x}\mathrm{d}x + \frac{\partial v}{\partial y}\mathrm{d}y.$$

When we now equate these equations with those that result from multiplying out (11), the Cauchy-Riemann equations follow.

Another way to understand the Cauchy-Riemann equations is by considering that whenever a 2×2 matrix is of the form $\begin{pmatrix} a & -b \\ b & a \end{pmatrix}$ it corresponds a the complex number. Thus, when the 2×2 Jacobian (8) of the real components of f is of that form, which means that the Cauchy-Riemann conditions apply, then the derivative of f exists in the form of a complex number and f is analytic. The Cauchy-Riemann equations also have physical interpretations, in particular regarding 2D incompressible flows, and they imply that a complex analytic function is a conformal mapping [Wik19a].

An obvious application of such derivatives is complex-valued optimization, which is the minimization of a real valued objective function of complex parameters: $f(c) : \mathbb{C} \to \mathbb{R}$, where $c = (x, y)$ is a complex number with real and imaginary components x and y. Complex-valued optimization problems [SBL12; SBL13] are of interest in several disciplines, such as recently in solving complex valued inverse problems [Flo+14], or in complex valued neural networks (CVNN) [ZM16].

A conceptual problem is appearent in this case, in that the derivative of a real-valued function f is by definition also real. It must have a zero imaginary part, since $\Im f = 0$. Accordingly, the Cauchy-Riemann conditions are not fulfilled and cease to apply, so a derivative is not defined in complex arithmethic.

On the other hand, the value of $f(c) : \mathbb{C} \to \mathbb{R}$ clearly may depend on both the real and the imaginary component of the variable c, or in other words, $\frac{\mathrm{d}\Re f}{\mathrm{d}\Im c}$ may well be non-zero. So, when we look at the individual real components in the complex numbers we can and must consider the derivatives of f w.r.t. $x = \Re c$ and w.r.t. $y = \Im c$ separately.

So, when a function f is analytic, we compute the limit $\lim_{h\to 0}(f(c+h) - f(c))/h =: \mathrm{d}f/\mathrm{d}c$, and we thus obtain a complex limit value $\mathrm{d}f/\mathrm{d}c = \frac{\mathrm{d}\Re f}{\mathrm{d}\Re c} + \mathrm{i}\frac{\mathrm{d}\Im f}{\mathrm{d}\Re c}$ and this is all that is required. Note that at this point it becomes clear that the limit value must be identical for any complex path that the step h may take to zero. For example, if we wanted to consider a complex directional derivative $\lim_{h\to 0}(f(c + v \cdot h) - f(c))/h =: \mathrm{d}f/\mathrm{d}c|_v$, for $0 < h \in \mathbb{R}$, the value is the same for all $v \in \mathbb{C}$.

When $f \in \mathbb{R}$ however, then the limit $\lim_{h\to 0}(f(c+h) - f(c))/h$ really is just that, a real value that results from taking the limit along some complex path h. Now, when $h = x + \mathrm{i} \cdot 0$, for $x \in \mathbb{R}$, that is, $h \in \mathbb{R}$, then the limit value is just $\frac{\mathrm{d}\Re f}{\mathrm{d}\Re c}$. Similarly, we can obtain $\frac{\mathrm{d}\Re f}{\mathrm{d}\Im c}$ by taking the limit along $h = 0 + \mathrm{i} \cdot y$, for $y \in \mathbb{R}$. Even further, now we can sensibly consider a directional derivative $\lim_{h\to 0}(f(c + v \cdot h) - f(c))/h =: \mathrm{d}f/\mathrm{d}c|_v$ along a complex-valued direction $v \in \mathbb{C}$ and we have, for $h \in \mathbb{R}$, that

$$\mathrm{d}f/\mathrm{d}c|_v := \lim_{h\to 0}(f(c + vh) - f(c))/h = \Re v \cdot \mathrm{d}f/\mathrm{d}x + \Im v \cdot \mathrm{d}f/\mathrm{d}y,$$

is a linear combination of these two derivatives. This means, for some $t \in \mathbb{R}$ such that $c = (x(t), y(t))$,

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t} f((x(t), y(t))) &= \frac{\mathrm{d}f}{\mathrm{d}c}\frac{\mathrm{d}}{\mathrm{d}t}(x(t), y(t)) \\
&= \frac{\mathrm{d}f}{\mathrm{d}c} \cdot \left(\frac{\mathrm{d}x}{\mathrm{d}t}, \frac{\mathrm{d}y}{\mathrm{d}t}\right) \\
&= \left(\frac{\mathrm{d}u}{\mathrm{d}x}, \frac{\mathrm{d}v}{\mathrm{d}x}\right) \cdot \left(\frac{\mathrm{d}x}{\mathrm{d}t}, \frac{\mathrm{d}y}{\mathrm{d}t}\right) \\
&= \left(\frac{\mathrm{d}u}{\mathrm{d}x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\mathrm{d}u}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}t}, \frac{\mathrm{d}v}{\mathrm{d}x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\mathrm{d}v}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}t}\right) \\
&= \left(\frac{\mathrm{d}u}{\mathrm{d}x}\frac{\mathrm{d}x}{\mathrm{d}t}, \frac{\mathrm{d}v}{\mathrm{d}x}\frac{\mathrm{d}x}{\mathrm{d}t}\right) + \left(\frac{\mathrm{d}u}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}t}, \frac{\mathrm{d}v}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}t}\right) \\
&= \frac{\mathrm{d}f}{\mathrm{d}x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\mathrm{d}f}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}t}
\end{aligned}$$

and we see we can differentiate through a complex operation w.r.t. a real parameter t as expected.

So summing up, the Cauchy-Riemann equations mean that we do not have to consider the two possible derivative directions separately in the case of a complex analytic function, because there is a complex derivative value and this captures all four derivatives, by (9) and (10), but when a function is not complex analytic we can and must consider the derivative directions separately. These basics of the differentiation of complex-valued operations are discussed further in Section 7.1

In the application of automatic differentiation to complex arithmetic we should consider first the forward mode. This has already been discussed in depth in [PBC95], for example. In the forward mode, everything basically works as expected and as usual. We have to provide for all the

elementary or builtin functions regarding complex arithmetic, like **real**, **abs**, **angle**, etc. Then, the result is equivalent to finite differences except for the limited accuracy of the latter. Regarding the differentiation of the builtin functions, we can summarize the approach very nicely by observing that we can use arithmetic propagation whenever the function is complex analytic. Then, a partial derivative exists. When it does not, we must resort to structural propagation, which in this context just means that we have to manipulate the real and imaginary parts separately. The FM AD of complex operations is discussed in more depth in Section 7.2.

When we look at the RM, we have to realize that the FM and the RM propagate structurally entirely different derivatives in complex arithmetic. More precisely, when we canonically seed with the complex number $1+\mathrm{i}0$, the imaginary part of the FM derivatives carries the LHS of (10) while the RM adjoints carry the RHS of (10) in the imaginary part. These two are identical when the function is complex analytic, and they may be different when it is not.

For example, consider a real real function $f(c) : \mathbb{C} \to \mathbb{R}$ in a complex variable $c = (x, y)$. Such a function can never be complex analytic. Now, by the RM mode we obtain both $\mathrm{d}f/\mathrm{d}x$ and $-\mathrm{d}f/\mathrm{d}y$ when seeding the adjoint with $\overline{f} = 1$ while the FM will yield only $\mathrm{d}f/\mathrm{d}x$ when seeding with $\mathrm{d}c = 1$. To obtain $\mathrm{d}f/\mathrm{d}y$ in FM we have to compute a second directional derivative by seeding with $\mathrm{d}c = \mathrm{i}$.

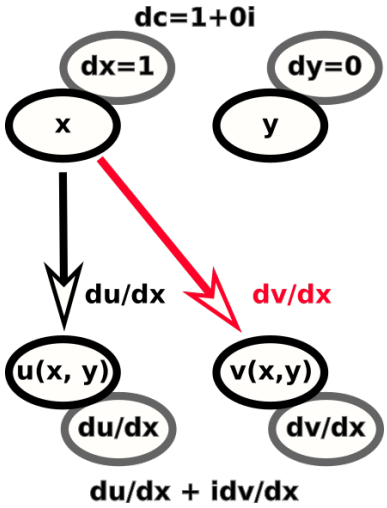

Figure 31: The forward mode propagates two specific components of the complex derivative.

This is depicted in Figure 31 and Figure 32. Figure 31 shows that when seeding the FM process, we can choose a complex number $\mathrm{d}c = (\mathrm{d}x, \mathrm{d}y)$. When we canonically set $\mathrm{d}c = 1$, then the FM delivers the result $\mathrm{d}f/\mathrm{d}c = (\mathrm{d}u/\mathrm{d}x, \mathrm{d}v/\mathrm{d}x)$. Using other values would return a linear combination of (8).

Regarding the basic AD business of the reverse mode, any builtin function that is not complex analytic may require different propagation rules in the RM than in the FM, which is not very surprising considering the above. These issues regarding the RM are discussed in Section 7.3.

Finally we want to provide two practical examples regarding non-analytic functions with complex arithmetic. We will see that generally, by evaluating a directional derivative along the imaginary direction i we obtain the derivative w.r.t. the imaginary component of the input, in forward mode, and the adjoint of the imaginary component of the result in reverse mode, respectively. And this means that both the forward and the reverse mode will return $\mathrm{d}\Im f/\mathrm{d}\Im c$ in the imaginary component when seeded with i. And, as we saw already, the FM will yield $\mathrm{d}\Re f/\mathrm{d}\Im c$ and the RM will yield $-\mathrm{d}\Im f/\mathrm{d}\Re c$ in the real component, which means that we can always compute all four terms of the Cauchy-Riemann equations and (8) with either the FM or the RM. This is discussed in Section 7.4.

Finally, the generic case of complex optimization, the AD of the **norm** function of complex

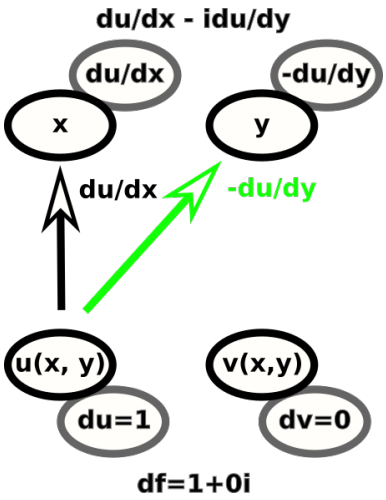

Figure 32: The reverse or adjoint mode propagates a structurally different component of the complex derivative in the imaginary part.

values is discussed in 7.5.

7.1 Methods to evaluate derivatives of non-analytic complex arithmetic

Let us consider a real-valued function in complex variables, $f(c) : \mathbb{C} \to \mathbb{R}$, and its derivative. The first method that we should consider is the composition method. It should be fairly clear that, when we wrap the function f in a bivariate real function $g(x, y) = f(x + \mathrm{i}y)$ and use a given method to differentiate g at the point $(x = \Re c, y = \Im c)$ w.r.t both x and y, we should be able to get the desired derivatives. It obviously works as expected with finite differences:

```
f = @(z) real((2 + 1i) * z)
c = 1 + 2.1i
[J, r] = admDiffFD(f, 1, c)
```

```
f =
@(z) real ((2 + 1i) * z)
c = 1 + 2.1i
J = 2
r = -0.1
```

We first differentiate w.r.t. z as customary, using finite differences. The derivative J is 2, as is expected, and obviously it is a real number, since the function f is real.

```
g = @(x,y) f(x + 1i * y)
[J, r] = admDiffFD(g, eye(2), real(c), imag(c))
```

```
g =
@(x, y) f (x + 1i * y)
J =
  2  -1
r = -0.1
```

The wrapper function g is defined as proposed above and differentiated. The result is the gradient tuple $J = (\mathrm{d}g/\mathrm{d}x, \mathrm{d}g/\mathrm{d}y)$, where the second entry is the derivative of the real function f w.r.t. the imaginary component of the input z, that is $\frac{\mathrm{d}\Re f}{\mathrm{d}\Im c}$.

This method, to obtain the derivatives of f w.r.t. $\Re z$ and w.r.t. $\Im z$, by considering $\Re z$ and $\Im z$ as two separate parameters is well-known and widely used in complex-valued optimization [Sav+12].

It is closely related, or even coincides with the so-called Wirtinger calculus [Wir27], which is also widely used [ZM16; Flo+14]. In Wirtinger calculus the wrapper function g is bivariate in two complex parameters x and y but otherwise identically defined as $g(x, y) = f(x + \mathrm{i}y)$, and it is evaluated at the point $(x, y) = ((c + \mathrm{conj}(c))/2, \frac{\mathrm{i}}{2}(\mathrm{conj}(c) - c))$ [Wik19c].

```
g = @(x,y) f(x + 1i * y)
cr = 0.5 .* (c + conj(c))
ci = 1i .* 0.5 .* (conj(c) - c)
[J, r] = admDiffFD(g, eye(2), cr, ci)
```

```
g =
@(x, y) f (x + 1i * y)
cr = 1
ci = 2.1
J =
  2  -1
r = -0.1
```

This methods also yields the desired results, and from the example is is clear why: $(c + \mathrm{conj}(c))/2 = \Re c$, and $\frac{\mathrm{i}}{2}(\mathrm{conj}(c) - c) = \Im c$. Since the point where we evaluate the derivative is identical, we can assume that the Wirtinger derivatives are equivalent the composition method, or a variant of it.

Another, approach is to consider the derivative of f w.r.t. y as the directional derivative along the direction i, that is

$$\frac{\mathrm{d}f}{\mathrm{d}y} = \lim_{h \to 0} \frac{f(c + \mathrm{i}h)}{h},$$

for $h \in \mathbb{R}$.

The Cauchy-Riemann theorem states that in the complex analytic case the defining derivative limit $\mathrm{d}f/\mathrm{d}z = \lim_{h \to 0} \frac{f(z+\mathrm{i}h)}{h}$ must exist and be identical for any complex path that a $h \in \mathbb{C}$ takes towards 0. Now, this is actually not the case here, but that may well be since the function f is not complex analytic and the Cauchy-Riemann conditions do not apply. However, in real arithmetic the limit taking the path along the imaginary axis clearly exists, and it is different from the limit value that results when taking the real axis.

This method can also readily be tested with finite differences:

```
[J, r] = admDiffFD(f, [1 i], c)
```

```
J =
  2  -1
r = -0.1
```

Note here how we use a seed matrix with two columns $S = [1, \mathrm{i}] \in \mathbb{C}^{(1\times 2)}$ on the univariate function. That means, we evaluate two different directional derivatives for the single parameter z, because it is complex, but the function is not analytic, and we have to consider the real and imaginary component separately. The charme of this method is that no wrapper function is required.

This method also shows that when an automatic differentiation process for f is at hand, we should be able to obtain $\frac{\mathrm{d}f}{\mathrm{d}y}$ by asking it to compute the derivative along the imaginary direction i. Let us now try the described methods with a more interesting example, which is in $\mathcal{C}^\infty$:

```
c = 1 + 2i
f = @(z) real(cos(sin(2 + 1i) * z))
```

```
g = @(x,y) f(x + 1i * y)
[Jcomp, r] = admDiffFD(g, eye(2), real(c), imag(c))
[Jfd, r] = admDiffFD(f, [1 1i], c)
```

```
c = 1 + 2i
f =
@(z) real (cos (sin (2 + 1i) * z))
g =
@(x, y) f (x + 1i * y)
Jcomp =
  -3.1734  -6.8347
r = -3.7119
Jfd =
  -3.1734  -6.8347
r = -3.7119
```

7.2 Application of complex arithmetic in forward-mode AD

As we saw in this chapter already, whenever a function is complex analytic we shall be able to obtain the complex derivative, both components of it, by a single directional derivative. When a function is complex analytic there is no need to consider the real components individually, or the derivatives w.r.t. real and imaginary parts for that matter.

When a function is not complex analytic, one can still fall back to considering the real components, at the cost of doubling the expense by having to consider twice the number of parameters or twice the number of directional derivatives.

This basic considerations above also shows the way to go about constructing an AD process for complex arithmetic: When a elementary computation is complex analytic we may apply arithmetic propagation (see Section 5.1.1), that is, we can create the local Jacobian and multiply by it. Otherwise, we have to resort to manipulation of the real and imaginary components, which results in a structural propagation (Section 5.1.2).

The latter regards those builtins that specifically convert and manipulate between complex and real numbers, such as **conj**, **abs**, **arg**, **real**, **imag**, and **complex**, which do not have a partial derivative in complex arithmetic. However, for the purpose of AD the right thing to do in such situations is to consider the operation on the elementary real numbers, the real and imaginary components, and provide the corresponding propagation rules for them.

One of the more interesting cases is **abs**. The corresponding runtime function for the FM propagation is called **g__abs** and is defined in ADiMat as follows, not showing any error handling code:

```
function [g_r r]= g_abs(g_p, p)
  r = abs(p);
  g_r = (real(p) .* real(g_p) + imag(p) .* imag(g_p)) ./ r;
```

This is the differentiation of $\sqrt{(\Re p)^2 + (\Im p)^2}$, performed in real terms, in a kind of structural propagation. Note that this code also handles the real case correctly. If we wanted to distinguish between a real and a complex case we must tread carefully for pitfalls: we still have to take care of a possible imaginary component in the derivative, as that must be discarded when p is real:

```
function [g_r r]= g_abs(g_p, p)
  r = abs(p);
  if isreal(p)
    g_r = sign(p) .* real(g_p);
  else
    g_r = g_p;
    g_r = (real(p) .* real(g_p) + imag(p) .* imag(g_p)) ./ r;
  end
```

If we left out the **real** in the **if** branch the code would be wrong. Basically, just because $p \in \mathbb{R}$ we cannot expect $\mathrm{d}p \in \mathbb{R}$ since p may be part of a complex valued computation and just happen to have a zero imginary part – in which case **isreal** applies. Another possibility is obviously that the code is in fact real in its entirety but the user chooses to set the derivative direction to a $v \in \mathbb{C}$ from the complex numbers.

7.3 Application of complex arithmetic in reverse-mode AD

When it comes to the reverse mode, another interesting aspect comes into play. In particular, the FM and the RM propagate structurally entirely different derivatives whenever complex arithmetic occurs.

We saw in the previous Section 3.5 already that the expansion from real to complex numbers is a case that is different all the other changes to data types and structures. In particular we saw that it would be wrong to expect an adjoint to be real only because the corresponding program variable is real. We also saw in the previous section that the same is frue for the FM.

From these considerations already, the correct approach to follow in the adjoint code is to consider the whole program to be in complex arithmetic, with the real variables just being complex numbers which happen to have a zero imaginary part.

The Cauchy-Riemann equations describe how the AD process must work on the underlying real numbers, the real and imaginary parts of complex numbers. The forward mode propagates *derivatives* $\mathrm{d}x \equiv (\mathrm{d}\Re x, \mathrm{d}\Im x) = (\mathrm{d}\Re x/\mathrm{d}\Re t, \mathrm{d}\Im x/\mathrm{d}\Re t)$, the reverse mode *adjoints* $\overline{x} \equiv (\overline{\Re x}, -\overline{\Im x}) = (\mathrm{d}\Re f/\mathrm{d}\Re x, -\mathrm{d}\Re f/\mathrm{d}\Im x)$. Thus the FM and RM both yield the LHS of (9) in the real part, but when they yield complex derivatives they return the LHS and RHS of (10), respectively, in the imaginary part. Accordingly FM and RM must produce the same result whenever the differentiated computations are *complex differentiable* or *analytic*. However, they may also yield different results in the imaginary part when the computations are not analytic, as in that case (10) does not hold.

The derivative on the other side of both (9) and (10), respectively, can be obtained in both FM and RM by seeding with the imaginary unit i. Then the FM produces the derivative $\mathrm{d}x/\mathrm{d}\Im c \equiv (\mathrm{d}\Re x/\Im c, \mathrm{d}\Im x/\mathrm{d}\Im c)$ while the RM produces the adjoints $(-\mathrm{d}\Im f/\mathrm{d}\Re x, \mathrm{d}\Im f/\mathrm{d}\Im x)$.

For all practical purposes the same effect can also be achieved with wrapper functions that construct the input from $2N$ real inputs, call the original function, and split the result into $2M$ real outputs.

Contained in these completely symmetrical rules is the case of a real-valued function of a complex variable. Here the FM produces the real result $(\mathrm{d}\Re f/\mathrm{d}\Re c, 0)$ while the RM produces the complex result $(\mathrm{d}\Re f/\mathrm{d}\Re x, -\mathrm{d}\Re f/\mathrm{d}\Im x)$.

This means that we can obtain all four possible derivatives of a scalar complex computation in either the FM or the RM, which is obviously very important in cases where only one of them is available, and also for testing purposes. Finite differences can be used equivalently to the FM. However, this is not always possible in Matlab as finite differencing with a complex step changes the input variable type to complex but certain builtins like **atan2** or **complex** accept only real arguments.

Obviously it is necessary to provide adjoint propagation rules for all the builtins that specifically convert and manipulate between complex and real numbers, such as **conj**, **abs**, **arg**, **real**, **imag**, and **complex**. This class is particularly interesting since for the purpose of complex arithmetic no function from complex to reals is differentiable. However, for the purpose of AD the right thing to do in such situations is to consider the operation on the elementary real numbers, the real and imaginary components, and provide the corresponding propagation rules for these. Unsurprisingly, these need to be different in the FM and the RM.

Continuing the example of the **abs** builtin, in the RM we use the following runtime function **a__abs**, again shown without error handling code:

```
function a_p = a__abs(a_r, p)
  [a_r a_i] = a_hypot(real(a_r), real(p), imag(p));
  a_p = complex(a_r, -a_i);
```

The adjoint is propagated by recurring to the two adjoints of the **hypot** builtin, which are composed to form a complex number. Any imaginary part of a_r must be discarded.

To give another example, the FM AD code of `imag(x)` is very obviously `imag(g_x)`, which is once again the typical structural propagation rule in FM: a self-differentiating one. The adjoint code however is not so obvious:

```
function [a_c nr_z] = a_fimag(c, a_z)
    z = imag(c);
    nr_z = z;
    a_c = a_zeros1(c);
    a_c = a_c + 1i .* -real(a_z);
end
```

Any imaginary part of a_z must be discarded, and the real part of a_z is placed in the imaginary part of the adjoint negated.

7.4 Case Study: A fully non-analytic example

A case where all four terms in the Cauchy-Riemann equations are actually different is quite hard to come by in the real world, but as an example we construct the following function `fconjtest`:

```
function r = fconjtest(x)
  r = abs(x) + exp(conj(x));
```

The **conj** and **abs** operations are not analytic, hence the Cauchy-Riemann equations do not hold, and the FM and the RM yield different values.

Let us calculate the function and its derivatives at one example point x in the complex domain:

```
x = 2 + 1i
z = fconjtest(x)
```

```
x = 2 + 1i
z = 6.2284 - 6.2177i
```

First we apply the FM to compute the directional derivative along $v = 1$:

```
admDiffFor(@fconjtest, 1, x)
```

```
ans = 4.8868 - 6.2177i
```

Then we apply the RM to compute the adjoint directional derivative along $w = 1$:

```
admDiffRev(@fconjtest, 1, x)
```

```
ans = 4.8868 + 5.7705i
```

At this point we have obtained the derivative $\frac{\mathrm{d}\Re f}{\mathrm{d}\Re x}$ with both the FM and the RM. Furthermore, the FM yielded as the imaginary component the derivative $\frac{\mathrm{d}\Im f}{\mathrm{d}\Re x}$ while the RM yielded as the imaginary component the derivative $-\frac{\mathrm{d}\Re f}{\mathrm{d}\Im x}$.

As we saw already, we can verify both of these different imaginary components with the other AD mode, respectively. Seeded with i, the FM yields $\frac{\mathrm{d}\Re f}{\mathrm{d}\Im x}$ in the real component.

```
admDiffFor(@fconjtest, i, x)
```

```
ans = -5.7705 - 3.9923i
```

Seeded with i, the RM yields $-\frac{\mathrm{d}\Im f}{\mathrm{d}\Re x}$ in the real component, since $\mathrm{i}\overline{x} = \frac{\mathrm{d}\Re f}{\mathrm{d}\Im x} + \mathrm{i}\frac{\mathrm{d}\Re f}{\mathrm{d}\Re x}$ and (10).

```
admDiffRev(@fconjtest, i, x)
```

```
ans = 6.2177 - 3.9923i
```

And finally, both the FM and the RM, when seeded with i, yield the derivative $\frac{\mathrm{d}\Im f}{\mathrm{d}\Im x}$ in the imaginary component. If we want to obtain the derivatives of the two real components in this computation w.r.t. to the two real components of the input, then we may use either the FM or the RM, and in either case we require two directional derivatives.

We check both of these directional derivatives against finite differences using the composition method:

```
admDiffFD(@(x,y) fconjtest(complex(x,y)), eye(2), real(x), imag(x))
```

```
ans =
  4.8868 - 6.2177i  -5.7705 - 3.9923i
```

7.5 Case study: the norm function and application to complex optimization

The derivative of a real objective function w.r.t. the imaginary parts of the inputs is required in the field of complex optimization [SBL12; SBL13]. There clearly may be a dependency of a real function $f : \mathbb{C}^N \to \mathbb{R}^M$ on the imaginary part of an input variable. This is the directional derivative $\mathrm{d}\Re f/\mathrm{d}\Im c = \lim_{h\to 0} f(c+\mathrm{i}h)/h$. We can either compute this value with the RM in time overhead $O(M)$ or with the FM in time overhead $O(2N) = O(N)$ using one of the techniques described above. As an example consider the following, where we differentiate the **norm** builtin applied to a complex argument:

```
function r = fnormtest(x)
  r = norm(x);

x = 1 + 2i
z = fnormtest(x)
J1 = admDiffFor(@fnormtest, 1, x)
J2 = admDiffRev(@fnormtest, 1, x)
J3 = admDiffFor(@fnormtest, [1 i], x)
J4 = admDiffFD(@(x,y) fnormtest(complex(x,y)), eye(2), real(x), imag(x))
J5 = admDiffRev(@fnormtest, [1; i], x)
```

```
x = 1 + 2i
z = 2.2361
J1 = 0.44721
J2 = 0.44721 - 0.89443i
J3 =
  0.44721  0.89443
J4 =
  0.44721  0.89443
J5 =
  0.44721 - 0.89443i
  0 + 0i
```

The RM yields a complex derivative (in $J2$) with the RHS of (10) as the imaginary part, while the FM derivative is real (in $J1$). The derivative in the imaginary part of the adjoint result can also be obtained either with the FM or FD by seeding with i (in $J3$) or by constructing the complex input from two reals and thus differentiating w.r.t. the imaginary part explicitely (in $J4$). When we seed with i in the reverse mode, we recompute the imaginary part of $J1$, and also $\frac{\mathrm{d}\Im f}{\mathrm{d}\Im x}$: both must be zero (in $J5$).

8 Conclusion

In this work we present the overall architecture and design of the reverse mode of automatic differentiation for MATLAB in the ADiMat source transformation. Several key topics are covered that are fundamental to the functionality of the generated adjoint code.

A convenient use interface was created that enables the user to easily apply ADiMat to his numerical function in any of the available modes of automatic differentiation: the forward mode and the reverse mode, both in scalar and vector mode respectively, and the forward-over-reverse mode for Hessian evaluations. The user can conveniently specify a seed matrix and is presented with a product of that seed matrix with the Jacobian or Hessian matrix in the common mathematical sense in return, with ADiMat automatically generating the required AD code and running it as required, including sparsity exploitation when possible. The common numerical differenrentiation methods using finite differences and the so called complex step method are provided with the same interface.

An efficient I/O framework called RIOS is available for the storage of large scale datasets called stacks that arise in the reverse mode evaluation. However this framework must clearly be seen as a bridging technology. The existing implementation of the reverse mode in ADiMat can also hope to benefit from future advances in computational technology, where an ever closer integration of large and fast memory with CPUs and FPUs is expected to occur. These advances will relativate the peculiarly large memory requirements of the reverse mode. Whether or not a special prefetching layer for the stack when read in reverse direction will still be needed then remains to be seen.

In many cases the adjoint computations that arise from MATLAB are not expressible in plain expressions, due to the fact that certain implicit behaviour in the forward evaluation must be undone, like binary scalar expansion (BSX), array reshaping, type propagation, data manipulation. This sometimes intricate logic must be outsourced into several runtime functions that arise in the adjoint code. The adjoint code generator can be flexibly configured to handle well-behaved code more efficiently by omitting some or all of these.

The propagation from real to complex values that can occur in MATLAB is another example for implicit behaviour in forward evaluation of MATLAB that requires almost no special consideration in the case of the forward mode of AD, but which becomes somewhat intricate in the reverse mode. In this case however it is somewhat surprisingly wrong to undo the type propagation in the adjoint code. More generally we can state that in complex-valued arithmetic, when seed with 1, the FM and RM compute the two repective sides of the second of the famous Cauchy-Riemann equations in the imaginary component. This can result in different results whenever a function is not analytic, i.e. complex differentiable. However, one obtains the correct derivatives of the real and imaginary part of the inputs w.r.t. those of the outputs, while they exist. For example, any function mapping complex to reals must have a zero imaginary part in its derivative, by definition, and hence in the forward mode. The adjoint evaluation in the reverse mode however naturally contains the dependence of the real part of the function w.r.t. the imaginary part of the inputs. This is of course interesting for the field of complex optimization. However, the same result can equivalently be obtained through the forward mode, by computing derivatives along the direction of the imaginary unit.

Another fundamental aspect in the development of ADiMat are more generally the many high-level mathematical builtin functions in MATLAB. Each of them must be treated carefully with the adequate derivative rules. Among those handled by ADiMat that have been added over the years of development are among many others the Bessel and the Legendre functions, the matrix exponential **expm**, the convolution operator **conv** and the Kronecker product **kron**. Arguably the most desirable case is when the partial derivative in the form of the local Jacobian matrix is available, as this allows for a very uniform and effective derivative propagation. This interestingly is also the only way to handle **subasign** and **subsref** correctly in the adjoint code, even though the only case which strictly mandates it is very much a fringe case, namely the occurence of repeated indices in the index set. Another option are so called structural propagations, where for example **mean** and **fft** are differentiated by themselves. This however requires that the derivative class provides the corresponding methods and in most cases the propagation in RM will be more

involved. A subclass of structural propagation is the application of AD to a given algorithm. This very elegant technique is not only used internally in ADiMat for certain operations where a derivative is especially hard to come by mathematically, in particular certain matrix operations like **expm**, **chol** and **qr**, but it is also open to any user of ADiMat.

Finally we present the novel method of compiler construction that we use in the adjoint code generator. The AST is represented by an in-memory XML document that is transformed step by step by several XSLT stylesheets. In particular the use of XML namespaces proves itself very useful in the handling of AST annotations. We also present some generic methods and techniqes of organizing XSLT pipelines. Another important aspect for enabling this programming paradigma is of course the provision of the input problem representation in XML in the first place. While in the case of ADiMat an existing parser for MATLAB was augmented to enable the XML output of the AST, we also discuss generic methods to handle text based input in an XML processing context by special input filters, some of which have been developed by ourself.

References

[AÅD12] Joel Andersson, Johan Åkesson, and Moritz Diehl. "CasADi: A Symbolic Package for Automatic Differentiation and Optimal Control". In: *Recent Advances in Algorithmic Differentiation*. Ed. by Shaun Forth et al. Vol. 87. Lecture Notes in Computational Science and Engineering. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 297–307. ISBN: 978-3-642-30023-3. DOI: 10.1007/978-3-642-30023-3.

[ACK03] Giuseppe Attardi, Antonio Cisternino, and Andrew Kennedy. "CodeBricks: code fragments as building blocks". In: *SIGPLAN Not.* 38.10 (June 2003), pp. 66–74. ISSN: 0362-1340. DOI: 10.1145/966049.777396. URL: http://doi.acm.org/10.1145/966049.777396.

[AMH09] Awad H. Al-Mohy and Nicholas J. Higham. "Computing the Fréchet Derivative of the Matrix Exponential, with an Application to Condition Number Estimation". In: *SIAM Journal on Matrix Analysis and Applications* 30.4 (Jan. 2009), pp. 1639–1657. DOI: 10.1137/080716426. URL: https://dl.acm.org/doi/10.1137/080716426.

[Ayg+09] E. Ayguade et al. "The Design of OpenMP Tasks". In: *IEEE Transactions on Parallel and Distributed Systems* 20.3 (2009), pp. 404–418.

[Bat77] KJ Bathe. *ADIMAT, a finite element program for automatic dynamic incremental analysis of temperature*. Tech. rep. Massachusetts Institute of Technology, Cambridge, MA, 1977.

[Bay+17] Atilim Günes Baydin et al. "Automatic Differentiation in Machine Learning: A Survey". In: *J. Mach. Learn. Res.* 18.1 (Jan. 2017), 5595–5637. ISSN: 1532-4435.

[BB08] Bradley M. Bell and James V. Burke. "Algorithmic Differentiation of Implicit Functions and Optimal Values". In: *Advances in Automatic Differentiation*. Ed. by Christian H. Bischof et al. Vol. 64. Lecture Notes in Computational Science and Engineering. Berlin: Springer, 2008, pp. 67–77. ISBN: 978-3-540-68935-5. DOI: 10.1007/978-3-540-68942-3_7.

[BBV05] Christian H. Bischof, H. Martin Bücker, and Andre Vehreschild. "A Macro Language for Derivative Definition in ADiMat". In: *Automatic Differentiation: Applications, Theory, and Implementations*. Ed. by H. M. Bücker et al. Vol. 50. Lecture Notes in Computational Science and Engineering. New York, NY: Springer, 2005, pp. 181–188. DOI: 10.1007/3-540-28438-9_16.

[BCG93] Christian H. Bischof, George F. Corliss, and Andreas Griewank. "Structured Second- and Higher-Order Derivatives Through Univariate Taylor Series". In: *Optimization Methods and Software* 2 (1993), pp. 211–232.

[Ben96] Jochen Benary. "Parallelism in the Reverse Mode". In: *Computational Differentiation: Techniques, Applications, and Tools*. Ed. by Martin Berz et al. Philadelphia, PA: SIAM, 1996, pp. 137–147. ISBN: 0–89871–385–4.

[Ber+10] Evangelos Bertakis et al. "Validated simulation of droplet sedimentation with finite-element and level-set methods". In: *Chemical Engineering Science* 65.6 (2010), pp. 2037–2051. ISSN: 0009-2509. DOI: https://doi.org/10.1016/j.ces.2009.11.043. URL: http://www.sciencedirect.com/science/article/pii/S0009250909008446.

[Ber+96] Martin Berz et al., eds. *Computational Differentiation: Techniques, Applications and Tools*. Philadelphia, PA: SIAM, 1996. ISBN: 0–89871–385–4.

[BEV06] H. Martin Bücker, Atya Elsheikh, and Andre Vehreschild. "A System for Interfacing MATLAB with External Software Geared Toward Automatic Differentiation". In: *Mathematical Software - ICMS 2006*. Ed. by Andrés Iglesias and Nobuki Takayama. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 373–384. ISBN: 978-3-540-38086-3.

[Béz+03] Jean Bézivin et al. “First experiments with the ATL model transformation language: Transforming XSLT into XQuery”. In: *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*. 2003, p. 50.

[BG92] Christian Bischof and Andreas Griewank. “ADIFOR – A FORTRAN system for portable automatic differentiation”. In: *4th Symposium on Multidisciplinary Analysis and Optimization*. 1992, pp. 433–441. DOI: 10.2514/6.1992-4744. URL: https://arc.aiaa.org/doi/abs/10.2514/6.1992-4744.

[BGJ91] Christian Bischof, Andreas Griewank, and David Juedes. “Exploiting parallelism in automatic differentiation”. In: *Proceedings of the 1991 International Conference on Supercomputing*. Ed. by Elias Houstis and Yoichi Muraoka. Also appeared as Preprint MCS–P204–0191, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., January 1991. Baltimore, Md.: ACM Press, 1991, pp. 146–153. DOI: 10.1145/109025.109067.

[BGN00] R. H. Byrd, J. C. Gilbert, and J. Nocedal. “A Trust Region Method Based on Interior Point Techniques for Nonlinear Programming”. In: *Mathematical Programming* 89.1 (2000), pp. 149–185.

[BH96] Christian H. Bischof and Mohammad R. Haghighat. “Hierarchical Approaches to Automatic Differentiation”. In: *Computational Differentiation: Techniques, Applications, and Tools*. Ed. by Martin Berz et al. Philadelphia, PA: SIAM, 1996, pp. 83–94. ISBN: 0–89871–385–4.

[Bha+03] Suparna Bhattacharya et al. “Asynchronous I/O support in Linux 2.5”. In: *Proceedings of the Linux Symposium*. 2003, pp. 371–386.

[BHN99] R. H. Byrd, Mary E. Hribar, and Jorge Nocedal. “An Interior Point Algorithm for Large-Scale Nonlinear Programming”. In: *SIAM Journal on Optimization* 9.4 (1999), pp. 877–900.

[Bis+02] C. H. Bischof et al. “Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs”. In: *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*. 2002, pp. 65–72.

[Bis+02] Christian H. Bischof et al. “Combining Source Transformation and Operator Overloading Techniques to Compute Derivatives for MATLAB Programs”. In: *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*. Los Alamitos, CA, USA: IEEE Computer Society, 2002, pp. 65–72. DOI: 10.1109/SCAM.2002.1134106.

[Bis+08] Christian H. Bischof et al., eds. *Advances in Automatic Differentiation*. Vol. 64. Lecture Notes in Computational Science and Engineering. Berlin: Springer, 2008. ISBN: 978-3-540-68935-5. DOI: 10.1007/978-3-540-68942-3.

[Bis91] Christian H. Bischof. “Issues in Parallel Automatic Differentiation”. In: *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. Ed. by Andreas Griewank and George F. Corliss. Philadelphia, PA: SIAM, 1991, pp. 100–113. ISBN: 0–89871–284–X.

[Bis+92a] Christian Bischof et al. “ADIFOR: Automatic differentiation in a source translator environment”. In: *Papers from the international symposium on Symbolic and algebraic computation*. 1992, pp. 294–302.

[Bis+92b] Christian H. Bischof et al. “ADIFOR: Generating Derivative Codes from Fortran Programs”. In: *Scientific Programming* 1.1 (1992), pp. 11–29.

[Bis+96] Christian Bischof et al. “ADIFOR 2.0: Automatic differentiation of Fortran 77 programs”. In: *IEEE Computational Science and Engineering* 3.3 (1996), pp. 18–32.

[BLV03] Christian Bischof, Bruno Lang, and Andre Vehreschild. "Automatic Differentiation for MATLAB Programs". In: *PAMM* 2.1 (2003), pp. 50–53. DOI: 10.1002/pamm.200310013. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/pamm.200310013.

[BMC+04] Paul Biron, Ashok Malhotra, World Wide Web Consortium, et al. "XML schema part 2: Datatypes". In: *W3C Recommendation* (2004). URL: https://www.w3.org/TR/xmlschema-2/.

[Bou06] Jake Bouvrie. *Notes on convolutional neural networks*. Tech. rep. cogprints.org, 2006.

[BPV08] H. Martin Bücker, Monika Petera, and Andre Vehreschild. "Code Optimization Techniques in Source Transformations for Interpreted Languages". In: *Advances in Automatic Differentiation*. Ed. by Christian H. Bischof et al. Springer, 2008, pp. 223–233. ISBN: 978-3-540-68935-5. DOI: 10.1007/978-3-540-68942-3_20.

[Bra+00] Tim Bray et al. *Extensible markup language (XML) 1.0*. 2000. URL: http://www.w3.org/TR/xml/.

[BRV08] H. M. Bücker, A. Rasch, and A. Vehreschild. "Automatic Generation of Parallel Code for Hessian Computations". In: *OpenMP Shared Memory Parallel Programming, Proceedings of the International Workshops IWOMP 2005 and IWOMP 2006, Eugene, OR, USA, June 1–4, 2005, and Reims, France, June 12–15, 2006*. Ed. by M. S. Mueller et al. Vol. 4315. Lecture Notes in Computer Science. Berlin / Heidelberg: Springer, 2008, pp. 372–381. DOI: 10.1007/978-3-540-68555-5_30.

[Büc02] H. M. Bücker. *Hierarchical Algorithms for Automatic Differentiation*. Habilitationsschrift. Aachen: Faculty of Mathematics, Computer Science, and Natural Sciences, Aachen University, Nov. 2002.

[Büc+05] H. Martin Bücker et al., eds. *Automatic Differentiation: Applications, Theory, and Implementations*. Vol. 50. Lecture Notes in Computational Science and Engineering. New York, NY: Springer, 2005. DOI: 10.1007/3-540-28438-9.

[Büc+10] H. M. Bücker et al. "Discrete and Continuous Adjoint Approaches to Estimate Boundary Heat Fluxes in Falling Films". In: *Optimization Methods & Software* 26.1 (2010), pp. 105–125. DOI: 10.1080/10556780903341711.

[Büs+14] Henrik Büsing et al. "Using exact Jacobians in an implicit Newton method for solving multiphase flow in porous media". In: *Int. J. Computational Science and Engineering* 9 (2014), pp. 499–508.

[BV08] H. Martin Bücker and Andre Vehreschild. "Coping with a Variable Number of Arguments when Transforming MATLAB Programs". In: *Advances in Automatic Differentiation*. Ed. by Christian H. Bischof et al. Vol. 64. Lecture Notes in Computational Science and Engineering. Berlin: Springer, 2008, pp. 211–222. ISBN: 978-3-540-68935-5. DOI: 10.1007/978-3-540-68942-3_19.

[BW17] H. M. Bücker and D. Walther. "Automatic Differentiation of Computer Programs in the Time and Frequency Domain". In: *2017 European Conference on Electrical Engineering and Computer Science (EECS)*. 2017, pp. 335–340.

[BW18] H. Martin Bücker and Johannes Willkomm. "Estimating the expansion coefficients of a geomagnetic field model using first-order derivatives of associated Legendre functions". In: *Optimization Methods and Software* 33.4-6 (2018), pp. 924–944. DOI: 10.1080/10556788.2018.1448086. URL: https://doi.org/10.1080/10556788.2018.1448086.

[BWZ70] David Barton, I. M. Willers, and R. V. M. Zahar. "The Automatic Solution of Ordinary Differential Equations by the Method of Taylor Series." In: *The Computer Journal* 14.3 (1970), pp. 243–248.

[CD+99] James Clark, Steve DeRose, et al. *XML path language (XPath)*. 1999. URL: http://www.w3.org/TR/xpath.

[CH03] Krzysztof Czarnecki and Simon Helsen. "Classification of model transformation approaches". In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. Vol. 45. 3. USA. 2003, pp. 1–17.

[Che+08] Y. Chen et al. "Hiding I/O latency with pre-execution prefetching for parallel applications". In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2008)*. 2008, pp. 1–10. DOI: 10.1109/SC.2008.5213209.

[Che09] Y. Chen. "A Hybrid Data Prefetching Architecture for Data-Access Efficiency". PhD thesis. Department of Computer Science, Illinois Institute of Technology, 2009.

[Cho02] Noam Chomsky. *Syntactic structures*. Berlin New York: Mouton de Gruyter, 2002. ISBN: 3110172798.

[Chr92] Bruce Christianson. "Automatic Hessians by Reverse Accumulation". In: *IMA J. Numerical Anal.* 12 (1992), pp. 135–150. DOI: 10.1093/imanum/12.2.135.

[Cla01] James Clark. *A RELAX NG validator in Java*. 2001. URL: http://www.thaiopensource.com/relaxng/jing.html.

[Cla02a] James Clark. *An algorithm for RELAX NG validation*. 2002. URL: http://www.thaiopensource.com/relaxng/derivative.html.

[Cla02b] James Clark. *RELAX NG Compact Syntax*. Nov. 2002. URL: https://www.oasis-open.org/committees/relax-ng/compact-20021121.html.

[Cla03] James Clark. *Modular Namespaces (MNS)*. 2003. URL: https://relaxng.org/jclark/mns.html.

[Cla+99] James Clark et al. "XSL transformations (XSLT)". In: *World Wide Web Consortium (W3C)* 103 (1999). URL: http://www.w3.org/TR/xslt.

[CM01] James Clark and Makoto Murata. *RELAX NG Specification*. Committee Specification 3 December 2001. Dec. 2001. URL: https://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

[CM05] P. Cusdin and J.-D. Müller. "On the Performance of Discrete Adjoint CFD Codes using Automatic Differentiation". In: *International Journal of Numerical Methods in Fluids* 47.6-7 (2005). http://www.ea.qub.ac.uk/pcusdin, pp. 939–945. URL: http://www3.interscience.wiley.com/cgi-bin/abstract/109880352/ABSTRACT.

[Cor+02] George Corliss et al., eds. *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Computer and Information Science. New York, NY: Springer, 2002.

[Cor+92] George F. Corliss et al. *Automatic Differentiation Applied to Unsaturated Flow — ADOL–C Case Study*. Technical Memorandum ANL/MCS–TM–162. Argonne, Ill.: Mathematics and Computer Science Division, Argonne National Laboratory, Apr. 1992.

[Cou+03] F. Courty et al. "Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation". In: *Optimization Methods and Software* 18.5 (2003), pp. 615–627.

[CP03] V Cechticky and A Pasetti. "Generative programming for space applications". In: *DASIA 2003*. Vol. 532. 2003, p. 3.

[CR84] George F. Corliss and Louis B. Rall. "Automatic generation of Taylor series in Pascal-SC: Basic operations and applications to differential equations". In: *Trans. of the First Army Conference on Applied Mathematics and Computing (Washington, D.C., 1983)*. Research Triangle Park, N.C.: ARO Rep. 84-1, U. S. Army Res. Office, 1984, pp. 177–209.

[Das07] Vinu V Das. *Compiler Design using FLEX and YACC*. PHI Learning Pvt. Ltd., 2007.

[DB17] Frederik De Bleser. *opentype.js – JavaScript parser/writer for OpenType and TrueType fonts*. 2017. URL: https://opentype.js.org/.

[Din+07] Xiaoning Ding et al. "DiskSeen: exploiting disk layout and access history to enhance I/O prefetch". In: *2007 USENIX Annual Technical Conference, Proceedings of the*. USENIX Association. Santa Clara, California, USA, 2007, pp. 1–14.

[DL09] P. M. Dickens and J. Logan. "Y-lib: A user level library to increase the performance of MPI-IO in a Lustre file system environment". In: *Proceedings of the 18th ACM international symposium on high performance distributed computing*. HPDC '09. Garching, Germany: ACM, 2009, pp. 31–38. ISBN: 978-1-60558-587-1. DOI: 10.1145/1551609.1551617. URL: http://doi.acm.org/10.1145/1551609.1551617.

[DS00] Charles Donnelly and Richard Stallman. "Bison. the yacc-compatible parser generator". In: (2000).

[ELN06] Javier Esclapés and Mercedes Llorens Nicolau. *ADIMAT, asistente de diagramas de fases para ingenieros de materiales*. 2006. URL: http://rua.ua.es/dspace/handle/10045/2821.

[Fak] *Fakturama – Die kostenlose OpenSource Faktura-Software*. July 2020. URL: https://www.fakturama.info/.

[Flo+14] Anisia Florescu et al. "A Majorize-Minimize Memory Gradient method for complex-valued inverse problems". In: *Signal Processing* 103 (2014). Image Restoration and Enhancement: Recent Advances and Applications, pp. 285 –295. ISSN: 0165-1684. DOI: https://doi.org/10.1016/j.sigpro.2013.09.026. URL: http://www.sciencedirect.com/science/article/pii/S0165168413003915.

[For06] Shaun A. Forth. "An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB". In: *ACM Transactions on Mathematical Software* 32.2 (June 2006), pp. 195–222. URL: http://doi.acm.org/10.1145/1141885.1141888.

[For09] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*. Message Passing Interface Forum, 2009.

[For+12] S. Forth et al., eds. *Recent Advances in Algorithmic Differentiation*. Vol. 87. Lecture Notes in Computational Science and Engineering. Berlin: Springer, 2012. ISBN: 978-3-642-30022-6. DOI: 10.1007/978-3-642-30023-3.

[Fou20] Python Software Foundation. *XPath and XSLT with lxml*. July 2020. URL: https://lxml.de/xpathxslt.html.

[Gay15] David M Gay. "The AMPL modeling language: An aid to formulating and solving optimization problems". In: *Numerical analysis and optimization*. Springer, 2015, pp. 95–116.

[Gay91] David M Gay. "Automatic differentiation of nonlinear AMPL models". In: *Automatic Differentiation of Algorithms: Theory, Implementation, and Application* (1991), pp. 61–73.

[Gay96] David M Gay. "More AD of nonlinear AMPL models: computing Hessian information and exploiting partial separability". In: *Computational Differentiation: Applications, Techniques, and Tools* (1996), pp. 173–184.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. Cambridge, Massachusetts: The MIT Press, 2016. ISBN: 9780262035613. URL: https://www.deeplearningbook.org.

[GC91] Andreas Griewank and George F. Corliss, eds. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. Philadelphia, PA: SIAM, 1991. ISBN: 0–89871–284–X.

[Gil08] Mike B. Giles. "Collected Matrix Derivative Results for Forward and Reverse Mode Algorithmic Differentiation". In: *Advances in Automatic Differentiation*. Ed. by Christian H. Bischof et al. Vol. 64. Lecture Notes in Computational Science and Engineering. Berlin: Springer, 2008, pp. 35–44. ISBN: 978-3-540-68935-5. DOI: 10.1007/978-3-540-68942-3_4. URL: https://people.maths.ox.ac.uk/gilesm/files/AD2008.pdf.

[GJU96] Andreas Griewank, David Juedes, and Jean Utke. "ADOL–C, A Package for the Automatic Differentiation of Algorithms Written in C/C++". In: *ACM Trans. Math. Software* 22.2 (1996), pp. 131–167.

[GK03] Ralf Giering and Thomas Kaminski. "Applying TAF to generate efficient derivative code of Fortran 77-95 programs". In: *PAMM* 2.1 (2003), pp. 54–57. DOI: 10.1002/pamm.200310014. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/pamm.200310014.

[GKS05] R. Giering, T. Kaminski, and T. Slawig. "Generating efficient derivative code with TAF: Adjoint and tangent linear Euler flow around an airfoil". In: *Future Generation Computer Systems* 21.8 (2005), pp. 1345 –1355. ISSN: 0167-739X. DOI: 10.1016/j.future.2004.11.003. URL: http://www.sciencedirect.com/science/article/pii/S0167739X04001785.

[Gnu] *GNUCash – Free Accounting Software*. July 2020. URL: https://www.gnucash.org/.

[GR13] S. Gross and A. Reusken. "Numerical simulation of continuum models for fluid-fluid interface dynamics". In: *European Physical Journal Special Topics* 222.1 (2013), pp. 211 –239. ISSN: 1951-6355. DOI: 10.1140. URL: https://doi.org/10.1140/epjst/e2013-01836-9.

[Gri+99] Andreas Griewank et al. *ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++*. Tech. rep. Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167. Institute of Scientific Computing, Technical University Dresden, 1999.

[Gro+08] W. Gropp et al. "Self-consistent MPI-IO Performance Requirements and Expectations". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by A. Lastovetsky, T. Kechadi, and J. Dongarra. Vol. 5205. Lecture Notes in Computer Science. 10.1007/978-3-540-87475-1_25. Springer Berlin/Heidelberg, 2008, pp. 167–176. ISBN: 978-3-540-87474-4. URL: http://dx.doi.org/10.1007/978-3-540-87475-1_25.

[Gru02] Mikhail Grushinskiy. *XMLStarlet Command Line XML Toolkit*. 2002. URL: http://xmlstar.sourceforge.net/.

[GTL99] William Gropp, Rajeev Thakur, and Ewing Lusk. *Using MPI-2: Advanced features of the message passing interface*. MIT press, 1999.

[GUW00] Andreas Griewank, Jean Utke, and Andrea Walther. "Evaluating Higher Derivative Tensors by Forward Propagation of Univariate Taylor Series". In: *Mathematics of Computation* 69 (2000), pp. 1117–1130.

[GVW96] G. A. Gibson, J. S. Vitter, and J. Wilkes. "Strategic directions in storage I/O issues in large-scale computing". In: *ACM Computing Surveys* 28.4 (1996), pp. 779–793.

[GW00] Andreas Griewank and Andrea Walther. "Algorithm 799: Revolve: An Implementation of Checkpoint for the Reverse or Adjoint Mode of Computational Differentiation". In: *ACM Transactions on Mathematical Software* 26.1 (Mar. 2000). Also appeared as Technical University of Dresden, Technical Report IOKOMO-04-1997., pp. 19–45. ISSN: 0098-3500. URL: http://doi.acm.org/10.1145/347837.347846.

[GW08a] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. 2nd. Other Titles in Applied Mathematics 105. Philadelphia, PA: SIAM, 2008. ISBN: 978–0–898716–59–7. URL: http://www.ec-securehost.com/SIAM/OT105.html.

[GW08b] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. 2nd. Other Titles in Applied Mathematics 105. Philadelphia, PA: SIAM, 2008. ISBN: 978–0–898716–59–7.

[HAP05] Laurent Hascoët and Mauricio Araya-Polo. "The Adjoint Data-Flow Analyses: Formalization, Properties, and Applications". In: *Automatic Differentiation: Applications, Theory, and Implementations*. Ed. by H. M. Bücker et al. Vol. 50. Lecture Notes in Computational Science and Engineering. New York, NY: Springer, 2005, pp. 135–146. DOI: 10.1007/3-540-28438-9_12.

[HH16] Desmond J Higham and Nicholas J Higham. *MATLAB guide*. Vol. 150. Siam, 2016.

[HHG05] Patrick Heimbach, Chris Hill, and Ralf Giering. "An efficient exact adjoint of the parallel MIT General Circulation Model, generated via automatic differentiation". In: *Future Generation Computer Systems* 21.8 (2005), pp. 1356 –1371. ISSN: 0167-739X. DOI: 10.1016/j.future.2004.11.010. URL: http://www.sciencedirect.com/science/article/pii/S0167739X04001797.

[HNN02] Paul D Hovland, Uwe Naumann, and Boyana Norris. "An XML-based platform for semantic transformation of numerical programs". In: *Software Engineering and Applications* (2002), pp. 530–538.

[HNP05] Laurent Hascoët, Uwe Naumann, and Valérie Pascual. ""To be recorded" analysis in reverse-mode automatic differentiation". In: *Future Generation Computer Systems* 21.8 (2005), pp. 1401–1417. DOI: 10.1016/j.future.2004.11.009.

[Hof87] Matthias Hofmann. "Experimental and mathematical investigation of response characteristics and aging phenomena in safety fuse elements". PhD thesis. Technische Univ., Brunswick (Germany, F.R.)., Jan. 1987.

[Hon+14] Sungpack Hong et al. "Simplifying scalable graph processing with a domain-specific language". In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 2014, pp. 208–218.

[HP04] Laurent Hascoët and Valérie Pascual. *TAPENADE 2.1 User's Guide*. Rapport technique 300. Sophia Antipolis: INRIA, 2004. URL: http://www.inria.fr/rrrt/rt-0300.html.

[HP13] L. Hascoët and V. Pascual. "The Tapenade Automatic Differentiation tool: Principles, Model, and Specification". In: *ACM Transactions on Mathematical Software* 39.3 (2013), 20:1–20:43. URL: http://dx.doi.org/10.1145/2450153.2450158.

[HWB15] Alexander Hück, Johannes Willkomm, and Christian Bischof. "Source Transformation for the Optimized Utilization of the Matlab Runtime System for Automatic Differentiation". In: *Recent Trends in Computational Engineering - CE2014: Optimization, Uncertainty, Parallel Algorithms, Coupled and Complex Problems*. Ed. by Miriam Mehl, Manfred Bischoff, and Michael Schäfer. Cham: Springer International Publishing, 2015, pp. 115–131. ISBN: 978-3-319-22997-3. DOI: 10.1007/978-3-319-22997-3_7. URL: https://doi.org/10.1007/978-3-319-22997-3_7.

[JX95] Shi Jin and Zhouping Xin. "The relaxation schemes for systems of conservation laws in arbitrary space dimensions". In: *Communications on pure and applied mathematics* 48.3 (1995), pp. 235–276.

[JZ18] Ting Jiang and XiaoJian Zhou. "Gradient/Hessian-enhanced least square support vector regression". In: *Information Processing Letters* 134 (2018), pp. 1 –8. ISSN: 0020-0190. DOI: https://doi.org/10.1016/j.ipl.2018.01.014. URL: http://www.sciencedirect.com/science/article/pii/S0020019018300292.

[Kay01a] Michael Kay. "Saxon: Anatomy of an XSLT processor". In: *IBM DeveloperWorks* (2001). URL: https://www.ibm.com/developerworks/library/x-xslt2/index.html.

[Kay01b] Michael Kay. *Saxon: the XSLT processor*. 2001. URL: http://sourceforge.net/projects/saxon.

[Kay10] Michael Kay. "A streaming XSLT processor". In: *Balisage: The Markup Conference*. 2010. URL: https://www.balisage.net/Proceedings/vol5/print/Kay01/BalisageVol5-Kay01.html.

[Kay15] Michael Kay. "Parallel Processing in the Saxon XSLT Processor". In: *XML Prague 2015*. 2015. URL: https://www.saxonica.com/papers/xmlprague-2015mhk.pdf.

[Kay20] Michael Kay. *Saxon-JS XSLT Processor*. 2020. URL: https://www.saxonica.com/html/saxon-js/index.html.

[Ked80] Gershon Kedem. "Automatic Differentiation of Computer Programs". In: *ACM Transactions on Mathematical Software* 6.2 (June 1980), pp. 150–165. DOI: 10.1145/355887.355890.

[Kes20] Anne van Kesteren. *DOM*. Tech. rep. Living Standard. whatwg.org, Sept. 2020. URL: https://dom.spec.whatwg.org/.

[KF06] Rahul V. Kharche and Shaun A. Forth. "Source Transformation for MATLAB Automatic Differentiation". In: *Computational Science – ICCS 2006*. Ed. by Vassil N. Alexandrov et al. Vol. 3994. Lecture Notes in Computer Science. Heidelberg: Springer, 2006, pp. 558–565. ISBN: 3-540-34385-7. DOI: 10.1007/11758549_77.

[Kha12] Rahul Vijay Kharche. "Matlab automatic differentiation using source transformation". PhD thesis. Cranfield University, 2012. URL: https://dspace.lib.cranfield.ac.uk/bitstream/handle/1826/7298/KharchePhD2011.pdf.

[Kru16] Fritz Kruger. *CPU Bandwidth – The Worrisome 2020 Trend*. 2016. URL: https://blog.westerndigital.com/cpu-bandwidth-the-worrisome-2020-trend/.

[Kun+12] Julian M. Kunkel et al. "Towards an energy-aware scientific I/O interface". English. In: *Computer Science - Research and Development* 27.4 (2012), pp. 337–345. ISSN: 1865-2034. DOI: 10.1007/s00450-011-0193-x. URL: http://dx.doi.org/10.1007/s00450-011-0193-x.

[LCD11] Komlanvi Lampoh, Isabelle Charpentier, and El Mostafa Daya. "A generic approach for the solution of nonlinear residual equations. Part III: Sensitivity computations". In: *Computer Methods in Applied Mechanics and Engineering* 200.45 (2011), pp. 2983–2990. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2011.06.009. URL: http://www.sciencedirect.com/science/article/pii/S0045782511002271.

[Lee08] David Lee. *XMLSH*. 2008. URL: http://xmlsh.org.

[Leu04] Theodore W. Leung. *Professional XML Development with Apache Tools: Xerces, Xalan, FOP, Cocoon, Axis, Xindice*. John Wiley & Sons, 2004.

[Lev09] John Levine. *Flex & Bison: Text Processing Tools*. O'Reilly Media, Inc., 2009.

[Li+93] Y. Li et al. "Variational Data Assimilation with a Semi-Lagrangian Semi-implicit Global Shallow-Water Equation Model and Its Adjoint". In: *Monthly Weather Review* 121.6 (June 1993), pp. 1759–1769. ISSN: 0027-0644. DOI: 10.1175/1520-0493(1993)121<1759:VDAWAS>2.0.CO;2. URL: https://doi.org/10.1175/1520-0493(1993)121<1759:VDAWAS>2.0.CO;2.

[LK08] Angelika Langer and Klaus Kreft. *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference*. 1st. Boston, MA, USA: Addison-Wesley Professional, 2008. ISBN: 0321585585, 9780321585585.

[LK+13] Christopher M. Lalau-Keraly et al. “Adjoint shape optimization applied to electromagnetic design”. In: *Opt. Express* 21.18 (Sept. 2013), pp. 21693–21701. DOI: 10.1364/OE.21.021693. URL: http://www.opticsexpress.org/abstract.cfm?URI=oe-21-18-21693.

[LM67] J.N. Lyness and C.B. Moler. “Numerical differentiation of analytic functions”. In: *SIAM Journal on Numerical Analysis* 4.2 (1967), pp. 202–210.

[Lof+08] Jay F. Lofstead et al. “Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)”. In: *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. CLADE ’08. Boston, MA, USA: ACM, 2008, pp. 15–24. ISBN: 978-1-60558-156-9. DOI: 10.1145/1383529.1383533. URL: http://doi.acm.org/10.1145/1383529.1383533.

[Lun07] Fredrik Lundh. *ElementTree Overview*. Sept. 2007. URL: http://effbot.org/zone/element-index.htm.

[Mal+10] Grzegorz Malewicz et al. “Pregel: a system for large-scale graph processing”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010, pp. 135–146.

[Mat06] MathWorks. *xslt*. 2006. URL: https://de.mathworks.com/help/matlab/ref/xslt.html.

[Mat13] MathWorks. *fmincon*. 2013. URL: https://de.mathworks.com/help/optim/ug/fmincon.html.

[Mat18] MathWorks. *Vectorization*. 2018. URL: https://www.mathworks.com/help/matlab/matlab_prog/vectorization.html.

[MC05] J.-D. Müller and P. Cusdin. “On the performance of discrete adjoint CFD codes using automatic differentiation”. In: *International Journal for Numerical Methods in Fluids* 47.8-9 (2005), pp. 939–945. ISSN: 1097-0363. DOI: 10.1002/fld.885. URL: http://dx.doi.org/10.1002/fld.885.

[MDN20] MDN contributors. *Element.outerHTML - Web APIs | MDN*. Feb. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/API/Element/outerHTML.

[MDN20a] MDN. *<math> - MathML*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/MathML/Element/math.

[MDN20b] MDN. *<svg> - SVG: Scalable Vector Graphics*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/SVG/Element/svg.

[MHM14] Viktor Mašíček and Irena Holubová (Mlýnková). “XSLTMark II – A Simple, Extensible and Portable XSLT Benchmark”. In: *New Trends in Databases and Information Systems*. Ed. by Barbara Catania et al. Cham: Springer International Publishing, 2014, pp. 113–120. ISBN: 978-3-319-01863-8.

[MMZ18] Priyadarshi Mahapatra, Jinliang Ma, and Stephen E Zitney. “Nonlinear Model Predictive Control Using Decoupled AB Net Formulation for Carbon Capture Systems-Comparison with Algorithmic Differentiation Approach”. In: *2018 Annual American Control Conference (ACC)*. IEEE. 2018, pp. 6439–6444.

[MR96] Michael Monagan and René R. Rodoni. “An Implementation of the Forward and Reverse mode in Maple”. In: *Computational Differentiation: Techniques, Applications, and Tools*. Ed. by Martin Berz et al. Philadelphia, PA: SIAM, 1996, pp. 353–362. ISBN: 0–89871–385–4.

[Mur+05] Makoto Murata et al. “Taxonomy of XML schema languages using formal language theory”. In: *ACM Transactions on Internet Technology (TOIT)* 5.4 (2005), pp. 660–704.

[MW14] Jorge J. Moré and Stefan M. Wild. "Do you trust derivatives or differences?" In: *Journal of Computational Physics* 273 (2014), pp. 268 –277. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2014.04.056. URL: http://www.sciencedirect.com/science/article/pii/S0021999114003325.

[Nad03] Siva Kumaran Nadarajah. "The discrete adjoint approach to aerodynamic shape optimization". PhD thesis. Citeseer, 2003. DOI: 10.2514/6.2001-2530. URL: https://arc.aiaa.org/doi/10.2514/6.2001-2530.

[Nau02] Uwe Naumann. "Reducing the Memory Requirement in Reverse Mode Automatic Differentiation by Solving TBR Flow Equations". In: *Computational Science — ICCS 2002*. Ed. by Peter M. A. Sloot et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 1039–1048. ISBN: 978-3-540-46080-0.

[Nau+04] Uwe Naumann et al. "Control flow reversal for adjoint code generation". In: *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. IEEE. 2004, pp. 55–64.

[Nau+06] Uwe Naumann et al. "Adjoint code by source transformation with OpenAD/F". In: *ECCOMAS CFD 2006: Proceedings of the European Conference on Computational Fluid Dynamics, Egmond aan Zee, The Netherlands, September 5-8, 2006*. Delft University of Technology; European Community on Computational Methods in Applied Sciences (ECCOMAS). 2006.

[Nau08] Uwe Naumann. "Call Tree Reversal is NP-Complete". In: *Advances in Automatic Differentiation*. Ed. by Christian H. Bischof et al. Vol. 64. Lecture Notes in Computational Science and Engineering. Berlin: Springer, 2008, pp. 13–22. ISBN: 978-3-540-68935-5. DOI: 10.1007/978-3-540-68942-3_2.

[Nei10] Richard D. Neidinger. "Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming". In: *SIAM Review* 52.3 (2010), pp. 545–563.

[Nic01] Miloslav Nic. *Graphotron – XPath based generation of graphs from XML documents*. 2001. URL: https://www.oxygenxml.com/archives/xsl-list/200112/msg00784.html.

[NJ01] Siva Nadarajah and Antony Jameson. "Studies of the continuous and discrete adjoint approaches to viscous automatic aerodynamic shape optimization". In: *15th AIAA Computational Fluid Dynamics Conference*. June 2001. DOI: 10.2514/6.2001-2530. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2001-2530.

[NK07] Petr Nalevka and Jirka Kosek. "Advanced approaches to XML document validation". In: *Extreme Markup Languages, Montreal, Québec* (Aug. 2007). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.134.7634&rep=rep1&type=pdf.

[NR06] Uwe Naumann and Jan Riehme. "Computing Adjoints with the NAGWare Fortran 95 Compiler". In: *Automatic Differentiation: Applications, Theory, and Implementations*. Ed. by Martin Bücker et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 159–169. ISBN: 978-3-540-28438-3.

[Obj17] Object Management Group. *Unified Modeling Language*. 2017. URL: https://www.omg.org/spec/UML/.

[oct12] octave.org. *Broadcasting*. 2012. URL: https://octave.org/doc/interpreter/Broadcasting.html.

[Oom17] Jeroen Ooms. *xslt: Extensible Style-Sheet Language Transformations*. 2017. URL: https://CRAN.R-project.org/package=xslt.

[Ope08] OpenMP Architecture Review Board. *OpenMP 3.0 Specifications*. 2008. URL: https://www.openmp.org/wp-content/uploads/spec30.pdf.

[Pas09] Ruud van der Pas. "An overview of OpenMP 3.0". In: *International Workshop on OpenMP*. 2009.

[PBC95] G.D. Pusch, C. Bischof, and A. Carle. "On automatic differentiation of codes with COMPLEX arithmetic with respect to real variables". In: *2* 2.2 (June 1995). DOI: 10.2172/95498.

[Pet07] M. Petera. "Automatic differentiation for portable process systems models". In: *Proceedings in Applied Mathematics and Mechanics* 7.1 (2007), pp. 1140201–1140202. DOI: 10.1002/pamm.200700308.

[Pet12] Monika Petera. "Automatic differentiation of the CapeML high-level language for process engineering". PhD thesis. RWTH Aachen University, 2012. URL: http://darwin.bth.rwth-aachen.de/opus3/volltexte/2012/4044/.

[PF71] Terrence W Pratt and Daniel P Friedman. "A language extension for graph processing and its formal semantics". In: *Communications of the ACM* 14.7 (1971), pp. 460–467.

[PH05] Valérie Pascual and Laurent Hascoët. "Extension of TAPENADE toward Fortran 95". In: *Automatic Differentiation: Applications, Theory, and Implementations*. Ed. by H. M. Bücker et al. Vol. 50. Lecture Notes in Computational Science and Engineering. New York, NY: Springer, 2005, pp. 171–179. DOI: 10.1007/3-540-28438-9_15.

[PH08] Valérie Pascual and Laurent Hascoët. "TAPENADE for C". In: *Advances in Automatic Differentiation*. Ed. by Christian H. Bischof et al. Vol. 64. Lecture Notes in Computational Science and Engineering. Berlin: Springer, 2008, pp. 199–209. ISBN: 978-3-540-68935-5. DOI: 10.1007/978-3-540-68942-3_18.

[POS01] POSIX.1. *IEEE Std 1003.1:2001. Standard for Information Technology – Portable Operating System Interface (POSIX)*. The Institute of Electrical and Electronic Engineers. 2001.

[PQ95] Terence J. Parr and Russell W. Quong. "ANTLR: A predicated-LL(k) parser generator". In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810.

[Pur+95] A. Purakayastha et al. "Characterizing parallel file-access patterns on a large-scale multiprocessor". In: *Parallel Processing Symposium, 1995. Proceedings., 9th International*. Santa Barbara, California, USA, 1995, pp. 165–172. DOI: 10.1109/IPPS.1995.395928.

[PWR13] Michael A Patterson, Matthew Weinstein, and Anil V Rao. "An efficient overloaded method for computing derivatives of mathematical functions in MATLAB". In: *ACM Transactions on Mathematical Software (TOMS)* 39.3 (2013), pp. 1–36.

[PZG05] DI Papadimitriou, AS Zymaris, and KC Giannakoglou. "Discrete and continuous adjoint formulations for turbomachinery applications". In: *EUROGEN 2005, Munich* (2005), pp. 12–14.

[Rac+18] Christopher Rackauckas et al. "A Comparison of Automatic Differentiation and Continuous Sensitivity Analysis for Derivatives of Differential Equation Solutions". In: *CoRR* abs/1812.01892 (2018). arXiv: 1812.01892. URL: http://arxiv.org/abs/1812.01892.

[Rag20] Dave Raggett. *HTML Tidy*. 2020. URL: http://www.htacg.org/.

[Ral81] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*. Vol. 120. Lecture Notes in Computer Science. Berlin: Springer, 1981. ISBN: 0–540–10861–0. DOI: 10.1007/3-540-10861-0.

[RB20] M. Ali Rostami and H. Martin Bücker. "Preconditioning Jacobian Systems by Superimposing Diagonal Blocks". In: *Computational Science – ICCS 2020*. Ed. by Valeria V. Krzhizhanovskaya et al. Cham: Springer International Publishing, 2020, pp. 101–115. ISBN: 978-3-030-50417-5.

[Reu+96] J. Reuther et al. “Aerodynamic shape optimization of complex aircraft configurations via an adjoint formulation”. In: *34th Aerospace Sciences Meeting and Exhibit*. Jan. 1996. DOI: 10.2514/6.1996-94. URL: https://arc.aiaa.org/doi/abs/10.2514/6.1996-94.

[SA09] Victor Travassos Sarinho and Antônio Lopes Apolinário. “A generative programming approach for game development”. In: *Games and Digital Entertainment (SBGAMES), 2009 VIII Brazilian Symposium on*. IEEE. 2009, pp. 83–92.

[Sav+12] R. Savitha et al. “A fully complex-valued radial basis function classifier for real-valued classification problems”. In: *Neurocomputing* 78.1 (2012). Selected papers from the 8th International Symposium on Neural Networks (ISNN 2011), pp. 104 –110. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2011.05.036. URL: http://www.sciencedirect.com/science/article/pii/S0925231211004838.

[SBL12] Laurent Sorber, Marc Van Barel, and Lieven De Lathauwer. “Unconstrained optimization of real functions in complex variables”. In: *SIAM Journal on Optimization* 22.3 (July 2012), pp. 879–898. URL: https://lirias.kuleuven.be/bitstream/123456789/342824/2/complex_optimization.pdf.

[SBL13] Laurent Sorber, Marc Van Barel, and Lieven De Lathauwer. *Complex Optimization Toolbox v1.0*. Tech. rep. KU Leuven, Feb. 2013. URL: http://esat.kuleuven.be/sista/cot/.

[SD10] Raymond J Spiteri and Ryan C Dean. “Stiffness analysis of cardiac electrophysiological models”. In: *Annals of biomedical engineering* 38.12 (2010), pp. 3592–3604.

[Seu+12] S. Seuren et al. “Sensitivity Analysis of a Force and Microstructure Model for Plate Rolling”. In: *Proceedings of the 14th International Conference on Metal Forming (Metal Forming 2012)*. Ed. by J. Kusiak, J. Majta, and D. Szeliga. AGH University of Science and Technology. Krakow, Poland: Wiley-VCH, Sept. 2012, pp. 91–94.

[Seu+13] S. Seuren et al. “Analyse von Sensitivitäten bei der Modellierung des Grobblechwalzens”. In: *28. Aachener Stahl Kolloquium Umformtechnik (ASK 2013), Tagungsband*. Ed. by G. Hirt. RWTH Aachen University. Aachen, Germany: Verlag Mainz, Mar. 2013, pp. 111–121.

[SF16] Soroush Saadatfar and David Filip. “Best Practice for DSDL-based Validation”. In: *XML London 2016 Conference Proceedings. Presented at the XML London*. 2016.

[SH02] Frank Schmuck and Roger Haskin. “GPFS: A shared-disk file system for large computing clusters”. In: *Proceedings of the First USENIX Conference on File and Storage Technologies*. Monterey, California, USA, 2002, pp. 231–244.

[SKF18] Filip Srajer, Zuzana Kukelova, and Andrew Fitzgibbon. “A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning”. In: *Optimization Methods and Software* 33.4-6 (2018), pp. 889–906. DOI: 10.1080/10556788.2018.1435651. eprint: https://doi.org/10.1080/10556788.2018.1435651. URL: https://doi.org/10.1080/10556788.2018.1435651.

[Sta15] Richard Stallman. *GNU Emacs*. 2015. URL: https://www.gnu.org/software/emacs/.

[SW13] Semih Salihoglu and Jennifer Widom. “GPS: A graph processing system”. In: *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. 2013, pp. 1–12.

[TFP03] Mohamed Tadjouddine, Shaun A. Forth, and John D. Pryce. “Hierarchical Automatic Differentiation by Vertex Elimination and Source Transformation”. In: *Computational Science and Its Applications – ICCSA 2003, Proceedings of the International Conference on Computational Science and its Applications, Montreal, Canada, May 18–21, 2003. Part II*. Ed. by V. Kumar et al. Vol. 2668. Lecture Notes in Computer Science. Berlin: Springer, 2003, pp. 115–124.

[TGL02] Rajeev Thakur, William Gropp, and Ewing Lusk. "Optimizing noncontiguous accesses in MPI–IO". In: *Parallel Computing* 28.1 (2002), pp. 83–105.

[The11] The Apache Software Foundation. *Apache Xalan*. 2011. URL: https://xalan.apache.org/.

[The13] The Apache Software Foundation. *Cocoon Main Site*. 2013. URL: https://cocoon.apache.org/.

[The20] The Apache Software Foundation. *Apache Batik SVG Toolkit*. May 2020. URL: https://xmlgraphics.apache.org/batik/.

[The99] The Apache Software Foundation. *Xalan-C++ version 1.10*. 1999. URL: https://xalan.apache.org/old/xalan-c/index.html.

[Tho+04] Henry S Thompson et al. "XML schema part 1: structures second edition". In: *W3C Recommendation* (2004). URL: https://www.w3.org/TR/xmlschema-1/.

[Tho15] Oliver Thoma. "Optimale Steuerung der relativistischen Maxwell-Newton-Lorentz Gleichungen". PhD thesis. Technische Universität Dortmund, 2015. URL: https://eldorado.tu-dortmund.de/bitstream/2003/34349/1/Dissertation.pdf.

[TKC13] C. Terboven, P. Kapinos, and T. Cramer. *Personal communication*. 2013.

[TLG04] R. Thakur, E. Lusk, and W. Gropp. *Users guide for ROMIO: A high-performance, portable MPI-IO implementation*. Tech. rep. ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, 2004.

[Tro20] Ulya Trofimovich. "RE2C: A lexer generator based on lookahead-TDFA". In: *Software Impacts* 6 (2020), p. 100027. ISSN: 2665-9638. DOI: https://doi.org/10.1016/j.simpa.2020.100027. URL: http://www.sciencedirect.com/science/article/pii/S266596382030018X.

[Utk+08a] Jean Utke et al. "OpenAD/F: A Modular Open-Source Tool for Automatic Differentiation of Fortran Codes". In: *ACM Trans. Math. Softw.* 34.4 (July 2008). ISSN: 0098-3500. DOI: 10.1145/1377596.1377598. URL: https://doi.org/10.1145/1377596.1377598.

[Utk+08b] Jean Utke et al. "OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes". In: *ACM Transactions on Mathematical Software (TOMS)* 34.4 (2008), pp. 1–36.

[Veh09] André Vehreschild. "Automatisches Differenzieren für MATLAB". PhD thesis. RWTH Aachen University, 2009. URL: http://darwin.bth.rwth-aachen.de/opus3/volltexte/2009/2680/.

[Vei03] Daniel Veillard. *Libxslt–The XSLT C library for Gnome*. 2003. URL: http://xmlsoft.org/libxslt.

[Vei04] Daniel Veillard. *Libxml2–The XML C parser and toolkit of Gnome*. 2004. URL: http://www.xmlsoft.org.

[VW13] André Vehreschild and Johannes Willkomm. *The ADiMat Handbook*. May 2013. URL: http://adimat.sc.informatik.tu-darmstadt.de/doc/.

[W3C02] W3C. *An XHTML + MathML + SVG Profile*. Aug. 2002. URL: https://www.w3.org/TR/XHTMLplusMathMLplusSVG/.

[W3C04] W3C. *Document Object Model (DOM) Level 3 Core Specification*. Apr. 2004. URL: https://www.w3.org/TR/DOM-Level-3-Core/.

[W3C09] W3C, XML Core Working Group. *Namespaces in XML 1.0*. Dec. 2009. URL: https://www.w3.org/TR/REC-xml-names/.

[W3C20] W3C. *DOM Parsing and Serialization*. Apr. 2020. URL: https://w3c.github.io/DOM-Parsing/.

[Wal99] Andrea Walther. "Program Reversal Schedules for Single- and Multi-processor Machines". PhD thesis. Germany: Institute of Scientific Computing, Technical University Dresden, 1999.

[WB10] J. Willkomm and H. M. Bücker. *Adjoint transformation of binary MATLAB operators*. Preprint of the Institute for Scientific Computing RWTH–CS–SC–10–04. Aachen: RWTH Aachen University, 2010.

[WBB12] J. Willkomm, C. H. Bischof, and H. M. Bücker. "The impact of dynamic data reshaping on adjoint code generation for weakly-typed languages such as Matlab". In: *Recent Advances in Automatic Differentiation*. Ed. by Shaun Forth et al. Vol. 87. Lecture Notes in Computational Science and Engineering. Berlin: Springer, 2012, pp. 127–138. ISBN: 978-3-642-30022-6. DOI: 10.1007/978-3-642-30023-3.

[WBB14] Johannes Willkomm, Christian H. Bischof, and H. Martin Bücker. "A new user interface for ADiMat: toward accurate and efficient derivatives of MATLAB programmes with ease of use". In: *International Journal of Computational Science and Engineering* 9.5-6 (2014), pp. 408–415. DOI: 10.1504/IJCSE.2014.064526. URL: https://www.inderscienceonline.com/doi/abs/10.1504/IJCSE.2014.064526.

[WBMB15] Johannes Willkomm, Christian Bischof, and H. Martin Bücker. "RIOS: efficient I/O in reverse direction". In: *Software: Practice and Experience* 45.10 (2015), pp. 1399–1427. ISSN: 1097-024X. DOI: 10.1002/spe.2252. URL: http://dx.doi.org/10.1002/spe.2252.

[WHO15] Hadley Wickham, Jim Hester, and Jeroen Ooms. *xml2: Parse XML*. 2015. URL: https://CRAN.R-project.org/package=xml2.

[Wie+13] MarcC. Wiedemann et al. "Towards I/O analysis of HPC systems and a generic architecture to collect access patterns". English. In: *Computer Science - Research and Development* 28.2-3 (2013), pp. 241–251. ISSN: 1865-2034. DOI: 10.1007/s00450-012-0221-5. URL: http://dx.doi.org/10.1007/s00450-012-0221-5.

[Wik19a] Wikipedia. *Cauchy–Riemann equations*. 2019. URL: https://en.wikipedia.org/wiki/Cauchy-Riemann_equations.

[Wik19b] Wikipedia. *Well-formed document*. Sept. 2019. URL: https://en.wikipedia.org/wiki/Well-formed_document.

[Wik19c] Wikipedia. *Wirtinger derivatives*. 2019. URL: https://en.wikipedia.org/wiki/Wirtinger_derivatives.

[Wik20a] Wikipedia. *Apache Airflow*. July 2020. URL: https://en.wikipedia.org/wiki/Apache_Airflow.

[Wik20b] Wikipedia. *DevOps*. July 2020. URL: https://en.wikipedia.org/wiki/DevOps.

[Wik20c] Wikipedia. *Identity transform*. May 2020. URL: https://en.wikipedia.org/wiki/Identity_transform.

[Wik20d] Wikipedia. *Infrastructure as code*. July 2020. URL: https://en.wikipedia.org/wiki/Infrastructure_as_code.

[Wik20e] Wikipedia. *Numerical differentiation*. July 2020. URL: https://en.wikipedia.org/wiki/Numerical_differentiation.

[Wik20f] Wikipedia. *Software as a service*. July 2020. URL: https://en.wikipedia.org/wiki/Software_as_a_service.

[Wik20g] Wikipedia. *XML pipeline*. May 2020. URL: https://en.wikipedia.org/wiki/XML_pipeline.

[Wik20h] Wikipedia. *XSLT*. July 2020. URL: https://en.wikipedia.org/wiki/XSLT.

[Wil10] Johannes Willkomm. *Generating adjoint expressions for Matlab*. 2010. URL: https://tuprints.ulb.tu-darmstadt.de/3428/1/willkomm-euroad10.pdf.

[Wil13a] Johannes Willkomm. *Computing Second Order Derivatives with ADiMat*. Tech. rep. TU Darmstadt, 2013. URL: https://tuprints.ulb.tu-darmstadt.de/3432/1/presentation-iwr-willkomm-2013.pdf.

[Wil13b] Johannes Willkomm. *P2X – Universal parser with XML output*. 2013. URL: http://github.org/rainac/p2x/.

[Wil13c] Johannes Willkomm. *Reverse mode IO stream*. 2013. URL: https://ourproject.org/projects/rios.

[Wil18] Johannes Willkomm. "Automatic differentiation of ODE integration". In: *CoRR* abs/1802.02247 (2018). arXiv: 1802.02247. URL: http://arxiv.org/abs/1802.02247.

[Wil20a] Johannes Willkomm. *R2X – R to XML Bridge*. 2020. URL: http://github.org/rainac/r2x/.

[Wil20b] Johannes Willkomm. *R/ADR Transformation Server*. 2020. URL: http://r-adr.de/.

[Wil20c] Johannes Willkomm. *XC – Electronic document and workflow system*. 2020. URL: https://github.com/aiandit/xc.

[Wil20d] Johannes Willkomm. *XHLP – XML Hierarchical Linear Pipelines*. 2020. URL: https://github.com/aiandit/xhlp.

[Wir27] W. Wirtinger. "Zur formalen Theorie der Funktionen von mehr komplexen Veränderlichen". In: *Mathematische Annalen* 97 (1927), pp. 357–375. URL: https://doi.org/10.1007/BF01447872.

[WMT10] Norman Walsh, Alex Milowski, and Henry S. Thompson. *XProc: An XML Pipeline Language*. Tech. rep. World Wide Web Consortium (W3C), May 2010. URL: http://www.w3.org/TR/xproc/.

[WR17] Matthew J. Weinstein and Anil V. Rao. "Algorithm 984: ADiGator, a Toolbox for the Algorithmic Differentiation of Mathematical Functions in MATLAB Using Source Transformation via Operator Overloading". In: *ACM Trans. Math. Softw.* 44.2 (Aug. 2017). ISSN: 0098-3500. DOI: 10.1145/3104990. URL: https://doi.org/10.1145/3104990.

[Zho96] Wu Zhongde. "A Thermoelastic Hydrodynamics Analysis of EMP-segments of Thrust Bearing". In: *LARGE ELECTRIC MACHINE AND HYDRAULIC TURBINE* 5 (1996).

[ZM16] H. Zhang and D. P. Mandic. "Is a Complex-Valued Stepsize Advantageous in Complex-Valued Gradient Learning Algorithms?" In: *IEEE Transactions on Neural Networks and Learning Systems* 27.12 (Dec. 2016), pp. 2730–2735. DOI: 10.1109/TNNLS.2015.2494361.