Application of Algorithmic Differentiation for Exact Jacobians to the Universal Laminar Flame Solver

Alexander Hück[1 (✉)], Sebastian Kreutzer[1], Danny Messig[2], Arne Scholtissek[2], Christian Bischof[1], and Christian Hasse[2]

[1] Institute of Scientific Computing,
Technische Universität Darmstadt, Darmstadt, Germany,
{alexander.hueck@sc., sebastian.kreutzer@stud.,
christian.bischof@sc.}tu-darmstadt.de
[2] Institute of Simulation of reactive Thermo-Fluid Systems,
Technische Universität Darmstadt, Darmstadt, Germany,
{messig, scholtissek, hasse}@stfs.tu-darmstadt.de

Abstract. We introduce algorithmic differentiation (AD) to the C++ Universal Laminar Flame (ULF) solver code. ULF is used for solving generic laminar flame configurations in the field of combustion engineering. We describe in detail the required code changes based on the operator overloading-based AD tool CoDiPack. In particular, we introduce a global alias for the scalar type in ULF and generic data structure using templates. To interface with external solvers, template-based functions which handle data conversion and type casts through specialization for the AD type are introduced. The differentiated ULF code is numerically verified and performance is measured by solving two canonical models in the field of chemically reacting flows, a homogeneous reactor and a freely propagating flame. The models stiff set of equations is solved with Newtons method. The required Jacobians, calculated with AD, are compared with the existing finite differences (FD) implementation. We observe improvements of AD over FD. The resulting code is more modular, can easily be adapted to new chemistry and transport models, and enables future sensitivity studies for arbitrary model parameters.

Keywords: combustion engineering · flamelet simulation · algorithmic differentiation · exact Jacobians · Newton method · C++

1 Introduction

The simulation of realistic combustion phenomena quickly becomes computationally expensive due to, e.g., inherent multi-scale characteristics, complex fluid flow, or detailed chemistry with many chemical species and reactions. Simulation codes for chemically reacting flows (e.g., turbulent flames) solve stiff systems of partial differential equations (PDE) on large grids, making the use of efficient computational strategies necessary. Here, algorithmic differentiation (AD, [3])

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_43

can help to resolve some of the computational challenges associated with chemically reacting flows. AD enables, for instance, sensitivity studies [1], or efficient optimization algorithms for key parameters [11] of these mechanisms.

In this work, we apply AD to the Universal Laminar Flame (ULF) solver [13], a C++ framework for solving generic laminar flame configurations. The code computes and tabulates the thermochemical state of these generic flames which is then parametrized by few control variables, i.e., the mixture fraction and reaction progress variable. The look-up tables obtained with ULF are used in 3D CFD simulations of chemically reacting flows, together with common chemical reduction techniques such as the flamelet concept [10]. The code is also used in other scenarios, e.g., for detailed flame structure analyses or model development, and is a key tool for studying the characteristics of chemically reacting flows.

The objective of this study is to introduce AD to ULF by using operator overloading. We focus on the technical details of this modification and how it impacts overall code performance. As an exemplary application, AD is used to compute exact Jacobians which are required by the numerical solver. The derivatives generated by AD are accurate up to machine precision, often at a lower computational cost w.r.t. finite differentiation (FD).

2 Two Canonical Models from the ULF framework

We study the impact of AD using two canonical problems in the field of chemically reacting flows, i.e., a constant-pressure, homogeneous reactor (HR) and a freely-propagating premixed flame (FPF). The FPF is a 1D, stationary flame which burns towards a premixed stream of fresh reactants (e.g., a methane-air mixture), such that the fluid flow velocity and the burning velocity of the flame compensate each other. This yields a two-point boundary value problem described by a differential-algebraic equation (DAE) set, cf. [7]. We obtain the HR problem if all terms except the transient and the chemical source terms are omitted, resulting in an ordinary differential equation (ODE) set.

The stiff equation set is solved with an implicit time-stepping procedure based on backward-differencing formulas (BDF) with internal Newton iterations, i.e., the solvers BzzDAE [2] and CVODE [4] are used for the FPF and the HR, respectively. The discretization for the FPF is done on a 1D grid with n points. The HR is, on the other hand, solved on a single grid point. Each grid point i has a state X_i that is dependent on the temperature T_i and mass fractions $Y_{i,j}$ for species $j = 1...k$.

The internal Newton iteration of each solver requires a Jacobian. ULF uses a callback mechanism for user-defined Jacobians and computes them either numerically, or analytically for certain mechanisms. The Jacobian of the equation set $f : \mathbb{R}^N \to \mathbb{R}^N$ is typically a block tridiagonal matrix of dimension $\mathbb{R}^{N \times N}$ with $N = n(k + 1)$. The main diagonal is the state X_i (Jacobian of the chemical source terms), the lower and upper diagonal describe the influence of the neighboring grid points, e.g., $i-1$ and $i+1$, determined by the transport terms.

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_43

3 Algorithmic Differentiation

AD [3] describes the semantic augmentation of scientific codes in order to compute derivatives. In the context of AD, any code is assumed to be a set of mathematical operations (e.g., `+`) and functions (e.g., `sin`) which have known analytical derivatives. Thus, the chain rule of differentiation can be applied to each statement, resulting in the propagation of derivatives through the code.

Two modes exist for AD, the forward mode (FM) and the reverse mode (RM). The FM applies the chain rule at each computational stage and propagates the derivatives of intermediate variables w.r.t. input variables along the program flow. The RM, on the other hand, propagates adjoints - derivatives of the final result w.r.t. intermediate variables - in reverse order through the program flow. This requires temporary storage, called *tape*, but computes gradients efficiently.

The semantic augmentation for AD is either done by a source transformation or by operator overloading. Due to the inherent complexity of the C++ language, there is no comprehensive source transformation tool and, thus, operator overloading is typically used in complex C++ simulation software. With operator overloading, the built-in floating point type T is re-declared to a user-defined AD type $\tilde{T}$. It stores the original value of T, called *primal value*, and overloads all required operations to perform additional derivative computations.

4 Introducing the AD type to ULF

Introducing AD to a code typically requires 1. a type change to the AD type, 2. integration with external software packages, and 3. addition of code to initialize (*seed*) the AD type, compute, and extract the derivatives [9]. For brevity, we do not show the seeding routines. The FM seeding is analogous to the perturbation required for the existing FD implementation to compute the Jacobian. The RM, on the other hand, requires *taping* of the function and seeding of the respective function output before evaluating the tape to assemble the Jacobian.

We apply the operator overloading AD tool CoDiPack. CoDiPack uses advanced metaprogramming techniques to minimize the overhead and has been successfully used for large scale (CFD) simulations [5].

4.1 Fundamentals of Enabling Operator Overloading in ULF

We define the alias `ulfScalar` in a central header which, in the current design, is either set to the built-in double type or an AD type at compile time, see Fig. 1.

```
using ulfScalar = double | codi::RealForward | codi::RealReverse;
```

Fig. 1. RealForward and RealReverse are the standard CoDiPack AD types for the FM and RM, respectively. The AD types have an API for accessing the derivatives etc.

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_43

The initial design of ULF did not account for user-defined types (e.g., an AD type). This leads to compile time errors after a type change as built-in and user-defined types are treated differently [6]. Hence, we refactored several components in ULF, i.e., eliminating implicit conversions of the AD data types to some other type, handling of explicit type casts and re-designing the ULF data structures.

Generic Data Structures. The fundamental numeric data structure for computations in ULF is the `field` class. It stores, e.g., the temperature and concentration of the different chemical species. In accordance to [9], we refactored and templated this class in order to support generic scalar types, specifying two parameters, 1. the underlying vector representation type, and 2. the corresponding, underlying scalar type, see Fig. 2. We extended this principle to every other data structure in ULF, having the scalar alias as the basic type dependency.

```
1 template <typename Vector, typename Scalar>
2 class fieldTmpl : public fieldDataTmpl<Vector, Scalar> { ... };
3 using field = fieldTmpl<ulfVector, ulfScalar>;
```

Fig. 2. The class `fieldTmpl` represents the base class for fields and provides all operations. They are generically defined in the `fieldDataTmpl`. Finally, the ULF framework defines `field` as an alias for the data structure based on the `ulfScalar` alias.

Special Code Regions: External Libraries and Diagnostics. The external solver libraries and, more generically, external API calls (e.g., assert and printing statements) in ULF, do not need to be differentiated. We denote them as special regions, as they require the built-in double type. ULF, for instance, previously interfaced with external solvers by copying between the internal data representation and the different data structures of the solvers. With AD, this requires an explicit value extraction of the primal value (cf. Sect. 3). To that end, we introduced a conversion function, using template specialization, see Fig. 3. This design reduces the undue code maintenance burden of introducing user-defined types to ULF. If, e.g., a multi-precision type is required, only an additional specialization of the conversion function in a single header is required.

5 Evaluation

We evaluate the ULF AD implementation based on two reaction mechanisms, shown in Table 1. For brevity, we focus on the RM.

Timings are the median of a series of multiple runs for each mechanism. The standard deviation was less than 3 % w.r.t. the median for each benchmark. All computations were conducted on a compute node of the Lichtenberg high-performance computer of TU Darmstadt, with two Intel Xeon Processor E5-2680 v3 at a fixed frequency of 2.5 GHz with 64 GB RAM.

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_43

```
1 namespace detail {
2   template <typename T>
3   struct ForSpecialization { static auto value(const T& v) { return v; } };
4 } /* namespace detail */
5 template <typename T>
6 auto value(const T& v) { return detail::ForSpecialization<T>::value(v); }
7 template <typename To, typename From>
8 auto recast(const From& v) { return static_cast<To>(value<From>(v)); }
```

Fig. 3. The functions `value` and `recast` are used for value extraction and type casting, respectively. The former makes use of the struct in the `detail` namespace which is used to specialize for user-defined types, e.g., the AD type (not shown). If built-in doubles are used, the value is simply returned (as shown).

Table 1. Mechanisms for the model evaluations.

Mechanism	Species	Reactions	Reference
`ch4_USCMech (USCMech)`	111	784	[12]
`c3_aramco (Aramco)`	253	1542	[8]

5.1 Homogeneous Reactor

Fig. 4 shows the absolute timings of the solver phase for FD and the RM, respectively. The substantial speedups observed for the RM mainly result from the cheap computation of the Jacobian J. One evaluation of J is 20 to 25 times faster, depending on the mechanism. With FD, each time J is computed, the HR model function F is executed multiple times. In contrast, with the RM, F is executed once to generate a trace on the *tape*. The tape is then evaluated multiple times to generate J, avoiding costly evaluations of F. Memory overheads are negligible for the HR as the RM adds about 5 MB to 7 MB memory use.

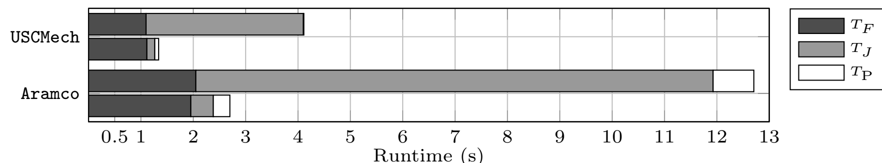

Fig. 4. For each mechanism, the top bar shows FD and the bottom bar shows AD RM measurements. The time of the solve process is divided into three phases: T_F and T_J are the cumulated time for the function and Jacobian evaluations, respectively. T_P is the time spent in CVODE.

5.2 Freely Propagating Flame

The mechanisms are solved on three different mesh resolutions on a 1D domain. As shown in Table 2, with AD, a single evaluation of F has a higher cost

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_43

compared to FD. Here, the overhead ratio between FD and AD appears to be approximately constant for the different mesh configurations of both mechanisms. The evaluation of J is initially cheaper with AD but with rising mesh sizes the speedup vanishes. The total execution time with AD is slightly faster on average as the solver requires fewer steps (i.e., less invocations of F and J) for a solution.

The memory overhead is mostly constant for each mesh configuration as the tape only stores the adjoints of F to generate the Jacobian. We observe an additional main memory usage of about 100 MB to 430 MB and 180 MB to 800 MB memory from the smallest to largest mesh setup for `USCMech` and `Aramco`, respectively. We partly attribute the small variations of memory overhead to the accuracy of the measurement mechanism and due to the tape of CoDiPack which stores the data in chunks of a fixed size. Hence, the total tape memory consumption might be slightly higher than required.

Table 2. Timings for the FPF. F and J are times for a single evaluation, $\#F$ and $\#J$ are the number of invocations of each respective routine. M_{AD}/M_{FD} is the memory ratio. Note: $\#F$ includes the evaluations required for computing J.

Mechanism	Mesh	Mode	Total time (s)	F (ms)	J (s)	$\#F$	$\#J$	$\frac{M_{AD}}{M_{FD}}$
`USCMech`	70	FD	425.009	11.194	3.842	20129	42	-
		AD	409.299	18.410	2.735	5329	42	1.64
	153	FD	1305.891	21.514	7.108	29704	58	-
		AD	1154.853	32.928	6.837	7618	55	1.69
	334	FD	2571.581	43.396	15.208	28946	67	-
		AD	2365.422	66.525	15.070	5167	65	1.7
`Aramco`	70	FD	1654.365	21.090	17.276	25169	25	-
		AD	1509.730	34.418	12.856	5650	24	1.26
	153	FD	5157.127	43.253	34.531	36223	40	-
		AD	4618.236	61.365	26.684	3790	40	1.26
	334	FD	11357.736	84.402	67.547	39806	47	-
		AD	11225.256	126.312	63.163	2822	42	1.27

6 Conclusion and Future Work

We presented the introduction of AD to ULF by using the operator overloading AD tool CoDiPack. In particular, we first introduced a global alias for the basic scalar type in ULF, which is set to the AD type at compile time. All other data structures were rewritten to use templates and are, subsequently, based on this alias. To interface with external APIs, template-based functions for casting and value extraction were introduced, and specialized for the AD types.

The HR model is solved on a single grid point, without transport properties. We observe substantial speedups due to the cheap computation of Jacobians with AD compared to FD. For the FPF, the underlying DAE solver typically requires less steps with AD but the evaluation of the model function is more expensive by a mostly fixed offset compared to FD.

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_43

In the future, experimentation with the ODE/DAE solver parameter settings to exploit the improved precision seems worthwhile. More importantly, the newly gained ability to calculate arbitrary derivatives up to machine precision in ULF enables us to conduct sensitivity studies not only limited to the chemical reaction rates, and model optimization experiments. In particular, advanced combustion studies such as uncertainty quantification, reconstruction of low-dimensional intrinsic manifolds or combustion regime identification become accessible.

References

1. Carmichael, G.R., Sandu, A., et al.: Sensitivity Analysis For Atmospheric Chemistry Models Via Automatic Differentiation. Atmospheric Environment **31**(3), 475–489 (1997)
2. Ferraris, G.B., Manca, D.: Bzzode: a new C++ class for the solution of stiff and non-stiff ordinary differential equation systems. Computers & Chemical Engineering **22**(11), 1595 – 1621 (1998)
3. Griewank, A., Walther, A.: Evaluating Derivatives. Society for Industrial and Applied Mathematics (SIAM), second edn. (2008)
4. Hindmarsh, A.C., Brown, P.N., Grant, K.E., Lee, S.L., Serban, R., Shumaker, D.E., Woodward, C.S.: SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. ACM Transactions on Mathematical Software (TOMS) **31**(3), 363–396 (2005)
5. Hück, A., Bischof, C., Sagebaum, M., Gauger, N.R., Jurgelucks, B., Larour, E., Perez, G.: A Usability Case Study of Algorithmic Differentiation Tools on the ISSM Ice Sheet Model. Optimization Methods and Software pp. 1–24 (2017)
6. Hück, A., Utke, J., Bischof, C.: Source Transformation of C++ Codes for Compatibility with Operator Overloading. Procedia Computer Science **80**, 1485–1496 (2016)
7. Kee, R.J., Grcar, J.F., Smooke, M.D., Miller, J.A., Meeks, E.: PREMIX: A Fortran Program for Modeling Steady Laminar One-Dimensional Premixed Flames. Tech. Rep. SAND85-8249, Sandia National Laboratories (1985)
8. Metcalfe, W.K., Burke, S.M., Ahmed, S.S., Curran, H.J.: A Hierarchical and Comparative Kinetic Modeling Study of C1-C2 Hydrocarbon and Oxygenated Fuels. International Journal of Chemical Kinetics **45**(10), 638–675 (2013)
9. Pawlowski, R.P., Phipps, E.T., Salinger, A.G.: Automating Embedded Analysis Capabilities and Managing Software Complexity in Multiphysics Simulation, Part I: Template-based Generic Programming. Sci. Program. **20**(2), 197–219 (2012)
10. Peters, N.: Laminar flamelet concepts in turbulent combustion. Symp. (Int.) Combust. **21**(1), 1231–1250 (1988)
11. Probst, M., Lülfesmann, M., Nicolai, M., Bücker, H., Behr, M., Bischof, C.: Sensitivity of optimal shapes of artificial grafts with respect to flow parameters. Computer Methods in Applied Mechanics and Engineering **199**(17-20), 997–1005 (2010)
12. Wang, H., You, X., Joshi, A.V., Davis, S.G., Laskin, A., Egolfopoulos, F., Law, C.K.: USC Mech Version II. High-Temperature Combustion Reaction Model of H2/CO/C1-C4 Compounds. online (2007), http://ignis.usc.edu/USC_Mech_II.htm
13. Zschutschke, A., Messig, D., Scholtissek, A., Hasse, C.: Universal Laminar Flame Solver (ULF) (2017), https://figshare.com/articles/ULF_code_pdf/5119855

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_43