Adjoint Code Design Patterns

Uwe Naumann and Jonathan Hüser
Department of Computer Science
RWTH Aachen University
D-52056 Aachen, Germany
[naumann,hueser]@stce.rwth-aachen.de

Abstract

Adjoint methods have become fundamental ingredients of the scientific computing toolbox over the past decades. Large-scale parameter sensitivity analysis, uncertainty quantification and nonlinear optimization as well as modern approaches to deep learning would otherwise turn out computationally infeasible. For nontrivial real-world problems the algorithmic derivation of adjoint numerical simulation programs quickly becomes a highly complex software engineering task requiring expertise in software analysis, transformation and optimization. Despite rather mature software tool support for algorithmic differentiation substantial user intervention is typically required. A large number of patterns shared by numerous application codes results in repeated duplication of development effort. The adjoint code design patterns discussed in this talk based on [8] aim to reduce this problem through improved formalization from the software engineering perspective.

1 Motivation

Motivated by the continuously growing complexity of mathematical models in Computational Science, Engineering and Finance (CSEF) algorithmic adjoint numerical simulations have experienced an increased level of attention over the past years. Applications range from traditional areas such as meteorology / physical oceanography or shape / topology optimization in aerospace and automotive engineering to more recent developments in deep (machine) learning [1] or computational finance. They share the need for large ($> O(10^3)$) gradients to be evaluated for mathematical models implemented as often highly complex numerical simulation software.

The development of adjoint numerical simulations benefits tremendously from work conducted in the field of algorithmic differentiation (AD) [6, 7] over the past decades. Progress in methods and software tools is supported by reports on a vast number of successful applications; see, e.g, [3, 2]. Nevertheless, the implementation of an adjoint, its testing and validation, and maintenance within a rapidly changing hard- and software environment remains an extremely challenging simulation software engineering task. Building on existing adjoint coding expertise and corresponding AD tool support the choice of the right level of abstraction in a modular approach to adjoint code design turns out to be crucial for ensuring efficiency (computation time and memory requirement), scalability (on parallel computers including shared and distributed memory architectures as well as accelerators), robustness (with respect to changes in requirements / the overall software design), flexibility (combination / substitution of adjoint modules) and sustainability (long-term integration into the compute infrastructure) of the software.

In this talk we discuss *adjoint code design patterns* as a (potential standard) approach to the development, documentation, and exchange of (building blocks of) adjoint simulation software. Both terminology and notation are inspired by object-oriented analysis and design methodology. We use UML class diagrams for graphical formalization. Case studies are implemented in the C++ pro-

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

gramming language based on our AD software tool dco/c++. Generalization to other programming paradigms and languages is possible and encouraged.

2 Adjoint Code Design Patterns

Algorithmic adjoint code contains an *augmented primal section* to record for the given sequence of $q = p + m$ *elemental function*[1] evaluations

$$v^i := \varphi^i(v_k)_{k \prec i} \ , \quad i = 1, \ldots, q \ ,$$

all data required for evaluation of the adjoint elementals

$$v^j_{(1)} := v^j_{(1)} + v^i_{(1)} \cdot \frac{d\varphi^i(v^k)_{k \prec i}}{dv^j} \ , \quad j \prec i = q, \ldots, 1 \tag{1}$$

as a sequence of fused multiply-add (FMA) operations within the *adjoint section*. Refer to [6, 7] for further information on algorithmic adjoints.

For notational convenience when introducing adjoint code design patterns we assume all elementals to map from the entire memory space of the program $(v_{1-n}, \ldots, v_q)$ onto itself. A similar approach is taken in [5].

The primal program for computing a multivariate vector function $F : \mathbb{R}^n \to \mathbb{R}^m$ as $\mathbf{y} = F(\mathbf{x})$ yields an elemental decomposition

$$\mathbf{v}^i = \Phi^i(\mathbf{v}^{i-1}), \quad \Phi^i : \mathbb{R}^{n+q} \to \mathbb{R}^{n+q} \text{ for } i = 1, \ldots, q$$

and $\mathbf{v}^0 = (x^0, \ldots, x^{n-1}, 0, \ldots, 0)$, $\mathbf{v}^q = (x^0, \ldots, x^{n-1}, v^1, \ldots, v^p, y^0, \ldots, y^{m-1})$. Consequently, $\mathbf{x} = P_n \cdot \mathbf{v}^0$ and $\mathbf{y} = \mathbf{v}^q \cdot Q_m^T$ for linear operators $P_n = (I_{n\times n}, 0_{n\times q}) \in \mathbb{R}^{n\times(n+q)}$ and $Q_m = (0_{m\times(n+p)}, I_{m\times m}) \in \mathbb{R}^{m\times(n+q)}$ extracting the first n and last m entries of a vector in $\mathbb{R}^{n+q}$, respectively. The identity in $\mathbb{R}^k$ is $I_{k\times k}$ and $0_{k\times l}$ denotes a matrix of all zeros in $\mathbb{R}^{k\times l}$.

The adjoint program evaluates the adjoint elemental decomposition

$$\mathbf{x}_{(1)} := \mathbf{x}_{(1)} + < \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) > \ ,$$

where

$$< \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) > = P_n \cdot \Phi^1_{(1)}(\mathbf{x}, \Phi^2_{(1)}(\mathbf{v}^1, \ldots \Phi^q_{(1)}(\mathbf{v}^{q-1}, \mathbf{v}^q_{(1)}) \ldots))$$

and for given $\mathbf{v}^q_{(1)} = (0, \ldots, 0, y^0_{(1)}, \ldots, y^{m-1}_{(1)})$ assuming availability of adjoint elementals

$$\mathbf{v}^{i-1}_{(1)} = \Phi^i_{(1)}(\mathbf{v}^{i-1}, \mathbf{v}^i_{(1)}) \equiv \nabla\Phi^i(\mathbf{v}^{i-1})^T \cdot \mathbf{v}^i_{(1)} \text{ for } i = q, \ldots, 1.$$

Adjoint elementals can be implemented in various ways including AD by overloading or (manual) source transformation, symbolic differentiation, finite difference approximation, preaccumulation, and AD of smoothed (discontinuous) elementals.

By default the adjoint elemental decomposition is generated homogeneously, typically using AAD by either overloading or source transformation. Special treatment of certain elementals (e.g, Φ^k) may become desirable or even essential, for example, to ensure feasibility of the memory requirement by checkpointing or preaccumulation, to exploit the implicit function theorem, to handle nonsmoothness or even discontinuity, or to integrate parts of the computation running on a different compute platform (e.g, GPU). The resulting gaps in the adjoint context need to be filled by custom versions of $\Phi^k_{(1)}$ yielding $< \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) >$ as

$$P_n \cdot (\nabla\Phi^1(\mathbf{x})^T \ldots \overbrace{\nabla\Phi^k(\mathbf{v}_{k-1})^T \underbrace{(\nabla\Phi^{k+1}(\mathbf{v}^k)^T \ldots (\nabla\Phi^q(\mathbf{v}^{q-1})^T \cdot \mathbf{v}^q_{(1)}) \ldots))}_{\mathbf{v}^k_{(1)}}}^{\mathbf{v}^{k-1}_{(1)}} \ .$$

[1]Elemental functions range from arithmetic operators and intrinsic functions built into programming languages via solution algorithms for implicit functions to arbitrary compact subsections of the primal code.

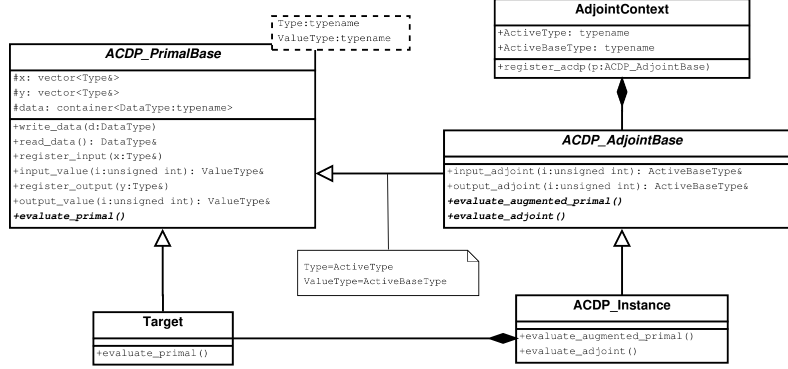

Figure 1: Base Pattern

An API needs to be provided allowing for

$$\mathbf{v}_{(1)}^{k-1} := \Phi_{(1)}^{k}(\mathbf{v}^{k-1}, \mathbf{v}_{(1)}^{k}) \equiv \nabla \Phi^{k}(\mathbf{v}_{k-1})^{T} \cdot \mathbf{v}_{(1)}^{k}$$

to be evaluated based on custom required data to be recorded by an appropriately augmented primal version of Φ^k. For example, checkpointing Φ^k requires its input arguments to be stored in order to allow context-free reevaluation. The adjoint $\Phi^k_{(1)}$ restores the argument checkpoint followed by an augmented primal evaluation of Φ^k (e.g, generation of a tape) and propagation of the adjoints (e.g, interpretation of the tape). Moreover, communication with the context needs to be established by enabling access to in- and outputs of Φ^k and to the adjoints of all active arguments.

Figure 1 shows a UML class diagram of the Base pattern, for example, in the context of dco/c++. To qualify as a context for an adjoint code design pattern (ACDP) the two abstract base patterns ACDP_PrimalBase and ACDP_AdjointBase need to be provided. Details of their implementation depend on the given context and should be irrelevant to the user of an instance of an ACDP. Instances of ACDP contain data (e.g, the solution of a system of nonlinear equations, see Sec. 3) and (optionally) user target code (e.g, the residual of the nonlinear system). The type-generic class ACDP_PrimalBase enables inclusion of the target code into the primal active flow of data and collection (write_data) and use (read_data) of additional pattern-specific and type-generic data. Registration of active in- and outputs yields storage of corresponding references in x and y, respectively. Their primal values are accessed via a unique index using the methods input_value and output_value. For a user target to specialize ACDP_PrimalBase it must implement the method evaluate_primal to evaluate the type-generic primal outputs y as a function of the inputs x. Further details will be discussed during the talk.

3 Case Study

Implicit functions defined as systems of nonlinear equations $F(\mathbf{x}(\mathbf{p}), \mathbf{p}) = 0$, where $\mathbf{x} \in \mathbb{R}^{n_x}$ and $F : \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_x}$ are fundamental numerical kernels in modern scientific computing. The mathematics of adjoint nonlinear systems is covered extensively in the literature, for example, [4]. Computation of $\mathbf{p}_{(1)}$ amounts to the solution of the linear system

$$\frac{\partial F(\mathbf{x}^*, \mathbf{p})}{\partial \mathbf{x}}^{T} \cdot \mathbf{z} = -\mathbf{x}_{(1)} \tag{2}$$

followed a single evaluation of the adjoint residual. The algorithmic adjoint differentiation of the nonlinear solver can be avoided. A (passive) primal solution is followed by the accumulation of the Jacobian of the residual with respect to the state $\mathbf{x}$ at the primal solution. A linear system is solved to obtain $\mathbf{z}$ and hence $\mathbf{p}_{(1)}$ as the adjoint of the residual with respect to $\mathbf{p}$ at the primal solution in direction $\mathbf{z}$. Derivatives of the residual can often be obtained at much lower computational cost than an algorithmic adjoint of the nonlinear solver.

Figure 2 formalizes the above as an adjoint code design pattern. Its detailed discussion will be part of the talk.

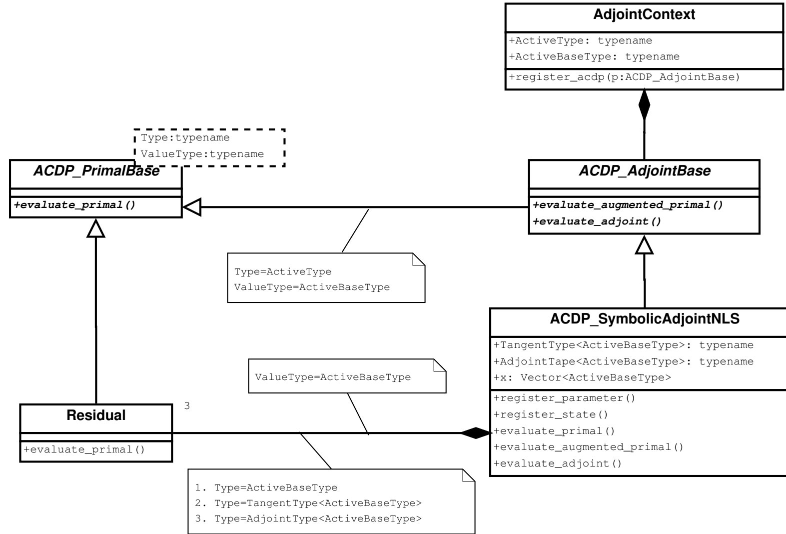

Figure 2: Symbolic Adjoint Nonlinear System Pattern

4 Conclusion

This talk makes an attempt to formalize the design of adjoints for a number of relevant structural and numerical software patterns. Relevance for machine learning results immediately from the growing need for (adjoint) gradients of general models as opposed to special types of (neural) networks. Ad hoc development of adjoint simulations suffers from a high degree of repetition in terms of conceptual results and their implementation. The provision of well-designed and properly documented design patterns is expected to improve robustness and compatibility of adjoint software as well as to streamline the development process. We are convinced that ongoing and future development efforts within the machine learning community can draw substantial benefit from the use of adjoint code design patterns. Details will be discussed during the talk.

References

[1] A. Baydin, B. Pearlmutter, and A. Radul. Automatic differentiation in machine learning: a survey. *CoRR*, abs/1502.05767, 2015.

[2] C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors. *Advances in Automatic Differentiation*, number 64 in Lecture Notes in Computational Science and Engineering (LNCSE). Springer, 2008.

[3] S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther, editors. *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2012.

[4] J. C. Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1:13–21, 1992.

[5] A. Griewank and U. Naumann. Accumulating Jacobians as chained sparse matrix products. *Mathematical Programming*, 95(3):555–571, 2003.

[6] A. Griewank and A. Walther. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation, Seocnd Edition*. Number OT105 in Other Titles in Applied Mathematics. SIAM, 2008.

[7] U. Naumann. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation.* Number SE24 in Software, Environments, and Tools. SIAM, 2012.

[8] U. Naumann. Adjoint code design patterns. Technical Report AIB-2017-08, Department of Computer Science, RWTH Aachen University, 2017. Submitted.