Sensitivity Analysis and Design Optimization through Automatic Differentiation

Sanjukta Bhowmick, Paul D. Hovland, Boyana Norris, Michelle Mills Strout, and Jean Utke

Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S Cass Ave, Argonne IL 60439, USA

E-mail: hovland@mcs.anl.gov

Abstract.

Automatic, or algorithmic, differentiation (AD) is a technique for transforming a program or subprogram that can be interpreted as computing a mathematical function, including arbitrarily complex simulation codes, into one that computes the derivatives of that function. We describe the implementation and application of automatic differentiation tools. We highlight recent advances in the combinatorial algorithms and compiler technology that underlie successful implementation of automatic differentiation tools. We discuss applications of automatic differentiation in design optimization and sensitivity analysis. We also describe ongoing research in the design of language-independent source transformation infrastructures and memory management for automatic differentiation algorithms.

1. Introduction

Automatic differentiation (AD) is a family of techniques for computing the derivatives of a function defined by a computer program. The basis for AD is the assumption that the computation of a vector function $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$ is accomplished by a sequence of p elemental operations $v_i = \phi_i(\ldots, v_j, \ldots), i = 1, \ldots, p$ as found in a computer program implementing an evaluation procedure for $\boldsymbol{f}$. Each of the ϕ is differentiable at least in open subdomains. The derivatives of these elemental operations are combined according to the chain rule of differential calculus. The associativity of the chain rule leads to the two major modes of computing derivatives with AD, the so-called forward (tangent-linear) mode and reverse (cotangent-linear or adjoint) mode.

The *forward mode* multiplies derivatives starting with the independent variables and proceeding toward the dependent variables. Because the order of the derivative computation parallels that of the function computation, intermediate function values can be used as they are computed, and the control flow of the derivative computation follows that of the original program. The *reverse mode* multiplies derivatives starting with the dependent variables and proceeding toward the independent variables. In this case, the control flow reverses that of the original program, and therefore intermediate function values and control flow decisions may need to be recorded. This increases the storage requirements of the reverse mode.

Viewed another way, the forward mode is the application of the chain rule to compute

directional derivatives at the level of elemental operations

$$\dot{v}_i = \sum_j \frac{\partial \phi_i}{\partial v_j} \cdot \dot{v}_j \tag{1}$$

for each elemental operation ϕ_i in the sequence. With the Jacobian $\boldsymbol{J}$ of $\boldsymbol{f}$, this yields $\dot{\boldsymbol{y}} = \boldsymbol{J}\dot{\boldsymbol{x}}$. The reverse mode is the application of the chain rule to compute adjoints

$$\overline{v}_j = \sum_i \frac{\partial \phi_i}{\partial v_j} \cdot \overline{v}_i \tag{2}$$

for each argument v_j. Since the $\overline{v}_j$ depend on the $\overline{v}_i$, the operations in the sequence have to be considered in reverse order. Executed for all ϕ, this yields $\overline{\boldsymbol{x}} = \boldsymbol{J}^T\overline{\boldsymbol{y}}$.

In purely mathematical terms the number of operations (temporal complexity) required for computing the Jacobian of $\boldsymbol{f}\colon \mathbb{R}^n \mapsto \mathbb{R}^m$ in forward mode is $\mathcal{O}(n)$, while the computation with reverse mode is $\mathcal{O}(m)$. Thus, the forward mode is appropriate for functions with small numbers of independent variables (or, via a simple transformation, Jacobian-vector products), while the reverse mode is appropriate for functions with small numbers of dependent variables (or, via a simple transformation, transposed-Jacobian-vector products). The target application evaluates a scalar objective ($m = 1$) for the purpose of data assimilation. Consequently, the gradient $\nabla\boldsymbol{f}$ has a computational cost proportional to the cost of evaluating $\boldsymbol{f}$. We can consider the evaluation of a Hessian $\nabla^2\boldsymbol{f}$ as the computation of the Jacobian of $\nabla\boldsymbol{f}\colon \mathbb{R}^n \mapsto \mathbb{R}^n$.

The first obstacle one faces when transitioning from a purely theoretical formulation of complexities to a practical algorithm is the spatial complexity of the reverse mode ($\leftarrow$). The forward mode ($\rightarrow$) can evaluate the $\frac{\partial \phi_i}{\partial v_j}$ along with the computation of $\boldsymbol{f}$ itself. Because of the reversed dependencies in (2), the reverse mode requires storage space for each value v_j from the time it is defined during the computation of $\boldsymbol{f}$ until it is needed to compute $\frac{\partial \phi_i}{\partial v_j}$. This implies a spatial complexity of $\mathcal{O}(p)$. For large-scale applications, this spatial complexity is overwhelming and is typically tackled via hierarchical checkpointing strategies [**?**].

The implementation of robust and effective automatic differentiation tools requires advances in compiler technology, graph algorithms, and automatic differentiation theory. A robust compiler infrastructure is required to support the source-to-source transformation process.

Compared with other methods, automatic differentiation offers a number of advantages:

Performance. The performance of automatic differentiation-generated code usually exceeds that of finite differences and often rivals that of code developed by hand.

Accuracy. Unlike finite difference approximations, derivatives computed via automatic differentiation exhibit no truncation error.

Reduced software costs. automatic differentiation eliminates the time spent developing and debugging derivative code by hand, or experimenting with step sizes for finite difference approximations. Consistency between a model and its derivatives is easily preserved, reducing software maintenance costs.

2. Foundations

Compiler infrastructure: OpenAD and OpenAnalysis

Automatic differentiation tools rely on compiler analyses, including traditional analyses such as alias analysis and side effect analysis, as well as domain-specific analyses such as activity analysis and linearity analysis, to improve the efficiency of the generated derivative code. Accurate analysis can improve performance by a factor of 2 or more.

Graph algorithms: graph coloring and flattening

Graph coloring is used to reduce the cost of computing sparse Jacobians and Hessians. We have developed a new backtracking heuristic for graph coloring. The new heuristic is superior to both the standard greedy algorithm and Culberson iteration. The figure shows the number of colors required for several graphs ranging in size from fifteen thousand to 2.6 million vertices, using the greedy, backtracking (B), or Culberson iterative (I) algorithm with one of several orderings: natural (NT), largest first (LF), smallest last (SL), saturation degree (SD), incidence degree (ID), and depth first (DF).

3. Applications

Design Optimization

Automatic differentiation is used to compute gradients and Hessians for the parallel solution of optimization problems in TAO [**?**]. FIXME: Steal some text from TAO paper... Figure **??** shows a sequence of solutions and their deviation from the optimal solution for a bound constrained minimization problem. The objective is the surface with minimal area that satisfies Dirichlet boundary conditions and is constrained to lie above a solid plate.

Ocean Modeling

As part of an NSF-sponsored collaboration with MIT and Rice, we are developing an automatic differentiation tool for Fortran 90 that is being applied to the MIT General Circulation Model. The figure displays a map of sensitivities of zonal volume transport through the Drake Passage to changes in bottom topography everywhere in a barotropic ocean model. The model is based on a shallow water model, extended to a global configuration at 2x2 degree horizontal resolution with realistic topography.

4. Conclusions and Future Work

Conclusions

Future work

First, we will investigate whether the symmetric construction can produce a theoretically optimal way to compute the Hessian and will explore how to integrate a scarcity and symmetry-preserving preaccumulation for efficient Hessian-vector products. Second, we will explore ways of extending the symmetry-preserving computation to gradient computations with checkpointing. We will develop an algorithm that allows efficient, symmetry-preserving computation of local Hessians and an implementation that extends the savings to the large-scale target application. The efficient computation of Hessians is of interest not only for the geophysical application considered here, but also for optimization, chemical and other engineering disciplines.