BRINGING PDES TO JAX WITH FORWARD AND REVERSE MODES AUTOMATIC DIFFERENTIATION

Ivan Yashchuk
Department of Computer Science, Aalto University
VTT Technical Research Centre of Finland
ivan.yashchuk@{aalto,vtt}.fi

ABSTRACT

Partial differential equations (PDEs) are used to describe a variety of physical phenomena. Often these equations do not have analytical solutions and numerical approximations are used instead. One of the common methods to solve PDEs is the finite element method. Computing derivative information of the solution with respect to the input parameters is important in many tasks in scientific computing. We extend JAX automatic differentiation library with an interface to Firedrake finite element library. High-level symbolic representation of PDEs allows bypassing differentiating through low-level possibly many iterations of the underlying nonlinear solvers. Differentiating through Firedrake solvers is done using tangent-linear and adjoint equations. This enables the efficient composition of finite element solvers with arbitrary differentiable programs. The code is available at github.com/IvanYashchuk/jax-firedrake.

1 INTRODUCTION

There is a growing interest in differentiable physical simulators and including them as components in machine learning systems. Recent work in this field includes either implementing physical simulators from scratch in popular deep learning frameworks (Holl et al., 2020) or developing new software specifically for physical simulations (Hu et al., 2019). In the case of simulators consisting of the partial differential equation (PDE) solving differentiating through all the operations of the PDE solver is not scalable and heavy memory-bound. For PDEs, explicit expressions are available for deriving efficient functions needed for automatic differentiation (AD). In this work, we integrate Firedrake PDE library with JAX AD library. As a result, it is possible to compose Firedrake models with arbitrary programs that JAX can differentiate.

2 BACKGROUND

The variational formulation also known as weak formulation allows finding the solution to problems modeled with PDEs using an integral form. Having the integral form gives simpler equations to solve using linear algebra methods over a vector space of infinite dimension or functional space. In finite element methods (FEM), the variational formulation of the problem is transformed into a system of nonlinear equations for unknown finite element function coefficients that can be solved numerically.

Let a system of PDEs that describe the physics of the problem of interest be written as

$$F(u, m) = 0, \tag{1}$$

where u is the solution and m represents parameters that affect the solution. The solution u can be written as an implicit function $u(m)$ of the parameters m, then we can formulate its derivative $\frac{du}{dm}$. In this work we consider the non-linear systems of PDEs that are discretized using the finite element method.

Let $J(u, m)$ be a functional of interest, it represents any quantity that depends on u and m. Then the problem is computing the derivatives $\frac{dJ(u(m),m)}{dm}$. It can be done using finite difference methods, tangent-linear or adjoint approaches.

The tangent-linear system is the same idea as the forward mode of automatic differentiation, while the adjoint approach corresponds to the reverse mode automatic differentiation. In (Farrell et al., 2013), authors describe the techniques that can be used to derive possibly time-dependant tangent-linear and adjoint models. Having the solution to the tangent-linear system we can evaluate the gradient of any functional. However, in many applications the functional is fixed and the goal is to calculate the derivative with respect to any parameter the chosen functional depends on. In this situation, the better approach is to use the adjoint equations for derivative computation. Higher-order derivatives can be derived by composing tangent-linear and adjoint models (Maddison et al., 2019).

Firedrake library Firedrake is an automated system for the solution of partial differential equations using the finite element method. It uses the UFL language to express variational problems (Rathgeber et al., 2016; Alnaes et al., 2014). Then this high-level problem representation is compiled into low-level C code for assembling vectors and matrices for the nonlinear solver. Nonlinear and linear solvers are based on PETSc library (Balay et al., 2019). UFL representation of the residual equation makes it straightforward to generate both tangent-linear and adjoint equations using built-in automatic differentiation. However, pure UFL implementation of derivative calculation is restricted only to one variational problem. Time-dependent problems are solved as a sequence of variational problems and to differentiate through this sequence program execution should be traced. Two libraries interface directly with Firedrake for automated derivative computation: dolfin-adjoint (Mitusch et al., 2019) and tlm_adjoint (Maddison et al., 2019). So it is not necessary to use dolfin-adjoint and relying only on UFL for adjoint and tangent-linear derivation is possible if the Firedrake model is a single variational problem or time-stepping is implemented in JAX. However, we chose to depend on dolfin-adjoint library here as it also makes possible to calculate shape derivatives for domain optimization and boundary conditions derivatives.

3 AUTOMATICALLY DIFFERENTIATING PDE SOLVERS IN JAX

JAX is a numerical computing library that includes forward and reverse mode automatic differentiation. JAX uses terminology and concepts from differential geometry, namely pushforward map for the forward mode AD and pullback map for the reverse mode AD. These maps can also be described in terms of the Jacobian: The pushforward is Jacobian-vector product (JVP), and pullback is Jacobian-transpose-vector product, or vector-Jacobian product (VJP).

Given a function $f : \mathbb{R}^n \to \mathbb{R}^m$, the Jacobian matrix of f evaluated at an input point $x \in \mathbb{R}^n$, denoted $\partial f(x)$, is often thought of as a matrix of partial derivatives of size $m \times n$. Alternatively, $\partial f(x)$ represents a linear map, which maps the tangent space of the domain of f at the point x to the tangent space of the codomain of f at the point $f(x)$:

$$\partial f(x) : \mathbb{R}^n \to \mathbb{R}^m.$$

This map is called the pushforward map of f at x. Given input point $x \in \mathbb{R}^n$ and a tangent vector $v \in \mathbb{R}^n$, we get back an output tangent vector in $\mathbb{R}^m$. This mapping, from (x, v) pairs to output tangent vectors, is called the Jacobian-vector product, and written as

$$(x, v) \mapsto \partial f(x)v.$$

The pullback map of f at x is

$$\partial f(x)^* : \mathbb{R}^m \to \mathbb{R}^n.$$

The Jacobian-transpose-vector product is then

$$(x, v) \mapsto \partial f(x)^* v.$$

Referring back to the tangent-linear model in the context of PDEs, there Jacobian-vector product corresponds to

$$(m, v) \mapsto \frac{du}{dm}v := \dot{u}_v \quad \in \mathbb{R}^m, \tag{2}$$

where $\dot{u}_v$ is the solution to the following tangent-linear equation with right hand side defined as the derivative of $F(u, m)$ with respect to m in the direction $v \in \mathbb{R}^n$:

$$\frac{\partial F(u, m)}{\partial u}\dot{u}_v = -\frac{\partial F(u, m)}{\partial m} \cdot v. \tag{3}$$

Note that compared to the standard tangent-linear equations the unknown in the equation 3 is not the full solution Jacobian matrix but its multiplication with a vector, making the problem easier to solve.

Jacobian-transpose-vector product function is defined as

$$(m, v) \mapsto \frac{du}{dm}^* v := -\frac{\partial F(u,m)}{\partial m}^* \cdot \lambda_v \quad \in \mathbb{R}^n,$$

where λ_v is the solution to the following adjoint equation with right hand side given by the cotangent vector $v \in \mathbb{R}^m$:

$$\frac{\partial F(u,m)}{\partial u}^* \lambda_v = v. \tag{4}$$

Compared to the standard adjoint equations here we do not need any information about the functional and it is AD's system responsibility to compose VJPs for reverse pass and supply cotangent vectors.

Internally, JAX functions are called Primitives and for each Primitive Jacobian-vector products (JVP), vector-Jacobian products (VJP), batching and just-in-time compilation rules are implemented. To make JAX work with external functions we need to implement new Primitives together with functions that define vector-Jacobian and Jacobian-vector products.

We have implemented a new Primitive function for the Firedrake solver that transforms inputs and outputs to appropriate data types and registers associated JVP and VJP functions that solve tangent-linear and adjoint equations respectively.

4 Numerical examples

For demonstrating the implementation we consider the Poisson equation as the forward model problem. Let $\Omega \subset \mathbb{R}^n, n \in \{1, 2, 3\}$ be an open, bounded domain and consider the following problem:

$$\begin{aligned} -\nabla \cdot (\kappa \nabla u) &= f \text{ in } \Omega, \\ u &= 0 \text{ on } \partial\Omega, \end{aligned} \tag{5}$$

where $u : \Omega \to \mathbb{R}$ is the unknown temperature, $\kappa \in \mathbb{R}$ is the thermal conductivity, $f : \Omega \to \mathbb{R}$ is the source term ($f(x) > 0$ corresponds to heating and $f(x) < 0$ corresponds to cooling).

The variational form of the equation 5: Find $u \in H_0^1(\Omega)$ such that

$$(\kappa \nabla u, \nabla v)_{L^2(\Omega)} - (f, v)_{L^2(\Omega)} = 0, \text{ for all } v \in H_0^1(\Omega), \tag{6}$$

where $H_0^1(\Omega)$ is the space of functions vanishing on $\partial\Omega$ with square integrable derivatives. $(\cdot\,, \cdot)_{L^2(\Omega)}$ denotes the L^2-inner product.

Optimal control of the Poisson equation We solve the standard problem in PDE-constrained optimization: the optimal control of the Poisson equation. Physically, this problem can be interpreted as finding the best heating or cooling of a surface to achieve a desired temperature profile. The problem is to minimize the following functional

$$\min_f J(f) := \frac{1}{2} \int_\Omega (u - u_d)^2 \, dx + \frac{\gamma}{2} \int_\Omega f^2 \, ds, \tag{7}$$

where u is the solution to the Poisson equation 6, u_d is the desired temperature profile, f is the unknown control function, γ is the regularization parameter. Additionally, f is constrained to bounds a, b such that $a \le f \le b$.

The unknown function f is discretized in a finite element space such that values of f at each cell of the mesh are the optimization parameters. As the mesh is refined the number of parameters in the optimization problem increases.

For our example we take the desired temperature to be $u_d = \frac{1}{2\pi^2} \frac{1}{1+4\gamma\pi^4} \sin(\pi x) \sin(\pi y)$, $\gamma = 10^{-6}$, and f is bounded between 0 and 0.8. L-BFGS-B optimizer from SciPy library (Virtanen et al., 2020) is used with the gradient values calculated by JAX. Convergence is achieved after 38 iterations with gradient norm tolerance 10^{-10}.

	# parameters	# iterations	$\|\kappa_{opt} - \kappa_{true}\|$	$\|u_{opt} - u_{true}\|$
FEM	981	31	1.356112e-1	2.711260e-4
NN	47	89	5.362389e-2	1.999892e-4

Table 1: Performance comparison between FEM and NN representation of the coefficient.

Coefficient field inversion with neural network representation Representing the model parameters at each point in space quickly leads to a large number of model parameters. The neural network can be used as an approximation to the spatially varying coefficients characterized by the weights of the neural network. The optimization problem is then posed in the space of network-parameters rather than at each cell of the computational grid. In the task of topology optimization, it was shown that neural network representation of the solution to the optimization problem helps to find a better design in many cases (Hoyer et al., 2019). In (Berg & Nyström, 2017), the authors studied neural network representation stability for inverse problems.

In this example, we demonstrate the use of neural nets to parameterize inputs of Firedrake solver and demonstrate the differentiability of the pipeline. The problem is to minimize the following functional

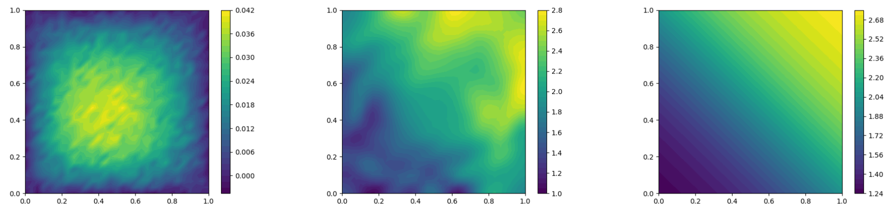
$$\min_{\kappa} J(\kappa) := \frac{1}{2}\int_{\Omega}(u - u_m)^2\,dx + \frac{\gamma}{2}\int_{\Omega}|\nabla\kappa|^2\,dx, \tag{8}$$

where u is the solution to the Poisson equation 6, u_m is the noisy temperature measurement, κ is the unknown material coefficient field, γ is the regularization parameter.

Here, for the simplicity, we choose feed-forward neural network with single hidden layer with 10 neurons and tanh activation function. We set up synthetic measurements temperature data with spatially varying coefficient $\kappa = 1 + x + y$ and adding small noise to the true temperature state. L-BFGS-B optimizer is used with gradient norm tolerance 10^{-6}. Number of iterations and L^2-norm of the difference between the optimal solution and the true one are summarized in Table 1. In this problem, finite element representation overfits to the noise present in the measurements while neural network representation can recover material coefficient function closer to the true function and visually seems to be unaffected by the noise (Figure 1).

5 Conclusions

We describe an extension to JAX that allows a seamless inclusion of PDE solvers written in Firedrake into arbitrary JAX differentiable programs, including, but not limited to, deep learning models, Bayesian probabilistic models. Ongoing work targets further integration of Firedrake code with JAX allowing just-in-time compilation and GPU computing and better interoperability of JAX parallelism with Firedrake's MPI parallelism.



Noisy temperature field Inverted κ, FEM Inverted κ, NN

Figure 1: Conductivity field inversion from noisy measurements using FEM and NN representations of optimization parameters.

REFERENCES

Martin S. Alnaes, Anders Logg, Kristian B. Oelgaard, Marie E. Rognes, and Garth N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw.*, 40(2), March 2014. ISSN 0098-3500. doi: 10.1145/2566630. URL `https://doi.org/10.1145/2566630`.

Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dmitry Karpeyev, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc Web page. `https://www.mcs.anl.gov/petsc`, 2019. URL `https://www.mcs.anl.gov/petsc`.

Jens Berg and Kaj Nyström. Neural network augmented inverse problems for pdes, 2017.

Patrick E. Farrell, David A. Ham, Simon W. Funke, and Marie E. Rognes. Automated derivation of the adjoint of high-level transient finite element programs. *SIAM Journal on Scientific Computing*, 35(4):C369–C393, January 2013. doi: 10.1137/120873558. URL `https://doi.org/10.1137/120873558`.

Philipp Holl, Vladlen Koltun, and Nils Thuerey. Learning to control pdes with differentiable physics, 2020.

Stephan Hoyer, Jascha Sohl-Dickstein, and Sam Greydanus. Neural reparameterization improves structural optimization, 2019.

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. Difftaichi: Differentiable programming for physical simulation, 2019.

James R. Maddison, Daniel N. Goldberg, and Benjamin D. Goddard. Automated calculation of higher order partial differential equation constrained derivative information. *SIAM Journal on Scientific Computing*, 41(5):C417–C445, January 2019. doi: 10.1137/18m1209465. URL `https://doi.org/10.1137/18m1209465`.

Sebastian Mitusch, Simon Funke, and Jørgen Dokken. dolfin-adjoint 2018.1: automated adjoints for FEniCS and firedrake. *Journal of Open Source Software*, 4(38):1292, June 2019. doi: 10.21105/joss.01292. URL `https://doi.org/10.21105/joss.01292`.

Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake. *ACM Transactions on Mathematical Software*, 43(3):1–27, December 2016. doi: 10.1145/2998441. URL `https://doi.org/10.1145/2998441`.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 2020. doi: https://doi.org/10.1038/s41592-019-0686-2.

A Derivations of tangent-linear and adjoint equations

Begin with the non-linear system of equations given by equation 1. Then, taking the total derivative of the above equation with respect to the parameter m gives

$$\frac{d}{dm}F(u,m) = \frac{\partial F(u,m)}{\partial u}\frac{du}{dm} + \frac{\partial F(u,m)}{\partial m}\frac{dm}{dm} = 0 \tag{9}$$

and re-arranging gives the tangent-linear equation associated with the PDE equation 1

$$\frac{\partial F(u,m)}{\partial u}\frac{du}{dm} = -\frac{\partial F(u,m)}{\partial m}. \tag{10}$$

The tangent equation is always linear.

Consider a functional $J(u,m)$. Let $\widehat{J}(m) := J(u(m),m)$ be a pure function of m. Applying the chain rules gives the expression for the gradient

$$\frac{d}{dm}\widehat{J}(m) = \frac{\partial J(u,m)}{\partial u}\frac{du}{dm} + \frac{\partial J(u,m)}{\partial m}. \tag{11}$$

Having the solution to the tangent-linear equation 10 we can evaluate the gradient of any functional $\widehat{J}$. However, in many applications the functional is fixed and the goal is to calculate the derivative with respect to any parameter the chosen functional depends on. In this case, the alternative is the adjoint approach.

Suppose the tangent-linear system is invertible. Then rewrite the solution to the equation 10 as

$$\frac{du}{dm} = -\left(\frac{\partial F(u,m)}{\partial u}\right)^{-1}\frac{\partial F(u,m)}{\partial m} \tag{12}$$

and substitute $\frac{du}{dm}$ into the expression for the gradient of $\widehat{J}$:

$$\frac{d\widehat{J}(m)}{dm} = -\frac{\partial J(u,m)}{\partial u}\left(\frac{\partial F(u,m)}{\partial u}\right)^{-1}\frac{\partial F(u,m)}{\partial m} + \frac{\partial J(u,m)}{\partial m}. \tag{13}$$

Take the adjoint of the above equation:

$$\frac{d\widehat{J}(m)}{dm}^{*} = -\frac{\partial F(u,m)}{\partial m}^{*}\left(\frac{\partial F(u,m)}{\partial u}\right)^{-*}\frac{\partial J(u,m)}{\partial u}^{*} + \frac{\partial J(u,m)}{\partial m}^{*}. \tag{14}$$

Now define the new variable as

$$\lambda = \left(\frac{\partial F(u,m)}{\partial u}\right)^{-*}\frac{\partial J(u,m)}{\partial u}^{*}. \tag{15}$$

This new variable is called the adjoint variable and it is the solution of the adjoint equation:

$$\frac{\partial F(u,m)}{\partial u}^{*}\lambda = \frac{\partial J(u,m)}{\partial u}^{*}. \tag{16}$$

The adjoint equation is always linear.