Generalized Physics-Informed Learning Through Language-Wide Differentiable Programming

Chris Rackauckas,[1,2] **Alan Edelman,**[1,3] **Keno Fischer,**[3] **Mike Innes**[3]
Elliot Saba,[3] **Viral B. Shah,**[3] **Will Tebbutt**[4]
[1]Massachusetts Institute of Technology, [2]University of Maryland, Baltimore
[3]Julia Computing, [4]University of Cambridge
182 Memorial Dr, Cambridge, MA 02142
crackauc@mit.edu

Abstract

Scientific computing is increasingly incorporating the advancements in machine learning to allow for data-driven physics-informed modeling approaches. However, re-targeting existing scientific computing workloads to machine learning frameworks is both costly and limiting, as scientific simulations tend to use the full feature set of a general purpose programming language. In this manuscript we develop an infrastructure for incorporating deep learning into existing scientific computing code through Differentiable Programming (∂P). We describe a ∂P system that is able to take gradients of full Julia programs, making Automatic Differentiation a first class language feature and compatibility with deep learning pervasive. Our system utilizes the one-language nature of Julia package development to augment the existing package ecosystem with deep learning, supporting almost all language constructs (control flow, recursion, mutation, etc.) while generating high-performance code without requiring any user intervention or refactoring to stage computations. We showcase several examples of physics-informed learning which directly utilizes this extension to existing simulation code: neural surrogate models, machine learning on simulated quantum hardware, and data-driven stochastic dynamical model discovery with neural stochastic differential equations.

Introduction

A casual practitioner might think that scientific computing and machine learning are different scientific disciplines. Modern machine learning has made its mark through breakthroughs in neural networks. Their applicability towards solving a large class of difficult problems in computer science has led to the design of new hardware and software to process extreme amounts of labelled training data, while simultaneously deploying trained models in devices. Scientific computing, in contrast, a discipline that is perhaps as old as computing itself, tends to use a broader set of modelling techniques arising out of the underlying physical phenomena. Compared to the typical machine learning researcher, many computational scientists works with smaller volumes of data but with more computational complexity and range. However, recent results like Physics-Informed Neural Networks (PINNs) suggest that data-efficient machine learning for scientific applications can be found at the intersection of the methods (Raissi, Perdikaris, and Karniadakis 2019). While a major advance, this technique requires re-implementing partial differential equation simulation techniques, such as Runge-Kutta methods, in the context of machine learning frameworks like TensorFlow. Likewise, it would be impractical to require every scientific simulation suite to re-target their extensive stiff differential equation solver and numerical linear algebra stacks to specific machine learning libraries for specific tasks. In order to truly scale physics-informed learning to big science applications, we see a need for efficiently incorporating neural networks into existing scientific simulation suites.

Differentiable Programming (∂P) has the potential to be the lingua franca that can further unite the worlds of scientific computing and machine learning. Here we present a ∂P system which allows for embedding deep neural networks into arbitrary existing scientific simulations, enabling these packages to automatically build surrogates for ML-acceleration and learn missing functions from data. Previous work has shown that differentiable programming systems in domain specific languages for image processing can allow for machine learning integration into the domain applications in a programmer-friendly manner (Li et al. 2018; Li 2019). Supporting multiple languages within a single ∂P system causes an explosion in complexity, vastly increasing the developer effort required, which excludes languages in which packages are developed in alternative languages like Cython or Rcpp. For this reason we extend the full Julia programming language (Bezanson et al. 2017) with differentiable programming capabilities in a way that allows existing package to incorporate deep learning. By choosing the Julia language, we arrive at an abundance of pure-Julia packages for both machine learning and scientific computing with both speed and automatic compatibility, allowing us to test our ideas on fairly large real-world applications.

Our system can be directly used on existing Julia packages, handling user-defined types, general control flow constructs, and plentiful scalar operations through source-to-source mixed-mode automatic differentiation, composing the reverse-mode capabilities of Zygote.jl and Tracker.jl with

ForwardDiff.jl for the "best of both worlds" approach. In this paper we briefly describe how we achieve our goals for a ∂P system and showcase its ability to solve problems which mix machine learning and pre-existing scientific simulation packages.

A simple `sin` example: Differentiate Programs not Formulas

We start out with a very simple example to differentiate $\sin(x)$ written as a program through its Taylor series:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots.$$

Note that the number of terms will not be fixed, but will depend on x through a numerical convergence criterion.

To run, install Julia v1.1 or higher, and install the Zygote.jl and ForwardDiff.jl packages with:

```
using Pkg
Pkg.add("Zygote")
Pkg.add("ForwardDiff")
using Zygote, ForwardDiff

function s(x)
    t = 0.0
    sign = -1.0
    for i in 1:19
        if isodd(i)
            newterm = x^i/factorial(i)
            abs(newterm)<1e-8 && return t
            sign = -sign
            t += sign * newterm
        end
    end
    return t
end
```

While the Taylor series for sine could have been written more compactly in Julia, for purposes of illustrating more complex programs, we purposefully used a loop, a conditional, and function calls to `isodd` and `factorial`, which are native Julia implementations. AD just works, and that is the powerful part of the Julia approach. Let's compute the gradient at x = 1.0 and check whether it matches `cos(1.0)`:

```
julia> ForwardDiff.derivative(s, 1.0)
0.540302303791887

julia> Zygote.gradient(s, 1.0)
(0.5403023037918872,)

julia> cos(1.0)
0.5403023058681398
```

Implementation

Recent progress in tooling for automatic differentiation (AD) has been driven primarily by the machine learning community. Many state of the art reverse-mode AD tools such as Tracker.jl (Innes et al. 2018; Gandhi et al. 2019), PyTorch (PyTorch Team 2018), JAX (Johnson et al. 2018), and TensorFlow (Abadi et al. 2016) (in the recent Eager version) employ tracing methods to extract simplified program representations that are more easily amenable to AD transforms. These traces evaluate derivatives only at specific points in the program space. Unfortunately, this generally unrolls control flow (i.e. building a tape that requires $\mathcal{O}(n)$ memory instead of keeping the loop construct intact) and requires compilation and optimization for every new input.

This choice has been driven largely by the fact that, as the JAX authors put it, "ML workloads often consist of large, accelerable, pure-and-statically-composed (PSC) operations" (Johnson et al. 2018). Indeed, for many ML models the per-executed-operation overhead (in both time and memory) incurred by tracing-based AD systems is immaterial, because these execution time and memory requirements of the operations dwarf any AD overhead.

However, this assumption does not hold for many scientific inverse problems, or even the cutting edge of ML research. Instead, these problems require a ∂P system capable of:

1. Low overhead, independent of the size of the executed operation
2. Efficient support for control flow
3. Complete, efficient support for user defined data types
4. Customizability
5. Composability with existing code unaware of ∂P
6. Dynamism.

Particularly, scientific programs tend to have adaptive algorithms, whose control flow depends on error estimates and thus the current state of the simulation, numerous scalar operations, define large nonlinear models using every term individually or implementing specialized numerical linear algebra routines, and pervasive use of user-defined data structures to describe model components, which require efficient memory handling (stack-allocation) in order for the problem to be computationally feasible.

To take these kinds of problems, Zygote does not utilize the standard methodology and instead generates a derivative function directly from the original source which is able to handle all input values. This is called a source-to-source transformation, a methodology with a long history (Baydin et al. 2018) going back at least to the ADIFOR source-to-source AD program for FORTRAN 77 (Bischof et al. 1996). Using this source-to-source formulation, Zygote can then be compile, heavily optimize, and re-use a single gradient definition for all input values. Significantly, this transformation keeps control flow in tact: not unrolling loops to allow for all possible branches in a memory-efficient form. However, where prior source-to-source AD work has often focused on static languages, Zygote expands upon this idea by supporting a full high level language, dynamic, Julia, in a way that allows for its existing scientific and machine learning package ecosystem to benefit from this tool.

Generality, Flexibility, and Composability

One of the primary design decisions of a ∂P system is how these capabilities should be exposed to the user. One convenient way to do so is using a differential operator $\mathcal{J}$ that

```
function J(f . g)(x)
    a, da = J(f)(x)
    b, db = J(g)(a)
    b, z -> da(db(z))
end
```

Figure 1: The differential operator $\mathcal{J}$ is able to implement the chain rule through a local, syntactic recursive transformation.

```
julia> f(x) = x^2 + 3x + 1
julia> gradient(f, 1/3)
(3.6666666666666665,)

julia> using Measurements;
julia> gradient(f, 1/3 +- 0.01)
(3.6666666666666665 +- 0.02,)
```

Figure 2: With two minimal definitions, Zygote is able to obtain derivatives of any function that only requires those definitions, even through custom data types (such as *Measurement*) and many layers of abstraction.

operates on first class functions and once again returns a first class function (by returning a function we automatically obtain higher order derivatives, through repeated application of $\mathcal{J}$). There are several valid choices for this differential operator, but a convenient choice is

$$\mathcal{J}(f) := x \to (f(x), J_f(x)z),$$

i.e. $\mathcal{J}(f)(x)$ returns the value of f at x, as well as a function which evaluates the jacobian-vector product between $J_f(x)$ and some vector of sensitivities z. From this primitive we can define the gradient of a scalar function $g : \mathbb{R}^n \to \mathbb{R}$ which is written as:

$$\nabla g(x) := [\mathcal{J}(g)(x)]_2\,(1)$$

($[]_2$ selects the second value of the tuple, $1 = \partial z/\partial z$ is the initial sensitivity).

This choice of differential operator is convenient for several reasons: (1) The computation of the forward pass often computes values that can be re-used for the computation of the backwards pass. By combining the two operations, it is easy to re-use this work. (2) It can be used to recursively implement the chain rule (see figure 1).

This second property also suggests the implementation strategy: hard code the operation of $\mathcal{J}$ on a set of primitive f's and let the AD system generate the rest by repeated application of the chain rule transform. This same general approach has been implemented in many systems (Pearlmutter and Siskind 2008; Wang et al. 2018) and a detailed description of how to perform this on Julia's SSA form IR is available in earlier work (Innes 2018).

However, to achieve our extensibility and composability goals, we implement a slight twist on this scheme. We define a fully user extensible function ∂ that provides a default fallback as follows

$$\partial(f)(args...) = \mathcal{J}(f)(args...),$$

where the implementation that is generated automatically by $\mathcal{J}$ recurses to ∂ rather than $\mathcal{J}$ and can thus easily be intercepted using Julia's standard multiple dispatch system at any level of the stack. For example, we might make the following definitions:

$$\begin{aligned}\partial(f)(::\text{typeof}(+))(a::\text{IntOrFloat}, b::\text{IntOrFloat}) = \\ a + b, z \to (z, z) \\ \partial(f)(::\text{typeof}(*))(a::\text{IntOrFloat}, b::\text{IntOrFloat}) = \\ a * b, z \to (z * b, a * z)\end{aligned}$$

i.e. declaring how to compute the partial derivative of $+$ and $*$ for two integer or float-valued numbers, but simultaneously leaving unconstrained the same for other functions or other types of values (which will thus fall back to applying the AD transform). With these two definitions, any program that is ultimately just a composition of '+', and '*' operations of real numbers will work. We show a simple example in figure 2. Here, we used the user-defined *Measurement* type from the *Measurements.jl* package (Giordano 2016). We did not have to define how to differentiate the $\wedge$ function or how to differentiate $+$ and $*$ on a *Measurement*, nor did the *Measurements.jl* package have to be aware of the AD system in order to be differentiated. Thus standard user definitions of types are compatible with the differentiation system. This extra, user-extensible layer of indirection has a number of important consequences:

- **The AD system does not depend on, nor require any knowledge of primitives on new types.** By default we provide implementations of the differentiable operator for many common scalar mathematical and linear algebra operations, written with a scalar LLVM backend and BLAS-like linear algebra operations. This means that even when Julia builds an array type to target TPUs (Fischer and Saba 2018), its XLA IR primitives are able to be used and differentiated without fundamental modifications to our system.
- **Custom gradients become trivial.** Since all operations indirect through ∂, there is no difference between user-defined custom gradients and those provided by the system. They are written using the same mechanism, are co-optimized by the compiler and can be finely targeted using Julia's multiple dispatch mechanism.

Since Julia solves the two language problem, its Base, standard library, and package ecosystem are almost entirely pure Julia. Thus, since our ∂P system does not require primitives to handle new types, this means that almost all functions and types defined throughout the language are automatically supported by Zygote, and users can easily accelerate specific functions as they deem necessary.

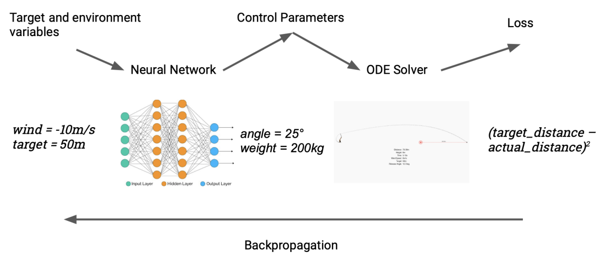

Figure 3: **Using a neural network surrogate to solve inverse problems**

∂P in Practice

Surrogates for Realtime ML-Acceleration of Inverse Problems

Model-based reinforcement learning has advantages over model-agnostic methods, given that an effective agent must approximate the dynamics of its environment (Atkeson and Santamaria 1997). However, model-based approaches have been hindered by the inability to incorporate realistic environmental models into deep learning models. Previous work has had success re-implementing physics engines using machine learning frameworks (Degrave et al. 2019; de Avila Belbute-Peres et al. 2018), but this effort has a large engineering cost, has limitations compared to existing engines, and has limited applicability to other domains such as biology or meteorology. For example, one notable example has shown that solving the 3-body problem can be accelerated 100 million times by using a neural surrogate approach, but required writing a simulator that was compatible with the chosen neural network framework (Breen et al. 2019).

Zygote can be used for control problems, incorporating the model into backpropagation with one call to `gradient`. We pick trebuchet dynamics as a motivating example. Instead of aiming at a single target, we optimize a neural surrogate that can aim it given any target. The neural net takes two inputs, the target distance in metres and the current wind speed. The network outputs trebuchet settings (the mass of the counterweight and the angle of release) that get fed into a simulator which solves an ODE and calculates the achieved distance. We compare to our target and backpropagate through the entire chain to adjust the weights of the network. Our dataset is a randomly chosen set of targets and wind speeds. An agent that aims a trebuchet to a given target can thus be trained in a few minutes on a laptop CPU, resulting in a network which is a surrogate to the inverse problem that allows for aiming in constant-time. Given that the forward-pass of this network is about $100\times$ faster than performing the full optimization on the trebuchet system (Figure 3), this surrogate technique gives the ability to decouple real-time model-based control from the simulation cost through a pre-trained neural network surrogate to the inverse. We present the code for this and other common reinforcement learning examples such as the cartpole and inverted pendulum (Innes, Joy, and Karmali 2019).

Quantum Machine Learning

On the one hand, a promising potential application and research direction for Noisy Intermediate-Scale Quantum (NISQ) technology (Preskill 2018) is *variational quantum circuits* (Benedetti, Lloyd, and Sack 2019), where a quantum circuit is parameterized by classical parameters in quantum gates, which may have less requirements on the hardware. Here, in many cases, the classical parameters are optimized with classical gradient-based algorithms. On the other hand, designing quantum circuits is hard, however, it is potentially an interesting direction to explore using gradient-based method to search circuit architecture for certain task with AD support.

One such state of the art simulator is the Yao.jl (zhe Luo et al. 2019) quantum simulator project. Yao.jl is implemented in Julia and thus composable with our AD technology. There are a number of interesting applications of this combination (Mitarai et al. 2018).

A subtle application is to perform traditional AD of the quantum simulator itself. As a simple example of this capability, we consider a Variational Quantum Eigensolver (VQE) (Kandala et al. 2017). A variational quantum eigensolver is used to compute the eigenvalue of some matrix H (generally the Hamiltonian of some quantum system for which the eigenvalue problem is hard to solve classically, but that is easily encoded into quantum hardware). This is done by using a variational quantum circuit $\Phi(\theta)$ to prepare some quantum state $|\Psi\rangle = \Phi(\theta)|0\rangle$, measuring the expectation value $\langle\Psi|H|\Psi 0\rangle$ and then using a classical optimizer to adjust θ to minimize the measured value. In our example, we will use a 4-site toy Hamiltonian corresponding to an anti-ferromagnetic Heisenberg chain:

$$H = \frac{1}{4}\left[\sum_{\langle i,j\rangle} \sigma_i^x\sigma_j^x + \sigma_i^y\sigma_j^y + \sigma_i^z\sigma_j^z\right]$$

We use a standard differentiable variational quantum circuit composed of layers (2 in our example) of (parameterized) rotations and CNOT entanglers with randomly initialized rotation angles. Figure 4 shows the result of the minimization process as performed using gradients provided by Zygote.

Data-Driven Stochastic Dynamical Model Discovery with Neural Stochastic Differential Equations

Neural latent differential equations (Chen et al. 2018; Álvarez, Luengo, and Lawrence 2009; Hu et al. 2013; Rackauckas et al. 2019) incorporate a neural network into the ODE derivative function. Recent results have shown that many deep learning architectures can be compacted and generalized through neural ODE descriptions (Chen et al. 2018; He et al. 2016; Dupont, Doucet, and Whye Teh 2019; Grathwohl et al. 2018). Latent differential equations have also seen use in time series extrapolation (Gao et al. 2008) and model reduction (Ugalde et al. 2013; Hartman and Mestha 2017; Bar-Sinai et al. 2018; Ordaz-Hernandez, Fischer, and Bennis 2008).

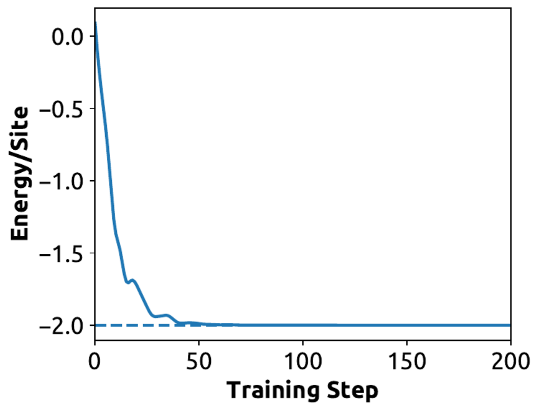

Figure 4: Optimization progress over steps of the classical optimizer to ground state.

Here we demonstrate automatic construction of Langevin equations from data using neural stochastic differential equations (SDE). Typically, Langevin equation models can be written in the form:

$$dX_t = f(X_t)dt + g(X_t)dW_t,$$

where $f : R^n \to R^n$ and $g : R^{n\times m} \to R^n$ with W_t as the m-dimensional Wiener process.

To automatically learn such a physical model, we can replace the drift function f and the diffusion function g with neural networks and train these against time series data by repeatedly solving the differential equation 10 times and using the average gradient of the cost function against the parameters of the neural network. Our simulator utilizes a high strong order adaptive integration provided by DifferentialEquations.jl (Rackauckas and Nie 2017a; 2017b). Figure 5 depicts a two-dimensional neural SDE trained using the l_2 normed distance between the solution and the data points. Included is a forecast of the neural SDE solution beyond the data to a final time of 1.2, showcasing a potential use case for timeseries extrapolation from a learned dynamical model.

The analytical formula for the adjoint of the strong solution of a SDE is difficult to efficiently calculate due to the lack of classical differentiability of the solution. However, Zygote still manages to calculate a useful derivative for optimization with respect to single solutions by treating the Brownian process as fixed and applying forward-mode automatic differentiation, showcasing Zygote's ability to efficiently optimize its AD through mixed-mode approaches (Rackauckas et al. 2018). Common numerical techniques require computing the gradient with respect to a difference over thousands of trajectories to receive an average cost, while our numerical experiments suggest that it is sufficient with Zygote to perform gradient decent on a neural SDE using only single or a few trajectories, reducing the overall computational cost by this thousands. This methodological advance combined with GPU-accelerated high order adaptive SDE integrators in DifferentialEquations.jl makes a whole new field of study

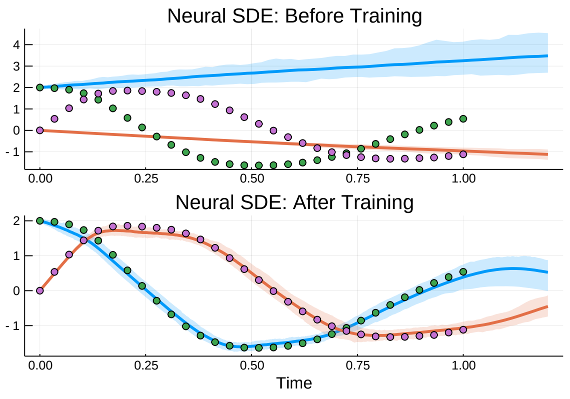

Figure 5: **Neural SDE Training**. For the SDE solution $X(t)$, the blue line shows $X_1(t)$ while the orange line shows $X_2(t)$. The green points shows the fitting data for X_1 while the purple points show the fitting data for X_2. The ribbons show the 95 percentile bounds of the stochastic solutions.

computationally accessible.

Example Code A more extensive code listing for these examples is available at the following URL: https://github.com/MikeInnes/zygote-paper.

Acknowledgments This work would not have been possible without the work of many people. We are particularly indebted to our users that contributed the examples in this paper. Thanks to Roger Luo and JinGuo Liu for the quantum experiments. Thanks also to Zenna Tavares, Jesse Bettencourt and Lyndon White for being early adopters of Zygote and its underlying technology, and shaping its development. Much credit is also due to the core Julia language and compiler team for supporting Zygote's development, including Jameson Nash, Jeff Bezanson and others. Thanks also to James Bradbury for helpful discussions on this paper.

References

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 265–283.

Atkeson, C. G., and Santamaria, J. C. 1997. A comparison of direct and model-based reinforcement learning. In *Proceedings of International Conference on Robotics and Automation*, volume 4, 3557–3564. IEEE.

Bar-Sinai, Y.; Hoyer, S.; Hickey, J.; and Brenner, M. P. 2018. Data-driven discretization: machine learning for coarse graining of partial differential equations. *arXiv e-prints* arXiv:1808.04930.

Baydin, A. G.; Pearlmutter, B. A.; Radul, A. A.; and Siskind, J. M. 2018. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research* 18:1–43.

Benedetti, M.; Lloyd, E.; and Sack, S. 2019. Parameterized quantum circuits as machine learning models. *arXiv preprint arXiv:1906.07682*.

Bezanson, J.; Edelman, A.; Karpinski, S.; and Shah, V. B. 2017.

Julia: A fresh approach to numerical computing. *SIAM Review* 59(1):65–98.

Bischof, C.; Khademi, P.; Mauer, A.; and Carle, A. 1996. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science and Engineering* 3(3):18–32.

Breen, P. G.; Foley, C. N.; Boekholt, T.; and Portegies Zwart, S. 2019. Newton vs the machine: solving the chaotic three-body problem using deep neural networks. *arXiv e-prints* arXiv:1910.07291.

Chen, T. Q.; Rubanova, Y.; Bettencourt, J.; and Duvenaud, D. K. 2018. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, 6571–6583.

de Avila Belbute-Peres, F.; Smith, K.; Allen, K.; Tenenbaum, J.; and Kolter, J. Z. 2018. End-to-end differentiable physics for learning and control. In *Advances in Neural Information Processing Systems*, 7178–7189.

Degrave, J.; Hermans, M.; Dambre, J.; and wyffels, F. 2019. A differentiable physics engine for deep learning in robotics. *Frontiers in Neurorobotics* 13.

Dupont, E.; Doucet, A.; and Whye Teh, Y. 2019. Augmented Neural ODEs. *arXiv e-prints* arXiv:1904.01681.

Fischer, K., and Saba, E. 2018. Automatic full compilation of Julia programs and ML models to cloud TPUs. *CoRR* abs/1810.09868.

Gandhi, D.; Innes, M.; Saba, E.; Fischer, K.; and Shah, V. 2019. Julia E Flux: Modernizando o Aprendizado de Máquina. 5.

Gao, P.; Honkela, A.; Rattray, M.; and Lawrence, N. D. 2008. Gaussian process modelling of latent chemical species: applications to inferring transcription factor activities. *Bioinformatics* 24(16):i70–i75.

Giordano, M. 2016. Uncertainty propagation with functionally correlated quantities. *ArXiv e-prints*.

Grathwohl, W.; Chen, R. T. Q.; Bettencourt, J.; Sutskever, I.; and Duvenaud, D. K. 2018. FFJORD: free-form continuous dynamics for scalable reversible generative models. *CoRR* abs/1810.01367.

Hartman, D., and Mestha, L. K. 2017. A deep learning framework for model reduction of dynamical systems. In *2017 IEEE Conference on Control Technology and Applications (CCTA)*, 1917–1922.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* 770–778.

Hu, Y.; Boker, S.; Neale, M.; and Klump, K. 2013. Coupled latent differential equation with moderators: Simulation and application. *Psychological methods* 19.

Innes, M.; Saba, E.; Fischer, K.; Gandhi, D.; Rudilosso, M. C.; Joy, N. M.; Karmali, T.; Pal, A.; and Shah, V. 2018. Fashionable modelling with Flux. *CoRR* abs/1811.01457.

Innes, M.; Joy, N. M.; and Karmali, T. 2019. Reinforcement learning vs. differentiable programming. https://fluxml.ai/2019/03/05/dp-vs-rl.html.

Innes, M. 2018. Don't unroll adjoint: Differentiating SSA-form programs. *CoRR* abs/1810.07951.

Johnson, M.; Frostig, R.; Maclaurin, D.; and Leary, C. 2018. JAX: Autograd and xla. https://github.com/google/jax.

Kandala, A.; Mezzacapo, A.; Temme, K.; Takita, M.; Brink, M.; Chow, J. M.; and Gambetta, J. M. 2017. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature* 549(7671):242.

Li, T.-M.; Gharbi, M.; Adams, A.; Durand, F.; and Ragan-Kelley, J. 2018. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph. (Proc. SIGGRAPH)* 37(4):139:1–139:13.

Li, T.-M. 2019. Differentiable Visual Computing. *arXiv e-prints* arXiv:1904.12228.

Mitarai, K.; Negoro, M.; Kitagawa, M.; and Fujii, K. 2018. Quantum circuit learning. *Physical Review A* 98(3):032309.

Ordaz-Hernandez, K.; Fischer, X.; and Bennis, F. 2008. Model reduction technique for mechanical behaviour modelling: Efficiency criteria and validity domain assessment. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science* 222(3):493–505.

Pearlmutter, B. A., and Siskind, J. M. 2008. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30(2):7.

Preskill, J. 2018. Quantum computing in the nisq era and beyond. *Quantum* 2:79.

PyTorch Team. 2018. The road to 1.0: production ready PyTorch. https://pytorch.org/blog/a-year-in/. Accessed: 2018-09-22.

Rackauckas, C., and Nie, Q. 2017a. Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia. 5(1). Exported from https://app.dimensions.ai on 2019/05/05.

Rackauckas, C. V., and Nie, Q. 2017b. Adaptive methods for stochastic differential equations via natural embeddings and rejection sampling with memory. *Discrete and continuous dynamical systems. Series B* 22 7:2731–2761.

Rackauckas, C.; Ma, Y.; Dixit, V.; Guo, X.; Innes, M.; Revels, J.; Nyberg, J.; and Ivaturi, V. 2018. A Comparison of Automatic Differentiation and Continuous Sensitivity Analysis for Derivatives of Differential Equation Solutions. *arXiv e-prints* arXiv:1812.01892.

Rackauckas, C.; Innes, M.; Ma, Y.; Bettencourt, J.; White, L.; and Dixit, V. 2019. Diffeqflux.jl - A julia library for neural differential equations. *CoRR* abs/1902.02376.

Raissi, M.; Perdikaris, P.; and Karniadakis, G. 2019. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics* 378:686 – 707.

Ugalde, H. M. R.; Carmona, J.-C.; Alvarado, V. M.; and Reyes-Reyes, J. 2013. Neural network design and model reduction approach for black box nonlinear system identification with reduced number of parameters. *Neurocomputing* 101:170 – 180.

Wang, F.; Wu, X.; Essertel, G.; Decker, J.; and Rompf, T. 2018. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *arXiv preprint arXiv:1803.10228.*

zhe Luo, X.; guo Liu, J.; Zhang, P.; and Wang, L. 2019. Yao.jl: Extensible, efficient quantum algorithm design for humans. In preparation.

Álvarez, M.; Luengo, D.; and Lawrence, N. D. 2009. Latent force models. In van Dyk, D., and Welling, M., eds., *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, 9–16. Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA: PMLR.