∂P: A Differentiable Programming System to Bridge Machine Learning and Scientific Computing

Mike Innes[1], Alan Edelman[2], Keno Fischer[1], Chris Rackauckas[2, 3], Elliot Saba[1], Viral B. Shah[1], and Will Tebbutt[4]

[1]Julia Computing, Inc.
[2]Massachusetts Institute of Technology
[3]University of Maryland, Baltimore
[4]University of Cambridge

ABSTRACT

We describe a Differentiable Programming (∂P) system that is able to take gradients of Julia programs making Automatic Differentiation a first class language feature. Our system supports almost all language constructs (control flow, recursion, mutation, etc.) and compiles high-performance code without requiring any user intervention or refactoring to stage computations. This enables an expressive programming model for deep learning and, more importantly, it enables users to utilize the existing Julia ecosystem of scientific computing packages in deep learning models.

Proceedings of JuliaCon

Keywords

Julia, Differentiable Programming, Automatic Differentiation, Compiler

1. Introduction

At first glance, 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. As we look closer, many similarities emerge. Both disciplines have a preference for using dynamic languages such as Python, R and Julia. Often, performance critical sections in Python and R are written in C++ and Fortran, less so in Julia. The core computational routines are grounded in numerical linear algebra, and fundamentally, hardware has been designed to accelerate this computational core.

There are deep historical and intellectual links between machine learning and scientific computing. On the surface, deep learning appears to be data driven and scientific computing is about very technical numerically intensive differential equations that mirror physical processes. It is our viewpoint that the fields are closer than is often realized with opportunities for machine learning and scientific computing to benefit from stronger interactions.

Specifically, machine learning has strongly benefited from **Numerical Linear Algebra** software. The optimized numerical linear algebra stack was first developed in the context of scientific computing - BLAS (Levels 1 through 3) kernels [15], LAPACK routines for computing matrix factorizations [3], and MPI for message passing in parallel computing [1]. The early CUDA libraries were developed by Nvidia for accelerating scientific kernels on GPUs.

This trend is continuing with more recent techniques such as **Neural ODEs**. A neural ODE [11] is a neural network layer which compacts the L layers of a ResNet into a single ODE definition solved in $N < L$ steps of an adaptive ODE solver. By changing to this form, the memory and computational costs for the residual network model are decreased. Additionally, the continuous ODE formulation does not restrict the model to a grid like the traditional RNN, naturally allowing for fitting time series with irregular observations. Existing ODE solver software can then be employed for efficient solution of the system.

This is a two-way street. The 2018 Gordon Bell prize [37], awarded for the largest high-performance scientific computing application, applied deep learning to climate analytics at exascale. Scientific computing also has much to benefit from advances in machine learning:

+

(1) **Surrogate modeling:** Scientific simulations are often expensive to run as they evaluate a system using first principles.

These simulations can be accelerated by having machine learning models approximate the input-output relation. Neural networks or other surrogate models can be trained on expensive simulations once and then used repeatedly in place of the simulations, making it possible to explore the parameter space, propagate uncertainties, and fit the data in ways that have previously been impossible.

(2) **Adjoint sensitivity analysis:** Calculating the adjoint of an ordinary differential equation system $\frac{du}{dt} = f(u, p, t)$ requires solving the reverse ODE $\frac{d\lambda^*}{dt} = \lambda^* \frac{df}{du} + \frac{df}{dp}$. The term $\lambda^* \frac{df}{du}$ is the primitive of backpropogation, and thus applying machine learning AD tooling to the ODE function f accelerates the scientific computing adjoint calculations.

(3) **Inverse problems:** For many parameterized scientific simulations, one can ask "what parameters would make my model best fit the data?" This inverse problem is pervasive yet difficult because it requires the ability to efficiently compute the gradient of a large existing simulation. One can train a model on a simulator, which can then be used to quickly solve inverse problems, but this currently requires generating massive amounts of simulation data for training, which is slow and computationally expensive. By being able to differentiate through simulators, we can learn much more quickly and efficiently.

(4) **Probabilistic Programming:** Inference on statistical models is a crucial tool in the sciences. Probabilistic programming enables more complex models and scaling to huge data sets by combining statistical methods with the generality of programming language constructs. Automatic differentiation is the backbone of many probabilistic programming tools, but domain specific languages lack access to an existing ecosystem of tools and packages. ∂P in a general purpose language has the benefit of higher composability, access to better abstractions, and enabling richer and more accurate models. The Turing.jl [21] and Gen.jl [12] packages are excellent examples of these capabilities.

Differentiable Programming (∂P) has the potential to be the lingua franca that can further unite the worlds of scientific computing and machine learning. The choice of a language to implement this system is an important one. Supporting multiple languages within a single ∂P system causes an explosion in complexity, vastly increasing the developer effort required. Our ∂P system extends the Julia programming language [8] with differentiable programming capabilities. We chose the Julia language because of the abundance of pure-Julia packages for both machine learning and scientific computing 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, state-based control flow, and plentiful scalar operations through source-to-source AD. 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.

1.1 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:

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

One feature of our example is 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
            println("i=",i)
            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, a print statement, 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> # Forward Mode AD

julia> ForwardDiff.derivative(s, 1.0)
i=1
i=3
i=5
i=7
i=9
i=11
0.540302303791887

julia> Zygote.gradient(s, 1.0) # Reverse Mode AD
i=1
i=3
i=5
i=7
i=9
i=11
(0.5403023037918872,)

julia> cos(1.0)
0.5403023058681398
```

2. 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 [31, 19], PyTorch [47], JAX [33], and TensorFlow [2] (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 all control flow and requires compilation and optimization for every new input value.

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" [33]. 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, and (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 [6] going back at least to the ADIFOR source-to-source AD program for FORTRAN 77 [9]. 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.

2.1 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 operates on first class functions and once again returns a first class function (by returning a function we automatically obtain higher order derivatives,

```
function 𝒥(f ∘ g)(x)
    a, a̲ =𝒥(f)(x)
    b, b̲ =𝒥(g)(a)
    b, z → begin
        a̲(b̲(z))
    end
end
```

Fig. 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,)
```

Fig. 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.

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 [44, 53] and a detailed description of how to perform this on Julia's SSA form IR is available in earlier work [30].

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:

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

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

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 [22]. 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. 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 [17], 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.

3. ∂P in Practice

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

3.1 Deep Learning

Zygote is a flexible backend for calculating gradients of deep learning models. A typical example is shown here, where a recurrent architecture using LSTMs [27] is used to learn Shakespeare. The code sample below demonstrates many powerful elements of Zygote, making use of several convenient Julia features in the process. First, the defined model has no special data types within it to enable AD; the models are defined for forward-pass calculation only, with backwards-pass definitions only for basic building blocks such as BLAS operations and basic array manipulation. Zygote is used to wrap the loss computation, explicitly denoting the bounds of the computation that should be differentiated to calculate the gradients imposed upon the model, but all other pieces of code (including the LSTM layer definition itself) are written without automatic differentiation in mind. This model executes on the CPU, GPU [31] and Google TPU architecture [17], with little to no change.

```julia
alphabet, Xs, Ys = load_data("shakespeare_input.txt")

model = Chain(
LSTM(length(alphabet), 128),
LSTM(128, 128),
Dense(128, length(alphabet)),
softmax)

opt = ADAM(0.01)

# Run through our entire dataset a few times
for epoch_idx in 1:10,
  (x_batch, y_batch) in zip(Xs, Ys)

  # Calculate gradients upon the model for this
  # batch of data, summing crossentropy loss
  # across each time index in this batch.
  grads = \Zygoteplain.gradient(model) do model
    return sum(crossentropy.(model.(x_batch), y_batch))
  end

  # Update the model, then reset its internal
  # LSTM states
  model = update!(opt, model, grads)
  Flux.reset!(model)
end
```

Zygote provides an extremely low-overhead AD interface. By performing source-to-source transformation, there is little runtime overhead from the AD transformation, beyond the actual cost of computing the backwards pass[1]. In addition, it has been shown to perform at the same level as TensorFlow for ResNet on a TPU pod [18].

In order to measure this, we have benchmarked the backwards pass of a stacked LSTM network, measuring the runtime as batch size trends toward zero. We hypothesize that overall runtime should be linear in batch size, and so by linearly extrapolating our readings to a batch size of zero, we will be able to estimate the fixed overhead induced by each operation within the AD system, where an operation is a single adjoint definition. We furthermore verify that this overhead is linear in the number of ops by measuring this for a variable number of stacked LSTMs and recording the number of ops per model.

Our benchmarks were run on an Intel(R) Core(TM) i5-6600 CPU @ 3.30GHz processor, running Linux with Julia version 1.3.

[1]In practice Zygote may currently generate code that pessimizes certain compiler assumption and thus leads to slower execution. We are working on improving the compiler for these case to achieve true zero overhead.

Layers	Total Overhead	Number of Ops	Overhead/op
1	147.0 us	255	576.3 ns
2	280.5 us	491	571.3 ns
3	406.1 us	727	558.6 ns

Table 1. : Zygote per-op overhead estimated across varying numbers of stacked LSTM layers.

In all tests, multithreading was disabled both in the AD framework and in the underlying BLAS library, so as to avoid performance cliffs around multithreading thresholds. Our results are shown in Table 1, displaying an average overhead of **568.8** ns per adjoint definition. This is a highly competitive as compared to other frameworks such as PyTorch, where *op* overhead is at least 1 μs [46].

This vanishing overhead lowers the bar for AD systems to be integrated more thoroughly into programming languages at an extremely fine scale without worrying about performance issues. The lower the AD overhead, the smaller the minimum feasible kernel size for an AD system that is restricted by backwards pass efficiency.

3.2 Differentiating a Trebuchet

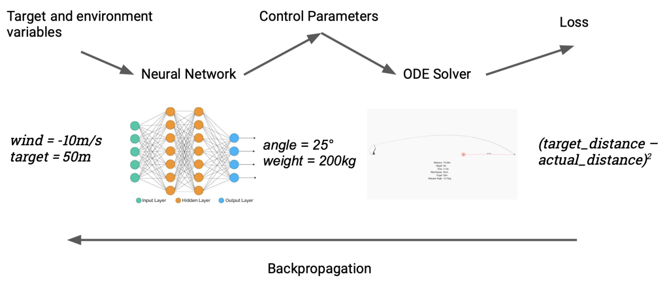

Fig. 3: Using a neural network surrogate to solve inverse problems

Model-based reinforcement learning has advantages over model-agnostic methods, given that an effective agent must approximate the dynamics of its environment [4]. 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 [14, 13], 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.

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 network 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 then 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 solves the inverse problem with constant-time aiming that is $100\times$ faster than optimizing the trebuchet system directly (Figure 3). We present the code for this and other common reinforcement learning examples such as the cartpole and inverted pendulum [32].

3.3 Computer Vision

An emerging direction for computer vision is to see the problem as 'inverse graphics': where vision models take pixel arrays to scene parameters, renderers take scene parameters to pixel arrays. Many high-fidelity, photo-realistic renderers are available which contain a vast amount of implicit knowledge about this mapping, and differentiation allows renderers and models to be used in an autoencoder-like setup to quickly bootstrap a full vision model.

As in other cases, the difficulty lies in getting efficient derivatives from a production-ready renderer, typically written in a performance language like C++. We repeat the going themes of this paper: ML framework-based reimplementations are typically limited compared to existing renderers (both by the available framework operations and by the years of work from rendering experts that has gone into production renderers), and workarounds such as Monte Carlo sampling [38] impose a high efficiency cost (around $40\times$ slower than a single render) compared to AD (at most around $5\times$ due to division, in principle). The Julia community's approach is to build a renderer suitable, first and foremost, for traditional production rendering tasks, using a general and high-performance numerical language, and then differentiate it. Of course, this approach does not preclude the use of domain-specific methods such as Monte Carlo sampling where appropriate (section 3.6).

In our examples we have used our prototype renderer [43] to demonstrate optimization of the position of a point light source, given the desired final rendered image. We define a loss function that accepts a point light source as input, renders the scene, and compares it to a reference image. As usual, gradients are then trivial and we can begin to update the position of the point source.

```julia
guess = PointLight(Vec3(1.0),
                   20000.0,
                   Vec3(1.0, 2.0, -7.0))

function loss_function(light)
  rendered_color =
    raytrace(origin, direction, scene, light, eye_pos)
  rendered_img =
    process_image(rendered_color, screen_size.w,
                  screen_size.h)
  return mean((rendered_img .- reference_img).^2)
end

gs = gradient(x -> loss_function(x, image), guess)
```

3.4 Financial Derivatives

Much of finance is concerned with how the value of a security will change according to changes in the market factors, such as interest rates, market indices, or spot prices of underlying assets. This is naturally expressed via the derivatives of security prices with respect to various conditions. Previous work has noted that financial

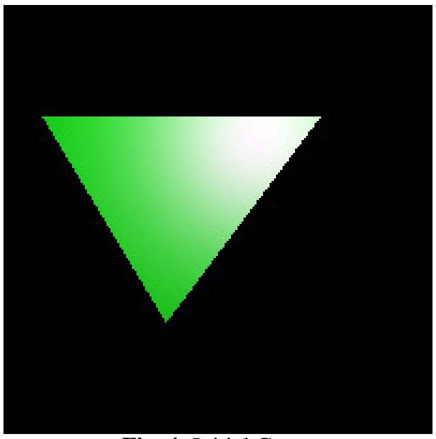

Fig. 4: Initial Guess

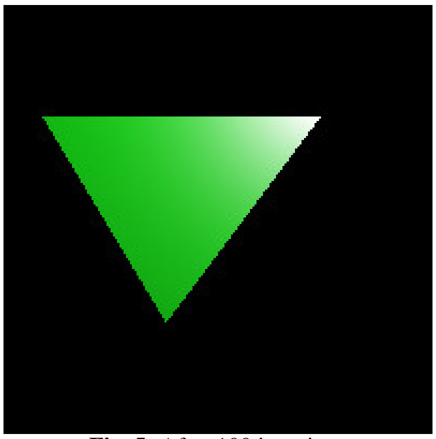

Fig. 5: After 100 iterations

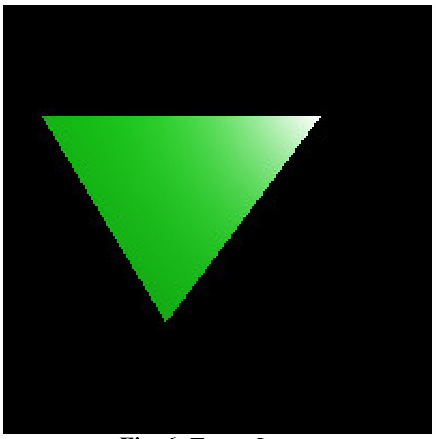

Fig. 6: Target Image

contracts are compositional, and form a kind of programming language [34, 35]. Work in Julia has generalized this to a differentiable contract language, Miletus [10], where greeks can be derived for arbitrary contracts. Contract valuation can be mixed with free-form market simulations as well as other ∂P tools like neural networks, enabling differentiable programming for finance.

In working with fixed-income securities such as bonds, the primary factors are interest rate curves. These are typically quoted as the *par rate* at a set of times in the future (e.g. 1 month, 3 month, 6 month, 1 year, etc.), which are referred to as "key rates". Using this to determine the price of a bond requires (1) interpolating a yield curve from the par rates: this is a process known as *bootstrapping*, and requires solving a sequence of implicit problems, typically via Newton iterations; and (2) computing the bond price from the yield curve by discounting cash flows.

The derivatives of a bond price with respect to the key rates are known as *key rate durations*: computing these via AD requires higher-order derivatives (in order to differentiate through the Newton solver). However, this is entirely invisible to the programmer; several unrelated uses of differentiation, and even entirely different ADs, simply compose together and do the right thing.

3.5 Quantum Machine Learning

On the one hand, a promising potential application and research direction for Noisy Intermediate-Scale Quantum (NISQ) technology [45] is *variational quantum circuits* [7], 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 [55] 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 [41, 39, 40].

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) [36]. 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. The corresponding code is showing in figure 8 [2]. The resulting plot can be seen in figure 8.

[2]For space efficiency, some details are factored into utility function omitted from this code listing. However, the code is very similar to the optimization loop from the deep learning example above. A full, expanded version of this example can be found with the full code listing in the supplemental material.

```
using Yao, Zygote
const nsites = 4
let H = heisenberg(nsites),
    v0 = statevec(zero_state(nsites))
    energy(circuit) =
        (Ψ =circuit*v0;
         real(Ψ′ *H *Ψ))
end
circuit_init =
    random_diff_circuit(nsites, 2)
optimize_plot_loss(
    energy, circuit_init, ADAM(0.1))
```

Fig. 7: An ADAM optimizer is used to tune parameters of a variational quantum circuit to find the ground state of a 4-site anti-ferromagnetic Heisenberg chain Hamiltonian. The necessary gradients are obtained by automatic differentiation of a Yao.jl quantum simulator.

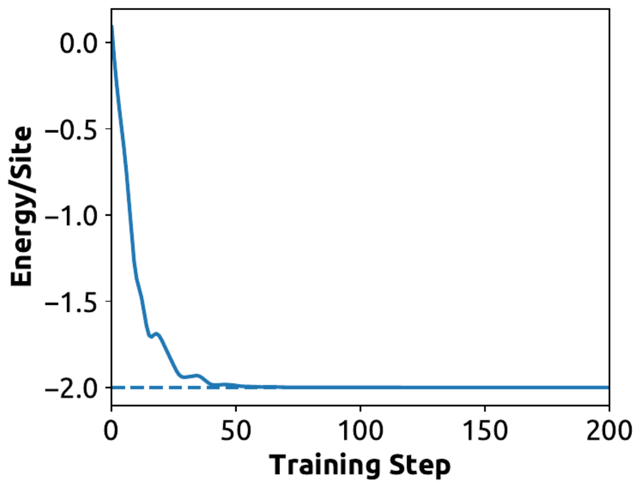

Fig. 8: Optimization progress over steps of the classical optimizer to ground state.

3.6 Neural Differential Equations with applications in finance

Neural latent differential equations [11, 56, 28, 48] 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 [11, 26, 16, 24]. Latent differential equations have also seen use in time series extrapolation [20] and model reduction [52, 25, 5, 42].

Here we demonstrate financial time series forecasting with a neural stochastic differential equation (SDE). Typically, financial SDE models follow 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. For example, the infamous Black-Scholes equation

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

is related through the Feynman-Kac Theorem to a probability measure driven by a geometric Brownian motion stock price $dS_t = rS_t dt + \sigma S_t dW_t$, where S is the stock price, r is the fixed interest rate of an option, and σ is the volatility. This signifies that the true value of an option contract can then be achieved by hedging such that the following conditional expectation holds:

$$V(S,t) = \mathbb{E}\left[\int_t^T e^{-\int_t^T r d\tau} f(S_\nu) d\nu + e^{-\int_t^T r d\tau}\psi(S_T) d\nu\right]$$

where $\psi(S_\nu)$ is the value of the contract at the terminal time T given a stock price S, showing that the PDE's solution is given by an average over the SDE solutions.

To generalize the model, we replacing the fixed interest rate r with a neural network train against financial time series data. Our financial simulator utilizes a high strong order adaptive integration provided by DifferentialEquations.jl [50, 51]. Figure 9 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.

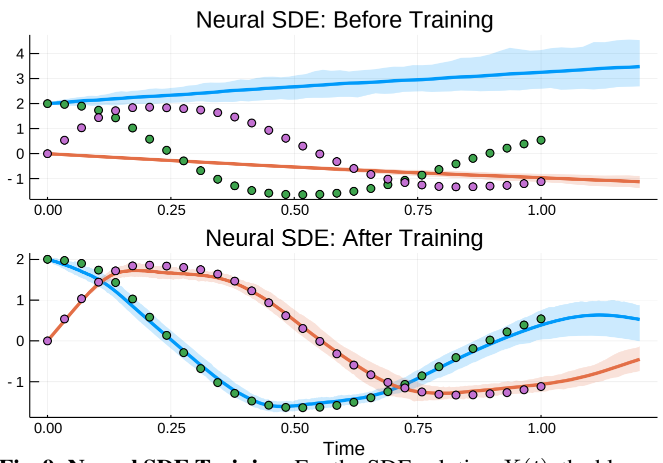

Fig. 9: Neural SDE Training. For the SDE solution $X(t)$, the blue line shows $X_1(t)$ while the orange line shows $X_2(t)$. The green points shows the fitting data for X_1 while the purple points show the fitting data for X_2. The ribbons show the 95 percentile bounds of the stochastic solutions.

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[3]. However, Zygote still manages to calculate a useful derivative for optimization with respect to single

[3]The the strong solution of stochastic differential equations are non-differentiable except on a measure zero set with probability 1. This means that standard Newtonian calculus cannot hold, and thus the derivation must take place in a the setting of a stochastic functional calculus known as the Malliavin Calculus [23, 29, 54].

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 [49]. 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 single 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 accessible.

4. Conclusion

The disciplines of machine learning and scientific computing have much to share. We presented a ∂P system that can serve as the basis of a common shared infrastructure in both disciplines. We demonstrated how we can compose ideas in machine learning and scientific computing to allow for new applications that transcend these domains, by using the same technology to differentiate programs in both domains. On the ML side, we show the same performance as existing ML frameworks for deep learning models (on CPUs, GPUs, and TPUs) and in reinforcement learning. In the case of scientific computing, we show neural SDEs and quantum machine learning. The system is open source, and we invite the reader to try their own examples.

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 Kai Xu for the Bayesian colours example, Avik Pal for building our differentiable ray tracer, Tejan Karmali and Neethu Mariya Joy for the model-based RL experiments, Roger Luo and JinGuo Liu for quantum experiments, and Simon Byrne for the finance example. 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 and many other things.

5. References

[1] *MPI - A Message Passing Interface Standard*. MPI Forum, 2015.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[3] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and D Sorensen. *LAPACK Users' guide*, volume 9. SIAM, 1999.

[4] Christopher G Atkeson and Juan Carlos Santamaria. A comparison of direct and model-based reinforcement learning. In *Proceedings of International Conference on Robotics and Automation*, volume 4, pages 3557–3564. IEEE, 1997.

[5] Yohai Bar-Sinai, Stephan Hoyer, Jason Hickey, and Michael P. Brenner. Data-driven discretization: machine learning for coarse graining of partial differential equations. *arXiv e-prints*, page arXiv:1808.04930, Aug 2018.

[6] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[7] Marcello Benedetti, Erika Lloyd, and Stefan Sack. Parameterized quantum circuits as machine learning models. *arXiv preprint arXiv:1906.07682*, 2019.

[8] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[9] Christian Bischof, Peyvand Khademi, Andrew Mauer, and Alan Carle. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science and Engineering*, 3(3):18–32, 1996.

[10] Simon Byrne. Miletus: Writing financial contracts in julia, 2019.

[11] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, pages 6571–6583, 2018.

[12] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. Gen: A general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 221–236, New York, NY, USA, 2019. ACM.

[13] Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. In *Advances in Neural Information Processing Systems*, pages 7178–7189, 2018.

[14] Jonas Degrave, Michiel Hermans, Joni Dambre, and Francis wyffels. A differentiable physics engine for deep learning in robotics. *Frontiers in Neurorobotics*, 13, Mar 2019.

[15] Jack J Dongarra, Jermey Du Cruz, Sven Hammarling, and Iain S Duff. Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):18–28, 1990.

[16] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented Neural ODEs. *arXiv e-prints*, page arXiv:1904.01681, Apr 2019.

[17] Keno Fischer and Elliot Saba. Automatic full compilation of Julia programs and ML models to cloud TPUs. *CoRR*, abs/1810.09868, 2018.

[18] Keno Fischer and Elliot Saba. XLA.jl: Compiling Julia to XLA. https://github.com/JuliaTPU/XLA.jl, 2018.

[19] Dhairya Gandhi, Michael Innes, Elliot Saba, Keno Fischer, and Viral Shah. Julia E Flux: Modernizando o Aprendizado de Máquina. page 5, 2019.

[20] Pei Gao, Antti Honkela, Magnus Rattray, and Neil D. Lawrence. Gaussian process modelling of latent chemical species: applications to inferring transcription factor activities. *Bioinformatics*, 24(16):i70–i75, 08 2008.

[21] Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: Composable inference for probabilistic programming. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*, pages 1682–1690, 2018.

[22] Mosè Giordano. Uncertainty propagation with functionally correlated quantities. *ArXiv e-prints*, October 2016.

[23] E. Gobet and R. Munos. Sensitivity analysis using Itô–Malliavin calculus and martingales, and application to stochastic optimal control. *SIAM Journal on Control and Optimization*, 43(5):1676–1713, 2005.

[24] Will Grathwohl, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, and David K. Duvenaud. FFJORD: free-form continuous dynamics for scalable reversible generative models. *CoRR*, abs/1810.01367, 2018.

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

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[27] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[28] Yueqin Hu, Steve Boker, Michael Neale, and Kelly Klump. Coupled latent differential equation with moderators: Simulation and application. *Psychological methods*, 19, 05 2013.

[29] Zhi-yuan Huang and Jia-an Yan. Malliavin calculus. In *Introduction to Infinite Dimensional Stochastic Analysis*, pages 59–112. Springer, 2000.

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

[31] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. Fashionable modelling with Flux. *CoRR*, abs/1811.01457, 2018.

[32] Mike Innes, Neethu Maria Joy, and Tejan Karmali. Reinforcement learning vs. differentiable programming. `https://fluxml.ai/2019/03/05/dp-vs-rl.html`, 2019.

[33] Matt Johnson, Roy Frostig, Dougal Maclaurin, and Chris Leary. JAX: Autograd and xla. `https://github.com/google/jax`, 2018.

[34] S Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering. *ACM SIG-PLAN Notices*, 35(9):280–292, 2000.

[35] SL Peyton Jones and Jean-Marc Eber. How to write a financial contract. 2003.

[36] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M Chow, and Jay M Gambetta. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature*, 549(7671):242, 2017.

[37] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, et al. Exascale deep learning for climate analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 51. IEEE Press, 2018.

[38] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable monte carlo ray tracing through edge sampling. In *SIGGRAPH Asia 2018 Technical Papers*, page 222. ACM, 2018.

[39] Jin-Guo Liu and Lei Wang. Differentiable learning of quantum circuit born machines. *Physical Review A*, 98(6):062324, 2018.

[40] Jin-Guo Liu, Yi-Hong Zhang, Yuan Wan, and Lei Wang. Variational quantum eigensolver with fewer qubits. *arXiv preprint arXiv:1902.02663*, 2019.

[41] Kosuke Mitarai, Makoto Negoro, Masahiro Kitagawa, and Keisuke Fujii. Quantum circuit learning. *Physical Review A*, 98(3):032309, 2018.

[42] K Ordaz-Hernandez, X Fischer, and F Bennis. 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, 2008.

[43] Avik Pal. Raytracer.jl: A differentiable renderer that supports parameter optimization for scene reconstruction. *CoRR*, abs/1907.07198, 2019.

[44] Barak A Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):7, 2008.

[45] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.

[46] PyTorch Team. PyTorch, a, year in... `pytorch.org/blog/a-year-in`, 2018. Accessed: 2018-09-22.

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

[48] Christopher Rackauckas, Mike Innes, Yingbo Ma, Jesse Bettencourt, Lyndon White, and Vaibhav Dixit. Diffeqflux.jl - A julia library for neural differential equations. *CoRR*, abs/1902.02376, 2019.

[49] Christopher Rackauckas, Yingbo Ma, Vaibhav Dixit, Xingjian Guo, Mike Innes, Jarrett Revels, Joakim Nyberg, and Vijay Ivaturi. A Comparison of Automatic Differentiation and Continuous Sensitivity Analysis for Derivatives of Differential Equation Solutions. *arXiv e-prints*, page arXiv:1812.01892, Dec 2018.

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

[51] Christopher V Rackauckas and Qing Nie. 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, 2017.

[52] Hector M. Romero Ugalde, Jean-Claude Carmona, Victor M. Alvarado, and Juan Reyes-Reyes. Neural network design and model reduction approach for black box nonlinear system identification with reduced number of parameters. *Neurocomputing*, 101:170 – 180, 2013.

[53] Fei Wang, Xilun Wu, Gregory Essertel, James Decker, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *arXiv preprint arXiv:1803.10228*, 2018.

[54] Han Zhang. *The Malliavan Calculus*. PhD thesis, 2004.

[55] Xiu zhe Luo, Jin guo Liu, Pan Zhang, and Lei Wang. Yao.jl: Extensible, efficient quantum algorithm design for humans. In preparation, 2019.

[56] Mauricio Álvarez, David Luengo, and Neil D. Lawrence. Latent force models. In David van Dyk and Max Welling, editors, *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 9–16, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr 2009. PMLR.