Proceedings of the 2022 Winter Simulation Conference
B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C.G. Corlu, L.H. Lee, E.P. Chew, T. Roeder, and P. Lendermann, eds.

AUTOMATIC DIFFERENTIATION FOR GRADIENT ESTIMATORS IN SIMULATION

Matthew T. Ford
Shane G. Henderson

School of Operations Research and
Information Engineering
Cornell University
206 Rhodes Hall
Ithaca, NY 14853, USA

David J. Eckman

Department of Industrial and
Systems Engineering
Texas A&M University
3131 TAMU
College Station, TX 77843, USA

ABSTRACT

Automatic differentiation (AD) can provide infinitesimal perturbation analysis (IPA) derivative estimates directly from simulation code. These gradient estimators are simple to obtain analytically, at least in principle, but may be tedious to derive and implement in code. AD software tools aim to ease this workload by requiring little more than writing the simulation code. We review considerations when choosing an AD tool for simulation, demonstrate how to apply some specific AD tools to simulation, and provide insightful experiments highlighting the effects of different choices to be made when applying AD in simulation.

1 INTRODUCTION

Gradient estimation is important in both sensitivity analysis and optimization in the practice of simulation. We emphasize the simulation-optimization context for concreteness, but the ideas we present also apply to sensitivity analysis. Consider the problem of minimizing $f(x) = \mathbb{E}h(x,Y)$ where $x \in \mathbb{R}^d$ is a vector of decision variables, Y is a random element capturing all stochastic primitives in the simulation, $h(x,Y)$ represents the logic of the simulation model returning some real-valued performance measure, and $f\colon \mathbb{R}^d \to \mathbb{R}$ is the objective function. We assume that x is a *structural* parameter, i.e., the distribution of Y does not depend on x. This is essentially without loss of generality since x can include any finite set of parameters that are needed to map Uniform$(0,1)$ random variables, which are then represented as Y, to the simulation output.

Assuming appropriate differentiability conditions, first-order optimization algorithms use information about the gradient of the objective function at a solution x, $\nabla f(x)$, to try to find a better neighboring solution x' such that $f(x') < f(x)$. In the simulation-optimization setting, we do not have direct access to $f(\cdot)$ or $\nabla f(\cdot)$. However, we have the ability to sample Y and then compute $h(x,Y)$ for any x. We may also be able to compute the gradient of $h(\cdot,Y)$ at x, $\nabla h(x,Y)$, through analytical derivation and manual coding, or using automatic-differentiation (AD) software. In that event, $\nabla h(x,Y)$ can be used to construct estimators of $\nabla f(x)$ to be used by first-order optimization algorithms.

We discuss considerations when selecting and using AD software for gradient estimation. Many modern AD tools have been designed primarily for machine-learning (ML) problems, e.g., PyTorch (Paszke et al. 2019), JAX (Bradbury et al. 2018), Autograd (Maclaurin et al. 2015), and Tensorflow (Abadi et al. 2015), and thus are tailored to a different assumed program structure than what is typical in simulation. As we note in Section 2, there is a direct relationship between AD and infinitesimal perturbation analysis (IPA) gradient estimators. There we also discuss some general computational principles of AD. Section 3 reviews some of the main AD tools and their suitability for simulation modeling in general. Section 4 discusses adapting simulation code for use with two of those tools, and Section 5 explores their use in the Python library SimOpt (Eckman et al. 2020). Section 6 provides timing experiments for running the various tools

on two relatively simple examples and Section 7 discusses the results of the experiments and provides some final perspectives. Code for experiments will be made available in a public repository (Ford 2022).

The use of AD in simulation has been contemplated for a long time, e.g., Fu et al. (2000). One area of simulation in which AD has found success is finance, e.g., Giles and Glasserman (2006), Glasserman (2013), Fries (2017), and Fries (2019), perhaps because of the relative simplicity of pricing simulation models compared to discrete-event simulation models. Another area is at the intersection of Monte Carlo simulation and machine learning, such as in variational inference (Kucukelbir et al. 2017) or in modern implementations of policy gradient estimators in reinforcement learning, building upon Williams (1992). Relative to such work, we emphasize issues with adapting existing AD software tools to simulation models, using two simple examples to emphasize the key messages and gain insights into coding complexity and run times.

2 GRADIENT ESTIMATION AND AUTOMATIC DIFFERENTIATION

When x is a structural parameter we can use infinitesimal perturbation analysis (IPA) for gradient estimation (Glasserman 1991). The value of the IPA gradient estimator $\nabla h(x,Y)$ can be obtained by differentiating (with respect to x) the source code describing the simulation logic h and evaluating the result at (x,Y).

The IPA gradient estimator is unbiased at a solution x if the interchange of differentiation and expectation,

$$\nabla f(x) = \nabla \mathbb{E} h(x,Y) = \mathbb{E} \nabla h(x,Y), \tag{1}$$

holds (Fu and Hu 1997). Cao (1985) gives conditions for when this interchange holds; see also Asmussen and Glynn (2007), VII.2). If these conditions do not hold, we can sometimes replace $h(x,Y)$ by a conditional expectation that is differentiable in x to obtain an unbiased gradient estimator; this technique is known as Smoothed Perturbation Analysis (Gong and Ho 1987). Even if the IPA gradient estimator is biased, it may still be useful for optimization (Eckman and Henderson 2020).

Relative to other gradient estimation methods, IPA gradient estimators possess certain advantages. IPA gradient estimators can be obtained from a single simulation replication, i.e., one realization of Y and evaluation of $\nabla h(x,Y)$, while methods such as finite differences or weak derivatives typically require approximately $2d$ replications to obtain a single gradient estimate. In this sense, IPA is computationally efficient. The method of finite differences also suffers from needing to choose a step size that balances bias and variance; IPA requires no such choice. Often, likelihood ratio gradient estimators (Glynn 1990) can also be obtained from a single replication, but they have been seen to suffer from higher variance than IPA (Cui et al. 2020). A further advantage of IPA gradient estimators relative to their competitors is that they are the *exact* (up to numerical precision) gradients of the pathwise function, which ensures that the function and gradient estimates are consistent with one another. This consistency could be expected by solvers when, e.g., taking very small steps. The consistency does *not* hold for gradient estimates obtained using, e.g., the likelihood ratio method. Fu (2015) provides a recent survey of gradient estimation in simulation. In the rest of this paper we focus on IPA gradient estimators and their calculation using AD.

AD is a method to differentiate functions which are compositions of primitive functions whose gradients are computable at all arguments of interest (Griewank 1989). The chain rule provides a formula to differentiate the composite function, which is $h(x,Y)$ for some fixed Y in our setting. Wikipedia contributors (2022) provides a highly accessible introduction to AD, including most of the topics discussed here.

It is now convenient to redefine $z = h(x,Y)$ to be an $\mathbb{R}^m$-valued vector of outputs as a function of the inputs x and random element Y. Thus, we now consider multiple outputs instead of the special case $m = 1$ assumed till now. We thus seek the Jacobian of h, giving a matrix of partial derivatives of each component of z with respect to each component of x. The Jacobian is important in simulation optimization, for example, when both the objective function and some constraint functions must be estimated.

The two most common methods for AD calculation of the Jacobian of a composite function are forward and reverse mode differentiation, both of which calculate the same numerical value as the IPA estimate but do so in different ways. Both methods work with the same code used to compute $h(x,Y)$. Forward mode

applies the chain rule from the "inside out," calculating and carrying forward derivatives of each intermediate variable with respect to each component of x as the original program runs. Reverse mode recursively works from the "outside in," i.e., from the final simulation output z back towards the simulation input x, using the chain rule after the initial program has finished execution, requiring access to the values of all intermediate variables from the forward pass. While it is possible to use a combination of forward and reverse mode in different parts of the program to reduce the amount of computational work, finding the approach that minimizes computation is an NP-complete problem (Naumann 2008) and thus is rarely pursued in practice. When implemented well, AD of a function has a computational cost that is on the order of the cost to evaluate the function. Forward mode has constant memory complexity and computational complexity which scales with the number of inputs. On the other hand, reverse mode has memory complexity which scales with the number of intermediate variables and computational complexity which scales with the number of outputs (van Merriënboer et al. 2018). Thus forward mode is typically preferred to reverse mode when there are more outputs than inputs and vice versa. In the simulation-optimization setting where the number of outputs m is typically small or even 1, reverse mode is preferred.

Reverse mode must define a way to construct the chain rule to work backwards from the output to the input. We summarize the two main approaches which are discussed in more detail in Chapter 6 of Griewank and Walther (2008) and whose practical tradeoffs are discussed in van Merriënboer et al. (2018). The first is operator overloading, which overloads the functions in the original program to build an instance-dependent computational graph on the primitive functions used in the forward pass. This makes it easy to handle control-flow operations because the computational graph is static for a fixed instance, making the backward pass equivalent to there being no control-flow operations in the original program. A disadvantage of this approach is that building the computational graph each time the function is called incurs overhead, and since the graph can change with each instance, it is hard or inefficient to optimize the code that executes the backwards pass. The other approach is source code transformation which, from the source code of the program, explicitly constructs another program, called the adjoint program, to perform the backward pass based purely on the intermediate values stored during the forward pass. The advantage of this approach is that the adjoint program only has to be constructed once and thus incurs no overhead at run time, other than potentially on the first replication. The program for the backward pass can also be constructed prior to run time and can be optimized and compiled prior to run time. The disadvantage is that implementing the transformation rules for certain function calls and control-flow operations to construct the adjoint program can be complex. This can also lead to long compilation times of the adjoint program.

3 CONSIDERATIONS FOR AUTOMATIC-DIFFERENTIATION TOOLS IN SIMULATION

Several factors impact the utility of AD tools for simulation:

- The tool must be able to automatically generate the derivatives of primitive functions, such as $\sin x$ and e^x, that are used in the code to compute $h(x,Y)$. Some tools, such as Autograd and JAX, have mimicking implementations of functions from popular libraries such as NumPy, so that code written in basic Python and NumPy can be differentiated with minimal changes to the code.
- Some AD tools require code to be restructured to address complications posed by control-flow operators such as `for` and `while` loops and `if` statements.
- Some AD tools restrict the types of data structures that can appear as inputs and outputs of the function to be differentiated.
- The run time of the AD tool can be important because simulations may require nontrivial computation.

The performance (run time) of AD tools depends on the structure of the software that is being differentiated. Certainly, ML programs differ in structure from simulation programs. The control flow of the forward pass of a neural network tends to be relatively simple. Deep neural networks can have millions of inputs but often only have a few outputs, which largely explains the popularity of a specialization of

reverse mode to the neural-network setting known as back propagation. Furthermore, the majority of the computational effort of a forward or backward pass through a deep neural network is typically due to large matrix operations. Thus, the overhead incurred by constructing the per-instance computational graph or the adjoint program for the backwards pass can be negligible relative to the matrix computations. Differences in implementations of matrix operations can therefore account for much of the performance differences between AD software tools in ML. See Baydin et al. (2017) for a more thorough discussion of factors impacting the run times seen in practice of AD tools in machine learning applications. On the other hand, in discrete-event simulation, the program used to compute the function $h(x,Y)$ can have very complicated control flow with many conditional statements and loops. There are often a large number of sequential, non-vectorizable operations involved in performing a single replication of a simulation. Thus, the overhead of the AD software is more likely to have a noticeable impact on the total compute time of the program than in standard ML applications. In simulation, the ratio of inputs to outputs varies but is typically much smaller than in ML. These differences in the typical program structures in ML and discrete-event simulation may cause some AD software to perform better in one application than in another.

Some AD tools perform exceptionally well in specific situations. Zygote (Innes et al. 2019) is a promising source-transformation-based AD tool for Julia that claims to be able to work with any data structure, trace through mutation, and thus differentiate any (differentiable) function in Julia, including those from other libraries not specially designed to work with Zygote. Enzyme (Moses and Churavy 2020) is another source transformation AD tool that works with any language whose compiler compiles the source code to the LLVM intermediate representation including C, C++, Fortran, Julia, Rust, Swift, MLIR, and more. We now focus on Autograd and JAX due to their compatibility with Python and implementation of NumPy-like functions as primitives.

3.1 Autograd

Autograd is an AD library in Python that uses reverse mode via operator overloading. Primitive functions include basic Python functions like addition, NumPy-like functions like sin, its own AD functions, and the ability to define your own primitive functions. The `autograd.value_and_grad` function returns the value and gradient of the function to which it is applied, while `autograd.grad` returns just the gradient. The function to which `autograd.value_and_grad` or `autograd.grad` is applied must have a scalar output $(m = 1)$ but is allowed to have multiple inputs $(d \geq 1)$ with various data structures such as scalars, lists, arrays, and dictionaries supported. The `autograd.jacobian` function only supports taking Jacobians where the input x and output z are either scalars or arrays. Further, all inputs x for which derivatives in the Jacobian are desired must be passed into the first argument of the function being differentiated. Therefore, if x is multidimensional, then its values must be concatenated into a single array. Similarly, if the output z is multidimensional, then the function being differentiated should return a single array containing those outputs. There does not exist an `autograd.value_and_jacobian` function that returns both the output and Jacobian in a single pass, but such a function can be easily written by modifying the code of `autograd.jacobian`. Another downside is that Autograd is not able to trace through certain operations such as in-place operations, commonly encountered when assigning new values to only part of an array, because these modify the intermediate variables. However, lists can be used freely and converted to arrays to perform calculations. More detail on what is supported and what workarounds exist are available on the tutorial page of the Autograd GitHub repository (Maclaurin et al. 2015).

3.2 JAX

JAX is another AD library in Python that combines an updated version of Autograd with the option of just-in-time (JIT, discussed below) compilation. JAX is capable of both forward and reverse mode. The implementation of reverse mode uses operator overloading. As primitives, JAX also has basic Python functions, NumPy like functions, its own AD functions, the ability to define custom primitives, and some

additional utility functions. Like Autograd, JAX has `jax.grad`, `jax.value_and_grad`, `jax.jacfwd`, and `jax.jacrev` functions. The `jax.grad` and `jax.value_and_grad` functions are used in the same way as Autograd's versions. There are two versions of a Jacobian function to support forward (`jax.jacfwd`) and reverse mode (`jax.jacrev`) differentiation. These functions can be used the same way as `autograd.jacobian`, but they also allow inputs and outputs to be data structures other than scalars or arrays, such as lists and dictionaries. JAX provides a functional interface for array-element assignment and control flows, such as loops, to allow differentiation and JIT compilation as discussed in Section 4.

JAX stands for "just after execution," which refers to JAX being capable of JIT compilation. Our discussion of JIT compilation is mostly adapted from Aycock (2003), to which we refer readers seeking more information. JIT compilers dynamically translate source code or bytecode to optimized machine code at run time. JIT compilers can provide the benefits of both static compilers and interpreters. While an overhead is paid at run time for the translation, the cost can be small relative to the cost of executing computationally heavy programs like long simulations or large numerical algebra tasks, and thus the speed can be similar to statically compiled programs. In contrast to static compilers, JIT compilers can recompile different pieces of code during run time if, for example, a new data type is encountered. This allows more flexibility in the source code than static compilers provide, yet not as much flexibility as interpreters. To JIT compile a function with JAX, the function must be pure-and-statically-composed, i.e., the function must only act on the inputs to the function and all changes must be passed as an output (as opposed to silently altering the state of an object) and must have a static computational graph on the primitive functions of JAX. A `while` loop is an example of a common component of simulation programs which violates this condition as the number of iterations, and thus the computational graph, varies with the input. This is why JAX provides a function which mimics the behavior of a `while` loop.

JIT compilation is not the only tool JAX provides to accelerate program execution. It also provides `vmap` and `pmap` functions which can somewhat automatically vectorize and parallelize code, respectively. Additionally, JAX provides GPU/TPU support providing further speedups in some cases. For more details regarding JAX, we refer the reader to the JAX GitHub repository (Bradbury et al. 2018).

4 ADAPTING SIMULATION CODE FOR AUTOGRAD AND JAX

Discrete-event simulation programs commonly feature pseudo-random number generation, custom written objects, event lists, loops and conditional statements. Depending on their implementation, these ingredients may cause issues when Autograd or JAX is used for AD. We discuss potential issues and solutions.

4.1 Pseudo-Random Number Generators

If x is a purely structural parameter, then there is no need to differentiate the code of the random number generator (RNG), but if any components of x are parameters of the distribution of Y, then we must differentiate the code of the RNG. Although both Autograd and JAX without JIT compilation (henceforth written JAXnoJIT) allow the use of any RNG, these software tools may not be able to differentiate through the RNG depending on the RNG's implementation. Indeed, many RNGs will not be compatible with JAX with JIT compilation (henceforth written JAXJIT) because the RNG may not be functionally pure. For example, an RNG object that outputs a number based on some internal state which gets updated each time the RNG is called would not be compatible with JAXJIT. This lack of compatibility arises because the output of the call to the RNG depends on the RNG's (internal) state which is neither static nor an input of the function being called. JAX's `jax.random` module revolves around a *key* object which acts as a state. This key object must be passed as an argument whenever a function is called to generate a random variate. For example, a sample from a Bernoulli(0.5) distribution could be drawn by calling `jax.random.bernoulli(key, p=0.5)`. The key must then be manually tracked and updated between its uses. Any generation of random numbers used with JAXJIT must have a similar functional structure built around the use of a key or seed.

4.2 Object-Oriented vs. Functional Programming

As discussed earlier, Autograd and JAX can compute the Jacobian of a function $z = h(x,Y)$ for a fixed Y. In the code, it is important that x is an input and z is an output in the function signature. For example, in an object-oriented simulation program, a `model` object may contain the simulation logic in a method `model.simulate()` that does not take in any inputs, but instead internally accesses the simulation model's parameters via an attribute `model.parameters`. This code architecture makes it impossible for Autograd or JAX to differentiate `model.simulate()`, because the method has no explicit inputs. To make the code differentiable, the `model.simulate()` method would need to be rewritten to accept the model's parameters as an explicit input, e.g., `model.simulate(model.parameters)`.

4.3 Event Lists

Simulation models are often implemented using an event-driven world view. If the event list is implemented as a Python list, then retrieving, adding, and removing events from the event list are likely compatible with Autograd and JAXnoJIT without any modification. However, if the event list is implemented as a NumPy array, then the simplest way to get the event-list manipulations to be compatible with Autograd and JAXnoJIT is to switch it to a list. To use JAXJIT, the event list should be implemented as a JAX array. JAX arrays are also compatible with JAXnoJIT. When using a JAX array, it is necessary to use the functions JAX provides for retrieving and setting elements of the event list such as `arr.at[].get()` or `arr.at[].set()`. If the simulation requires taking a dynamic-sized slice of the event list, i.e., the size depends on values not known until run time, one must instead use a functional approach yielding a static-sized slice. This is due to the requirements of the accelerated-linear-algebra compiler needing to know how much memory to allocate to the array regardless of the input. An example of a functional solution is given in Figure 2.

4.4 Loops and Conditionals

Standard Python loops and conditional statements are compatible with Autograd and JAXnoJIT. However, JAXJIT requires any loops or conditional statements to be replaced by functional equivalents, which can be found in the `jax.lax` module. Examples of converting a `for` loop and `if` statement are given in Figure 2. Additionally, if JAXJIT is used to differentiate a simulation containing the JAX functional equivalent of a `while` loop, then reverse mode will not be possible. This is again due to the compiler requiring a static bound on the size of the array needed to store the intermediate variables for the backwards pass. If a simulation requires a `while` loop and JAXJIT is used, forward mode is the only option.

4.5 An Example

Figure 1 gives code for a function that is used in an ambulance simulation in Section 6. This function takes in an event list and the number of events in the list and then returns the next event, the newly updated event list with the next event deleted, and the updated number of events in the list decremented by 1. The event list is unsorted, the first `num_events` elements are events in the list, and the rest of the list contains some dummy `place_holder_event`. Each event is itself a list containing information about the event such as the time at which it occurs and the type of the event. Thus, `event_list` is a list of lists. This code is compatible with both Autograd and JAXnoJIT, but not with JAXJIT.

We now provide a modified version of the code, as seen in Figure 2, as an example of adapting code to be JAXJIT compatible. This modified code assumes that `event_list` is a two dimensional JAX array with each event itself being a JAX array.

The code in Figure 1, which is compatible with Autograd and JAXnoJIT, looks indistinguishable from typical code, except perhaps that we use a list of lists instead of an array or custom defined object for our event list. In contrast, in Figure 2, which is compatible with JAXJIT, the code had to be restructured:

```
import autograd.numpy as np # or import jax.numpy as np if using JAX

def get_next_event(event_list, num_events):
   # initialize next event time to essentially infinity
   mini = SIM_LENGTH + 2.0
   # search through events to find which one has earliest event time
   for i in range(num_events):
      if event_list[i][0] < mini:
         mini = event_list[i][0]
         next_event_index = i
   next_event = event_list[next_event_index]
   # delete next_event from event_list
   event_list[next_event_index:num_events+1] = event_list[next_event_index+1:num_events+2]
   num_events -= 1
   return event_list, num_events, next_event
```

Figure 1: Event list as an unordered list of lists, compatible with Autograd and JAX without JIT compilation.

1. The `for` loop had to be replaced with a functional equivalent.
2. Though not shown, we also have to write the `get_next_event_body` function that carries out the logic inside each iteration of the `for` loop to update `mini` and `next_event_index` as needed.
3. This function itself has an `if` statement which must be replaced by a functional equivalent that, in turn, also requires the writing of two more functions to handle each Boolean value of the conditional.
4. Each of these functional equivalents require us to write additional functions to pass to them as arguments.
5. The `.at[]` and `.get()` notation is needed to index and retrieve values from arrays.
6. When updating `event_list` in Figure 1, we accessed a slice of the `event_list` that depended on the run time value `num_events`. In Figure 2 we use `np.where` instead.

5 INCORPORATING AUTOMATIC DIFFERENTIATION INTO SIMOPT

SimOpt is a Python library intended for benchmarking the finite-budget performance of simulation-optimization solvers and understanding their behavior on a variety of problems. In light of this goal, SimOpt retains a simple enough architecture so that researchers from varied backgrounds can easily contribute problems and solvers that are of interest to them. Furthermore, the motivation for incorporating AD into SimOpt is to reduce the burden on contributors by automating the implementation of IPA estimators. Thus, the decision of which AD tool to use in SimOpt is based on ease of use followed by reasonable (run time) performance. As SimOpt already heavily utilizes `NumPy`, it is convenient to choose an AD tool that includes NumPy-like function primitives. These considerations led us to consider Autograd and JAX and rule out other options such as PyTorch. We ruled out JAXJIT as an option because it would require an extensive restructuring of the SimOpt library. Between Autograd and JAXnoJIT, we chose to implement Autograd because the performance is better in two representative examples; see Section 6.

In the SimOpt architecture, a simulation-optimization problem is specified by two objects: the `Model` class which represents the simulation model and the `Problem` class which formulates the optimization problem built around the model. The parameters of the simulation model are stored in a dictionary attribute `Model.factors`. The `Model` class also has a `replicate()` method that returns dictionaries with stochastic values of `responses` and `gradients` as the outputs of the simulation. The `Model.replicate()` method does not have any explicit inputs but instead accesses the simulation model parameters internally through `Model.factors`. The `Problem` class defines decision variables as a function of `Model.factors` and a performance metric as a function of the `responses` returned

```python
from jax import jit
from jax.lax import cond, fori_loop
import jax.numpy as np

@jit
def get_next_event(event_list, num_events):
    mini = SIM_LENGTH + 2.0
    i = 0
    next_event_index = 0
    # loop below is to find index of next event
    _, _, next_event_index = fori_loop(0, num_events, get_next_event_body,
                             (event_list, mini , next_event_index))
    next_event = event_list.at[next_event_index].get()
    place_holder_event = np.zeros((EVENT_SIZE))+(SIM_LENGTH+1)
    # deleting next_event from event_list
    event_list = np.where(np.arange(len(event_list)) < next_event_index,
                  np.vstack([event_list[:-1], place_holder_event]).T,
                  np.vstack([event_list[1:], place_holder_event]).T).T
    num_events -= 1
    return event_list, num_events, next_event
```

Figure 2: Event list as an unordered JAX array of JAX arrays, compatible with JAX with JIT compilation.

from the `Model.replicate()` function. Currently, we have only implemented AD to calculate the `Model.replicate()` Jacobian associated with the `Model.factors` and `responses`. At this time, the Jacobian of the mapping from decision variables of `Problem` to `Model.factors` and the gradient of the mapping from `responses` to the final performance metric of `Problem` must be computed manually and multiplied with the automatically obtained Jacobian of `Model.replicate()` to complete the gradient calculation of the performance metric with respect to the decision variables. We intend to eventually extend AD to the `Problem` class so that the entire gradient calculation is handled automatically.

To enable automatic computation of the Jacobian of the `responses` with respect to the `factors`, we created a subclass of `Model` named `Auto_Model`, which must have an `Auto_model.inner_replicate()` method. The `Auto_model.inner_replicate()` method is essentially the same as `Model.replicate()` but with a different signature and must follow the general rules to be compatible with Autograd so that we can use Autograd to differentiate it. Thus, it requires little to no additional work for the programmer to write when compared to `Model.replicate()`. The `Auto_Model` constructor automatically creates an `Auto_model.replicate()` method by applying the `replicate_wrapper()` method that we wrote to `Auto_model.inner_replicate()`. The `replicate_wrapper()` method converts dictionaries to arrays for the inputs and vice versa for the outputs. It also calls the `value_and_jacobian()` method we wrote applied to `Auto_model.inner_replicate()` to get its output and Jacobian. The method `value_and_jacobian()` is not included in Autograd but can easily be written by modifying the code for `autograd.jacobian()` to return both the function output and Jacobian instead of just the Jacobian. We allow the calculation of a partial Jacobian with respect to only a subset of outputs, saving computation since `value_and_jacobian()` uses reverse mode whose run-time complexity scales with the number of outputs. A user can write a subclass of `Auto_Model` instead of just `Model`, requiring comparable amounts of work, to automatically get an IPA Jacobian estimate out of their simulation model when they call `replicate()` instead of having to manually code the IPA Jacobian estimate.

While incorporating Autograd into SimOpt, we inevitably encountered bugs. Most were caused by using functions or arrays from `numpy` instead of `autograd.numpy` inside of a function being differentiated. For example, the `Auto_model.inner_replicate()` function typically makes calls to SimOpt's custom RNG which is defined in a separate file from model definitions. It was necessary to change `import`

Table 1: SAN results, with AD applied after each replication then averaged. The entry N/A indicates that the run did not complete in reasonable time.

Reps	Manual (REV)	FFD	Autograd	JAX	JAX JIT 1st	JAX JIT 2nd
1	996 (μs)	5.92 (ms)	3.46 (ms)	1.52 (s)	1.38 (s)	185 (μs)
10^2	9.27 (ms)	33.9 (ms)	283 (ms)	5.54 (s)	1.88 (s)	509 (μs)
10^4	762 (ms)	5.71 (s)	16.1 s (s)	7 (min) 42(s)	1.60 (s)	10.1 (ms)
10^6	33.5 (s)	4 (min) 1 (s)	19 (min) 55 (s)	N/A	1.48 (s)	646 (ms)

`numpy as np` to `import autograd.numpy as np` in the file where the RNG was defined. Another source of bugs is in-place updating of arrays. These are easily fixed by using lists instead of arrays until all in-place operations are complete, at which point an array can be constructed from the list for computations if needed. Finally, it is possible for Autograd to unexpectedly return 0 as a value for a derivative when this is not the desired output. This is because Autograd may not be able to trace the output as being dependent on the inputs if the rules of compatibility with Autograd are broken. Therefore, if a value of 0 is encountered as a derivative value, it may be good practice to use finite differences to verify whether 0 is indeed the correct value.

6 EXPERIMENTS

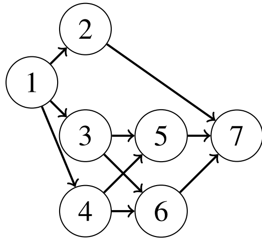
We explore the run times of AD software on simulation models of a stochastic activity network (SAN) and an ambulance service. We time each method of obtaining an IPA gradient estimate using a varying number of replications by averaging over replications. AD can be applied at two places in the code: after each replication or after averaging over all replications, and we shall see that the run-time difference is large. For the SAN we include both methods, but for the ambulance system we perform only the latter. We ran JAXJIT twice for each number of replications to demonstrate the time spent on the initial program compilation versus executing the program after compilation. All AD tools used reverse mode except JAXJIT in the ambulance system where forward mode was required due to the inclusion of a `while` loop in the code. We also manually implemented IPA. For the manual implementations, we used reverse mode for the SAN and forward mode for the ambulance system. We also used forward finite differences (FFD) for both models. The timing experiments were performed in Google Colab using two Intel Xeon CPU 2.20GHz cores.

6.1 Stochastic Activity Network

A SAN is a feed-forward network that models the precedence of a collection of subtasks for completing a task. Each arc represents a subtask which we assume takes an exponentially distributed amount of time to complete. Figure 3 depicts the specific SAN we used. The parameters of the simulation model, $x \in \mathbb{R}^{10}$, are the mean times of each subtask. The output of the simulation model is the time to complete the total task, which is the (random) length of the longest path from node 1 to node 7, plus a deterministic cost $\sum_{i=1}^{10} x_i^{-1}$ so that $h(x,Y) = Y + \sum_{i=1}^{10} x_i^{-1}$, where $Y = Y(x)$ is the length of the longest path from node 1 to 7. Table 1 gives the average per-replication time to obtain a derivative when AD is applied to each replication individually. Figure 3 gives the corresponding results when AD is applied once to the entire body of replications collectively.

6.2 Ambulance System

We use the model described in Eckman and Henderson (2020) but with modified parameters. The following model description is taken, almost verbatim, from that reference. Calls arrive according to a Poisson process in time at rate $1/2.5$. Call locations are uniformly distributed in a square with side length 20. The closest free ambulance is dispatched to the call, and travels to the call in Manhattan fashion (traveling first in the x direction, followed by the y direction) at a constant speed of 1. The ambulance then spends a random



Reps	Autograd	JAX	JAX JIT 1st	JAX JIT 2nd
1	4.16 ms	2.23 s	1.71 s	6.38 ms
10^2	562 ms	8.67 s	2.81 s	7.81 ms
10^4	26.4 s	13 min 17 s	2.53 s	47.5 ms
10^6	N/A	N/A	3.26 s	680 ms

Figure 3: Left: The SAN we used in our experiments. Right: SAN results with AD applied after averaging all replications. N/A results are due to running out of memory.

Table 2: Ambulance system timing results, with AD applied after each replication and then averaged.

Reps	Manual (FWD)	FFD	Autograd	JAX	JAX JIT 1st (FWD)	JAX JIT 2nd (FWD)
1	24.2 ms	46.8 ms	61.6 ms	3.08 s	4.86 s	23 ms
10^1	100 ms	552 ms	643 ms	9.01 s	6.60 s	168 ms
10^2	837 ms	4.19 s	4.53 s	40.7 s	5.26 s	1.57 s
10^3	9.82 s	30.2 s	30.5 s	4 min 59 s	19.5 s	15.8 s

amount of time at the scene of the call (exponentially distributed with mean 10) and afterwards returns to its base in Manhattan fashion. For simplicity, hospitals are not modeled, and ambulances returning to base are not diverted to new calls. Calls arising when all ambulances are busy are queued and answered in first-in-first-out order. Call arrivals, locations, and scene times are mutually independent. The simulation starts from both ambulances being available and at base, with no queued calls. The variable x specifies the locations of the two ambulance bases, which are located at $(15,15)$ and $(5,5)$; thus, $x=(15,15,5,5)$. The output $h(x,Y)$ is the sum of the ambulance response times to calls over a time interval of length 100. This simulation uses an event list and has a while loop that keeps the simulation running until time 100. Timing results for AD on this ambulance system simulation are reported in Table 2.

7 DISCUSSION AND CONCLUSION

The SAN experiment suggests that it is beneficial to apply AD after each replication rather than after averaging all replications due to faster run times and lower memory requirements. Moreover, access to per-replication IPA estimates allows the computation of sample variances of the IPA estimates. The memory issue when applying AD after averaging all replications is expected, because the number of intermediate variables stored for the backwards pass then scales linearly with the number of replications. We did not run into memory issues when AD was applied to each replication. Autograd appears to run faster than JAXnoJIT. JAXJIT can provide substantial speedups over Autograd on larger scale tasks but requires significant effort and care to restructure the code to be fully functionally oriented with static-sized intermediate variables. JAXJIT is limited to forward mode when a while loop is present in the code, as in many event-list based simulations. This may narrow the run-time gap when compared to Autograd in such simulations as seen in the ambulance example. The compile time of JAXJIT dominates the actual program execution time for small programs. The run times when JAX is run a second time after already applying JAXJIT from the first run are faster than the first time because the overhead for JIT compilation is not repeated.

The manual method was faster than the AD methods in all comparisons, except with JAXJIT on the SAN example for large numbers of replications. JAXJIT was faster than FFD for large numbers of replications. Autograd was slower than FFD on the SAN problem, but typically on the same order of magnitude, whereas the two methods had similar run times on the ambulance problem. Our experiments were not designed to show the scaling of the run times with the dimension, d, however, FFD should scale linearly with d because $d+1$ simulation runs are needed to estimate the gradient. We expect that reverse-mode manual

and AD methods will remain approximately constant in d. Thus, for larger d we expect that Autograd will be faster than FFD. We did not use GPU/TPU support or the `vmap` or `pmap` functions of JAX, which could significantly speed up JAX and JAXJIT for some models.

AD can give yield IPA estimates on non-trivial simulations without requiring manual derivation and coding of the IPA estimator, with a competitive run time compared to FFD. In addition to saving analyst effort, getting an IPA estimate from AD is likely less error prone than manually deriving and implementing the IPA estimator. The use of existing AD tools requires some care in coding, but for many models is straightforward to implement. Additionally, IPA has different bias and variance properties than FFD which may be advantageous in some settings. We look forward to developments in AD tools that improve their utility in simulation applications, perhaps through careful consideration of common flow structures in discrete-event simulation models.

ACKNOWLEDGMENTS

This work was partially supported by National Science Foundation grant CMMI 2035086.

REFERENCES

Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. 2015. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems". https://www.tensorflow.org/, accessed 28th April 2022.

Asmussen, S., and P. W. Glynn. 2007. *Stochastic Simulation: Algorithms and Analysis*, Volume 57 of *Stochastic Modeling and Applied Probability*. New York: Springer.

Aycock, J. 2003. "A Brief History of Just-in-Time". *ACM Computing Surveys* 35(2):97–113.

Baydin, A. G., B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. 2017. "Automatic Differentiation in Machine Learning: A Survey". *Journal of Machine Learning Research* 18(1):5595–5637.

Bradbury, J., R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. 2018. "JAX: Composable Transformations of Python+NumPy Programs". http://github.com/google/jax, Version 0.2.5, accessed 28th April 2022.

Cao, X.-R. 1985, Sep.. "Convergence of Parameter Sensitivity Estimates in a Stochastic Experiment". *IEEE Transactions on Automatic Control* 30(9):845–853.

Cui, Z., M. C. Fu, J.-Q. Hu, Y. Liu, Y. Peng, and L. Zhu. 2020. "On the Variance of Single-Run Unbiased Stochastic Derivative Estimators". *INFORMS Journal on Computing* 32(2):390–407.

Eckman, D. J., and S. G. Henderson. 2020. "Biased Gradient Estimators in Simulation Optimization". In *Proceedings of the 2020 Winter Simulation Conference*, edited by K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, 2935–2946. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Eckman, D. J., S. G. Henderson, R. Pasupathy, and S. Shashaani. 2020. "Simulation Optimization Library". http://github.com/simopt-admin/simopt, accessed 28th April 2022.

Ford, M. T. 2022. "Experiments with Automatic Differentiation for Stochastic Simulation". http://github.com/fordmatt18/Autodiff_for_Sim, accessed 28th April 2022.

Fries, C. 2017, 01. "Fast Stochastic Forward Sensitivities in Monte-Carlo Simulations Using Stochastic Automatic Differentiation (with Applications to Initial Margin Valuation Adjustments (MVA))". *SSRN Electronic Journal*.

Fries, C. 2019, 01. "Stochastic automatic differentiation: automatic differentiation for Monte-Carlo simulations". *Quantitative Finance* 19:1–17.

Fu, M., S. Andradottir, J. Carson, F. Glover, C. Harrell, Y.-C. Ho, J. Kelly, and S. Robinson. 2000. "Integrating Optimization and Simulation: Research and Practice". In *Proceedings of the 2020 Winter Simulation Conference*, edited by J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, 610–616. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Fu, M. C. 2015. "Stochastic Gradient Estimation". In *Handbook of Simulation Optimization*, edited by M. C. Fu, Volume 216 of *International Series in Operations Research & Management Science*, Chapter 5, 105–148. Springer New York.

Fu, M. C., and J.-Q. Hu. 1997. *Conditional Monte Carlo: Gradient Estimation and Optimization Applications*. Boston: Kluwer.

Giles, M., and P. Glasserman. 2006. "Smoking Adjoints: Fast Monte Carlo Greeks". *Risk* 19(1):88–92.

Glasserman, P. 1991. *Gradient Estimation Via Perturbation Analysis*. The Netherlands: Kluwer.

Glasserman, P. 2013. "Adjoints and Averaging". In *Stochastic Simulation Optimization for Discrete Event Systems: Perturbation Analysis, Ordinal Optimization, and Beyond*, 63–73. Singapore: World Scientific.

Glynn, P. W. 1990. "Likelihood Ratio Gradient Estimation for Stochastic Systems". *Communications of the ACM* 33(10):75–84.

Gong, W.-B., and Y.-C. Ho. 1987. "Smoothed (Conditional) Perturbation Analysis of Discrete Event Dynamical Systems". *IEEE Transactions on Automatic Control* 32(10):858–866.

Griewank, A. 1989. "On Automatic Differentiation". In *Mathematical Programming: Recent Developments and Applications*, edited by M. Iri and K. Tanabe, 83–108. Dordrecht, The Netherlands: Kluwer Academic Publishers.

Griewank, A., and A. Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Philadelphia: SIAM.

Innes, M., A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt. 2019. "A Differentiable Programming System to Bridge Machine Learning and Scientific Computing". https://arxiv.org/abs/1907.07587, accessed 28th April 2022.

Kucukelbir, A., D. Tran, R. Ranganath, A. Gelman, and D. M. Blei. 2017. "Automatic Differentiation Variational Inference". *Journal of Machine Learning Research* 18(1):430–474.

Maclaurin, D., D. Duvenaud, and R. P. Adams. 2015. "Autograd: Effortless Gradients in Numpy". In *ICML 2015 AutoML Workshop*. July 11th, Lille, France.

Moses, W., and V. Churavy. 2020. "Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients". In *Advances in Neural Information Processing Systems 33*, edited by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, 12472–12485. Red Hook, New York: Curran Associates, Inc.

Naumann, U. 2008, Apr. "Optimal Jacobian Accumulation is NP-Complete". *Mathematical Programming* 112(2):427–441.

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In *Advances in Neural Information Processing Systems 32*, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, 8024–8035. Red Hook, New York: Curran Associates, Inc.

van Merriënboer, B., O. Breuleux, A. Bergeron, and P. Lamblin. 2018. "Automatic Differentiation in ML: Where We Are and Where We Should Be Going". In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, edited by S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, and N. Cesa-Bianchi, 8771–8781. Red Hook, New York: Curran Associates, Inc.

Wikipedia contributors 2022. "Automatic Differentiation — Wikipedia, The Free Encyclopedia". "https://en.wikipedia.org/wiki/Automatic_differentiation", accessed 28th April 2022.

Williams, R. J. 1992. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". *Maching Learning* 8(3):229–256.

AUTHOR BIOGRAPHIES

MATTHEW T. FORD is a Ph.D. Student in the School of Operations Research and Information Engineering at Cornell University. His research interests include simulation optimization and decision making under uncertainty. He is a contributor to SimOpt, a testbed of simulation optimization problems and solvers. His web page is http://fordmatt18.github.io/ and his email address is mtf62@cornell.edu.

DAVID J. ECKMAN is an Assistant Professor in the Wm Michael Barnes '64 Department of Industrial and Systems Engineering at Texas A&M University. His research interests deal with optimization and output analysis for stochastic simulation models. He is a co-creator of SimOpt, a testbed of simulation optimization problems and solvers. His e-mail address is eckman@tamu.edu.

SHANE G. HENDERSON is the Charles W. Lake, Jr. Chair in Productivity in the School of Operations Research and Information Engineering at Cornell University and a Fellow of INFORMS. His research interests include simulation optimization, emergency services and selected transportation problems. He is a co-creator of SimOpt, a testbed of simulation optimization problems and solvers. His web page is http://people.orie.cornell.edu/shane and his email address is sgh9@cornell.edu.