`auto_diff`: AN AUTOMATIC DIFFERENTIATION PACKAGE FOR PYTHON

Parth Nobel

Department of Electrical Engineering and Computer Science
University of California, Berkeley
545D Cory Hall
Berkeley CA, USA
parthnobel@berkeley.edu

ABSTRACT

We present `auto_diff`, a package that performs automatic differentiation of numerical Python code. `auto_diff` overrides Python's NumPy package's functions, augmenting them with seamless automatic differentiation capabilities. Notably, `auto_diff` is non-intrusive, *i.e.*, the code to be differentiated does not require `auto_diff`-specific alterations. We illustrate `auto_diff` on electronic devices, a circuit simulation, and a mechanical system simulation. In our evaluations so far, we found that running simulations with `auto_diff` takes less than 4 times as long as simulations with hand-written differentiation code. We believe that `auto_diff`, which was written after attempts to use existing automatic differentiation packages on our applications ran into difficulties, caters to an important need within the numerical Python community. We have attempted to write this paper in a tutorial style to make it accessible to those without prior background in automatic differentiation techniques and packages. We have released `auto_diff` as open source on GitHub.

Keywords: automatic differentiation, numerical methods, python, library, implementation.

1 INTRODUCTION

Computing derivatives is critical in simulation. Many systems requiring simulation — circuits, chemical reactions, Newtonian mechanical systems, *etc.* — are naturally modeled as systems of Differential Algebraic Equations (DAEs). First-order derivatives are essential for any numerical algorithm that involves solving linear or nonlinear equations, such as DC, transient, Harmonic Balance, Shooting, *etc.* (Roychowdhury 2009) (2nd-order derivatives — Hessians — are also needed for optimization algorithms). For linear equations $A\vec{x} - \vec{b} = \vec{0}$, the matrix A is the derivative. Nonlinear equations are typically solved iteratively, for example using the Newton-Raphson method, by solving a sequence of different linear equations. Moreover, typical applications involve large vector systems, *i.e.*, simulating them requires derivatives of vector functions of vector arguments, called *Jacobian matrices*.

There are many options to obtain derivatives of computer code. The age-old practice, especially before the development of modern object-oriented computer languages, was for the author of a model to manually derive and then write out code for the derivatives of the model. However, this process can be tedious and prone to errors. For example, the BSIM3 MOS transistor model had an undetected derivative bug in its production code that took a decade to discover (Wang et al. 2015). Another choice is symbolic differentiation (Guenter 2007), where derivative equations/code are generated automatically by a graph-based analysis of

the function code supplied. Symbolic generation of derivative code avoids the mistakes, and (with careful automated code optimization built in) can generate very efficient code. However, it is not best suited for code development in an interpretive environment like Python, where small snippets of code are typically written and debugged interactively before progressing on the next small block of code. Moreover, available implementations in Python, such as SymPy, a Computer Algebra System, cannot differentiate functions which contain control flow (such as if/elif/else), which are widespread in applications. It should be noted that hand-generated and symbolic derivatives are both "exact" (barring numerical errors due to finite precision arithmetic).

Another very natural approach is finite differences, *i.e.*, approximating the first-principles definition of a derivative: the input of a function $f(x)$ is perturbed by a small amount Δx, the original value (without perturbation) is subtracted, and the difference divided by the magnitude of perturbation. While this is very easy to implement, it can suffer from significant accuracy problems, and is also typically more expensive to compute than alternatives. Using too large a Δx moves away from the definition of a derivative (in which $\Delta x \to 0$). However, using values that are too small leads to large errors; since two almost-equal numbers are subtracted to find a much smaller number, errors in the low-order bits (due to cancellation in finite precision arithmetic) can become very significant. Moreover, choosing a good $\Delta\vec{x}$ becomes much more complicated when $\vec{x}$ and $\vec{f}(\vec{x})$ are size-n vectors, as is typical in applications. Compounding the accuracy issue, the function needs to be evaluated $n+1$ times to find the derivative, which is usually significantly more expensive than other approaches. Because of such issues, finite differences are typically strongly deprecated in application domains (but are invaluable for checking the correctness of other approaches). An elegant variant of finite differences for real-valued functions with real inputs is the Complex-Step Derivative Approximation (CSD) (Martins, Sturdza, and Alonso 2003) In CSD, Δx is *purely imaginary* and the derivative is approximated by $\mathrm{Im}\{f(x+\Delta x)/\Delta x\}$. Note that there is no subtraction involved, completely avoiding finite precision cancellation errors. Correctness is easily shown using Taylor expansions on $f(\cdot)$. However, the result computed is corrupted by errors proportional to the third, fifth and higher odd-degree derivatives of the function, and there can be practical issues propagating complex numbers through code written for real numbers.

Finally, a category of techniques called "automatic differentiation", which leverages operator overloading in modern object-oriented languages, combines the advantages of being "exact", being easy to use, and approaching the efficiency of hand- or symbolically-generated code. x is replaced by an object that contains not only a (real or complex) value, but also a vector of derivatives (with respect to any number of designated independent variables). Every mathematical operation involved in computing $f(x)$ (such as $+, -, \times, \div, \sin(x), e^x$, *etc.*) is overloaded to return a similar object containing the operation's value as well as the vector of derivatives of that value. The derivatives are computed by applying the chain rule of differentiation numerically. For languages in which variables and objects are not explicitly typed (like Python and MATLAB), automatic differentiation provides "non-intrusive" usability — *i.e.*, code implementing functions needs no changes to support finding derivatives. This is of great value in applications, particularly during interactive code development using interpretive languages. In Sec. 2 below, we provide concrete examples illustrating how automatic differentiation works.

There are many libraries available that implement automatic differentiation, but there are currently none in Python that are non-intrusive, *i.e.*, that "just work" without the need to alter derivative-unaware code. In Python, Google's JAX provides Autograd, an automatic differentiation package (Bradbury et al. 2018). Other Python offerings include the `ad` package and CasADi; the latter of which which makes no effort to implement the NumPy API, requiring models be designed to work in its framework (Lee 2013, Andersson, Gillis, Horn, Rawlings, and Diehl 2019). For C++, Sandia National Laboratory developed Sacado, an automatic differentiation library, which due to C++'s statically-typed nature requires some code changes for derivative computation (Phipps and Gay 2006). Other C++ offerings include ADOL-C and `gdouble`, the latter of which uses templating to greatly reduce its intrusiveness (Griewank, Juedes, and Utke 1996,

Melville, Moinian, Feldmann, and Watson 1993, Feldmann, Melville, and Moinian 1992).MATLAB has a number of offerings available: MAPP, MAD, and the valder package all provide non-intrusive implementations of automatic differentiation (Wang et al. 2015, Forth 2006, Neidinger 2010).

In this work, we present `auto_diff`, a new automatic differentiation package in Python. Unlike Autograd, `auto_diff` attempts to perfectly recreate NumPy's API. Autograd does not support array mutation; this limitation causes many valid NumPy programs to not work with Autograd. A simple example which works with `auto_diff` is presented in Sec. 3.4, along with a discussion of Autograd's other limitations. The `ad` package does not support allocating NumPy arrays in code that is unaware it is being differentiated. This limitation of `ad` causes it to fail on the same example function presented in Sec. 3.4. As CasADi does not attempt to implement the NumPy API, it cannot be used with existing code.

`auto_diff` uses forward automatic differentiation, rather than reverse automatic differentiation, to compute derivatives – the distinction is discussed in Sec. 3.4. One of the effects of this choice is that it becomes easy to compute derivatives of vector functions with multiple outputs seamlessly; this application requirement was our primary motivation for developing `auto_diff`. A key feature is that our package augments the widely-used NumPy package in Python in such a way that all relevant NumPy functions are overloaded. This makes `auto_diff` almost completely non-intrusive; indeed, NumPy users and code writers need not even be aware that their code will compute derivatives if real/complex arguments are replaced with our `VecValDer` automatic differentiation object. The package has been tested successfully on simulation code for electronics devices, for circuits and for simple mechanical systems. Initial benchmarking indicates that running a simulation with `auto_diff` instead of hand-written code takes at most 4 times as long—this degree of slowdown is typical for automatic differentiation packages. The code implementing the package is simple to understand and modify for neophytes, which we hope will help with open-source development.

The remainder of the paper is organized as follows. In Sec. 2, we explain how automatic differentiation works. In Sec. 3, we summarize key implementation details, particularly those relating to seamless augmentation of NumPy, touch on the differences between forward and reverse mode approaches, and discuss some shortcomings of Autograd that motivated this work. Finally, in Sec. 4, we validate `auto_diff` on electronic device, circuit and mechanical system examples and provide initial estimates of performance.

2 HOW AUTOMATIC DIFFERENTIATION WORKS

The chain rule from elementary calculus provides the core result of forward differentiation. Consider a function which can be computed in two steps, *i.e.*, by computing $y = g(x)$ and then $z = f(y)$. We can find $\frac{dz}{dx}$ by applying the chain rule to obtain $\frac{dz}{dx} = \frac{df}{dy}\Big|_{y=g(x)} \frac{dy}{dx}$, and since $\frac{dy}{dx} = \frac{dg}{dx}\Big|_{x}$, we can directly compute the derivative.

The trick to automatic differentiation is replacing x, which originally is just a floating point variable, with a new object that tracks both the value of x and its derivative 1. Similarly, y and z are replaced with objects that tracks their values and their derivatives. This relies on *operator overloading*, which allows custom versions of exponentiation, addition, multiplication, *etc.*, to be defined for custom datatypes (Corliss and Griewank 1993). Derivative calculations using the chain rule are implemented in the custom versions of these functions.

As an example, consider $g(x) = x^2$, $f(y) = \sin(y)$, and $x = 3$. Initially, `x.val = 3`; and we set `x.der = 1`, designating x as an independent variable. Then we compute y, obtaining `y.val = 9`, and because Python detects it is operating on our data type and not a float, it uses our overloaded code for the $\star$ operator to compute that `y.der = 2 * x.val * x.der = 2 * 3 * 1 = 6`. Similarly, to compute $\frac{dz}{dx}$, we

evaluate `z.val = sin(y.val)` and `z.der = cos(y.val) * y.der`. How exactly `*`, `sin`, *etc.*, are replaced with our own functions is described in Sec. 3.

In order to demonstrate how to apply this approach to vector-valued functions with vector inputs, consider the example $\vec{y} = \vec{g}(\vec{x}) = 2\vec{x}$, $\vec{z} = \vec{f}(\vec{y}) = A\vec{y} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \vec{y}$, and $\vec{x} = \begin{bmatrix} 5 \\ 10 \end{bmatrix}$. Again, we replace $\vec{x}$, $\vec{y}$, and $\vec{z}$ with new custom objects of type `VecValDer`, which contains its value, named `val`, and a 2×2 matrix representing its Jacobian, named `der`. Initially `x.val` $= \begin{bmatrix} 5, 10 \end{bmatrix}^\top$ and we set `x.der` $= I_2$, the identity matrix of size 2, denoting that the entries of $\vec{x}$ are independent variables. Next, we apply $\vec{g}$ and by overloading scalar multiplication, we obtain `y.val` $= \begin{bmatrix} 10 \\ 20 \end{bmatrix}$ and `y.der` $= 2$ `x.der` $= \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$. Finally, we apply $\vec{f}$, and by overloading matrix multiplication, we obtain `z.val` $= \begin{bmatrix} 30 \\ 20 \end{bmatrix}$ and `z.der` $= A$ `y.der` $= \begin{bmatrix} 2 & 2 \\ 0 & 2 \end{bmatrix}$.

In general, a `VecValDer z` of a vector $\vec{z}$ being differentiated with respect to $\vec{x}$ has the property that

$$\texttt{z.val} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix}, \qquad \text{and} \qquad \texttt{z.der} = \begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \frac{\partial z_1}{\partial x_2} & \cdots & \frac{\partial z_1}{\partial x_n} \\ \frac{\partial z_2}{\partial x_1} & \frac{\partial z_2}{\partial x_2} & \cdots & \frac{\partial z_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_m}{\partial x_1} & \frac{\partial z_m}{\partial x_2} & \cdots & \frac{\partial z_m}{\partial x_n} \end{bmatrix}.$$

We now will discuss how to define `VecValDer`, how to overload all the relevant NumPy functions, and some other details of the implementation.

3 `auto_diff` IMPLEMENTATION

From the previous section, we see that we need to make our own type, `VecValDer`, that needs to store both the value it represents and the corresponding derivative. We do so in the fields `val` and `der`, respectively, as we did in the previous section. The type must be usable exactly as the NumPy array stored in `val` is. Further, the type needs to track the independent variables we are differentiating with respect to. Finally, the type must allow the extraction of the Jacobian from `der`.

In order to emulate NumPy during automatic differentiation completely, we need to overload every function which returns or modifies a NumPy array to also perform derivative computations. This includes binary functions, unary functions, and functions that take no vector arguments, but return a new NumPy array. Examples of binary functions and operators, *i.e.*, those which take two arguments, include addition, subtraction, matrix multiplication, *etc.*. Examples of unary functions and operators (those taking one argument) include the sine, cosine, square root, and absolute value functions, as well as unary operators like $+$ and $-$. Examples of functions that take no vector arguments are the constructors for the NumPy `ndarray` class and include `np.ones`, `np.zeros`, `np.array`, `np.ndarray`, and many other similar functions.

In the remainder of this section, we describe how we create `VecValDer`. In Sec. 3.1, we outline how `VecValDer` stores its data and implements some important methods, such as indexing. In Sec. 3.2, we explain how `VecValDer` implements NumPy's functions. As part of this, we describe how to track which variables are independent, how to extract the Jacobian, and provide other important implementation details. We illustrate the use of our library with an example, and touch upon how the code is organized in Sec. 3.3. Finally, we provide a brief comparison against Autograd, including a small example on which Autograd fails, but which that `auto_diff` handles correctly.

```
class VecValDer(np.lib.mixins.NDArrayOperatorsMixin):
    __slots__ = 'val', 'der'

    def __init__(self, val, der):
        self.val = val
        self.der = der

    def __setitem__(self, key, value):
        if isinstance(value, VecValDer):
            self.val[key] = value.val
            self.der[key] = value.der
        else:
            self.val[key] = value
            self.der[key] =true_np.zeros(self.der[key].shape)

    def __getitem__(self, key):
        return VecValDer(self.val[key], self.der[key])

    def __array_ufunc__(self, ufunc, method, *args,**kwargs):
        if method == '__call__' and ufunc in _HANDLED_UFUNCS:
            return _HANDLED_UFUNCS[ufunc](*args, **kwargs)
        return NotImplemented
```

Figure 1: Implementation of `VecValDer` and some of its methods.

3.1 `VecValDer` Data Layout

As described above, we only have two fields in the class, `val` and `der`, which store the two arrays. We use `__slots__` to inform Python's optimizer that there are only two fields in this object, allowing for memory compression (Python Software Foundation 2019). The class declaration, along with its `__init__` function, is reproduced in Figure 1. The other methods will be described later. Note that we inherit the class `np.lib.mixins.NDArrayOperatorsMixin`, which will be described in Sec. 3.2.

The exact size and shape of `der` evolved during the development process. Initially, we supported differentiating column vectors with respect to column vectors. As in the example above, if we start with a column vector $\vec{x}$ of size m, we store it in `val` as a NumPy array and then create `der` as the identity matrix to indicate that each value in $\vec{x}$ is an independent variable. As we compute other dependent vectors, we have `der` store a matrix where each row is a row vector storing the gradient of an entry in `val`. As an example, consider an intermediate column vector $\vec{y}$ of size 5 being differentiated with respect to a vector, $\vec{x}$, of size 4. `val` has shape 5, while `der` has shape 5×4. The $(i, j)^{\text{th}}$ entry in `der` corresponds directly to $\frac{\partial y_i}{\partial x_j}$, the partial derivative of the i^{th} entry of $\vec{y}$ with respect to the j^{th} entry of $\vec{x}$.

We then added support for dependent objects beyond column vectors. Support for computing the derivative of *matrices* can be very useful, *e.g.*, for the outer product $\vec{x}\vec{x}^\top$ of a vector $\vec{x}$. We changed `der` to have one-more dimension then the array in `val`. For example, if `val` is a matrix A of shape 3×4 being differentiated with respect to a vector, $\vec{x}$, of size 5, then `der` is a 3-dimensional array of shape $3 \times 4 \times 5$. In this 3-dimensional array, after indexing on the first two dimensions, say to `[2,1]`, we have a vector with 5 entries that is the gradient of the corresponding entry in `val`, $\nabla_{\vec{x}} A_{2,1}$.

In the last iteration of defining `der`, we added support for differentiating with respect to any NumPy array. We do this by making the shape of `der` the concatenation of the shape of `val` and the vector we're differentiating with respect to. For example, if we differentiate a 3×4 matrix A with respect to a vector $\vec{x}$ of shape 5×1, then `der` must have shape $3 \times 4 \times 5 \times 1$. Indexing into this four dimensional array to position i, j, k, ℓ corresponds to the partial derivative $\frac{\partial A_{i,j}}{\partial x_{k,\ell}}$.

One of the benefits of this design decision is that it allows us to support slicing *i.e.*, writing `x[i:j]` to select multiple entries from an array, in a simple and clean manner, which generalizes easily to more complicated slicing. We can implement the `__getitem__` and `__setitem__` methods as reproduced in Figure 1, because the arrays have the same indexing in the initial dimensions. These methods are called when a user executes code like `x[3], x[1:5]`, or `x[4,0]` to access values or to assign values, respectively.

3.2 Overloading NumPy Functions

In this section, we first describe how we overload NumPy's functions which take arguments. Second, we explain how we replace the constructors of the `ndarray` class. As part of the discussion, we address a few loose ends, such as how we indicate which variables are independent.

NumPy has two types of functions that take arrays as arguments, *ufuncs* and what NumPy refers to as *functions*; however, as NumPy is currently trying to reimplement many functions as ufuncs, we will only describe how we handle ufuncs. Curious readers can rest assured that what NumPy calls functions are handled almost identically to ufuncs and that the documentation is fairly thorough (SciPy 2019a).

NumPy's ufuncs include most functions which operate element-wise, such as `np.negative`, `np.sin`, and `np.multiply`, as well as, some others such as `np.matmul`. A number of these operations, including `np.negative`, `np.multiply`, and `np.matmul`, have corresponding operators in Python; `-`, `*`, and `@` for our three examples. In order to avoid duplicating implementations between the operator and the ufunc, we inherit the `np.lib.mixins.NDArrayOperatorsMixin` class. This class implements all of the operators with methods that defer to the corresponding ufunc (SciPy 2019b).

NumPy allows any class to implement a magic method named `__array_ufunc__` to implement NumPy ufuncs. This method is called on argument types that NumPy does not recognize, so that they can properly handle the ufunc. For example, if NumPy sees a `VecValDer` in any of the parameters of a ufunc, it calls `VecValDer.__array_ufunc__` with parameters of a pointer to the original ufunc, the constant string `'__call__'`, and the parameters passed into the ufunc. This allows us to maintain a dictionary named `_HANDLED_UFUNCS` that maps from the NumPy ufunc to our implementation of that ufunc. We implement `__array_ufunc__` as reproduced in Figure 1.

An example entry in `_HANDLED_UFUNCS` would map `np.sin` to the function in Figure 2(a) that correctly computes both the value of sine and the corresponding derivative with the chain rule.

To implement the binary ufunc multiply, we need to consider the case of either one of or both parameters being a `VecValDer`. The code is presented in Figure 2(b).

```
1 def sin(x):
2     val = np.sin(x.val)
3     der = np.ndarray(x.der.shape)
4     for index, y in np.ndenumerate(np.cos(x.val)):
5         der[index] = y * dx[index]
6     return VecValDer(val, der)
```

(a) Our sine function.

```
1 def multiply(x1, x2):
2     if isinstance(x1,VecValDer) and isinstance(x2,VecValDer):
3         return VecValDer(x1.val*x2.val,
4                          x1.der*x2.val + x1.val*x2.der)
5     elif isinstance(x1, VecValDer):
6         return VecValDer(x1.val * x2, x1.der * x2)
7     elif isinstance(x2, VecValDer):
8         return VecValDer(x1 * x2.val, x1 * x2.der)
```

(b) Our multiply function.

Figure 2: Our implementations of sine and multiply.

Finally, in order to overload NumPy methods with no vector arguments, we replace the NumPy functions with new ones that output `VecValDer`s. For example, `np.ones` no longer points at NumPy's `ones` function, but to ours. For this to work, we do impose a limitation on the user's code; they must use `import numpy as np` to import NumPy instead of writing `from numpy import *`. The first form of the import statement imports a global instance of the NumPy module, which we modify when we introduce our own functions. The second form copies the original NumPy methods into the local namespace, preventing us from being able to replace them. As the first form of the import statement is common practice in Python scripting, we do not consider this restriction significant. As these functions are allocating new `VecValDer` derivative matrices, we need to know the shape of the vector we're differentiating with respect to. This leads to two problems: (1) how do we ensure we undo the changes to the global NumPy module (so as not to

```
def value_and_jacobian(z):
    return z.val, z.der.reshape((-1,z.der.shape[-2]))


class AutoDiff:
    def __init__(self, x):
        self.x = x

    def __enter__(self):
        new_np = {fn_name: getattr(self, fn_name)
                  for fn_name in _list_of_masked_functions}
        self.old_nps = _swap_numpy_methods(new_np)

        val = np.asarray(self.x)
        der = true_np.zeros((*val.shape, *val.shape))
        for i in np.ndindex(val.shape):
            der[(*i, *i)] = 1.0
        return VecValDer(val, der)

    def __exit__(self, type_, value, traceback):
        _swap_numpy_methods(self.old_nps)

    def zeros(self, shape):
        val = true_np.zeros(shape)
        der = true_np.zeros((*val.shape, *self.x.shape))
        return VecValDer(val, der)


def _swap_numpy_methods(new):
    output = {fn_name: getattr(np, fn_name)
              for fn_name in _list_of_masked_functions}
    for fn_name in _list_of_masked_functions:
        setattr(np, fn_name, new[fn_name])
    return output
```

Figure 3: Implementation of `AutoDiff` and `value_and_jacobian`.

break normal NumPy code), especially if we encounter an exception in the user's code, and (2) how do we best provide the shape of the vector we're differentiating with respect to the constructors?

The solution to both of these is a Python object known as a context manager and the `with` statement and corresponding block. A context manager `ctxmgr` is used as follows:

```
with ctxmgr as c:
    <the with block>
<code outside block>
```

A context manager has two magic methods, `__enter__` and `__exit__`. The `__enter__` method is called at the beginning of the `with` block. Its return value is assigned to the variable `c` in this example, though any identifier can be used there. Whenever control exits the `with` block, whether that be through an exception or by finishing executing, then the `__exit__` method is called to run cleanup code.

We define a context manager `AutoDiff` to capture when we are masking these constructors to instead return `VecValDers`. `AutoDiff`'s constructor takes one argument, `x`, the vector which we differentiate with respect to. `AutoDiff.__enter__` returns the `VecValDer` that has a `val` of `x`, and a `der` of the identity matrix, to represent that `x` is independent. `AutoDiff`'s `__exit__` method restores the original NumPy functions.

We reproduce partial code for `AutoDiff` in Figure 3. We only show the implementation of one masked method; the others are similar. `_list_of_masked_functions` is a list of strings naming each function we are masking. Additionally, `true_np` contains copies of all the original NumPy functions and is not modified when we change the NumPy methods. We have removed internal error handling code from the `_swap_numpy_methods` helper method for brevity.

3.3 How to Use `auto_diff`

To extract the Jacobian from the `der` field of a `VecValDer`, we just need to reshape the `der` array into a matrix. We included the function `value_and_jacobian` in Figure 3 to extract the Jacobian from a `der` array. We assume that we're differentiating a column vector, *i.e.*, a NumPy array with shape $m \times 1$, with respect to a column vector.

Putting everything together, we can compute and print the derivative of the function `f` with the code in Figure 4.

Finally, we give a brief overview of the codebase, available on GitHub. There are 5 files:

```
import numpy as np
import auto_diff as ad

def f(x):
    out = np.zeros((3, 1))
    out[0, 0] = x[1, 0]
    out[1, 0] = np.sin(out[0, 0])
    out[2, 0] = x[0, 0]
    return out

with ad.AutoDiff(np.array([[np.pi], [2]])) as x:
    y, J = ad.value_and_jacobian(f(x))

print("f(x) =", y)
print("Jacobian of f at x", J)
```

Figure 4: An example of using `auto_diff`.

- `vecvalder.py` implements the `VecValDer` class, implementing all the methods described here and the NumPy `ndarray` methods.
- `vecvalder_funcs_and_ufuncs.py` contains all of our ufunc implementations.
- `numpy_masking.py` implements the `AutoDiff` class and its helper methods.
- `true_np.py` copies all of the contents of `np`, so that `auto_diff` can access them even in `AutoDiff` contexts.
- `__init__.py` implements `value_and_jacobian` and copies `AutoDiff`, these two form the public API of `auto_diff`.

3.4 Comparison with Autograd

Google's JAX project provides Autograd, another automatic differentiation library for Python (Bradbury et al. 2018). Autograd's approach to solving the NumPy overloading problem, which we solve with `__array_ufunc__` and the `AutoDiff` context manager, is to reimplement NumPy. They require users of their code to `import jax.numpy as np` instead of using NumPy itself. Autograd's NumPy implementation is very different from the default in a number of key ways; for example, all arrays are immutable, and the `np.ndarray` constructor cannot be called directly. Further, `jax.numpy` is incompatible with real NumPy's objects, making calling into libraries and modules that were not designed to use JAX impossible without code modification. Code modification is often non-trivial, because all mutation needs to be eliminated. For example, the function `f` from the previous section, reproduced in Figure 5, cannot be differentiated by Autograd due to its dependence on mutating `out`.

```
def f(x):
    out = np.zeros((3, 1))
    out[0, 0] = x[1, 0]
    out[1, 0] = np.sin(out[0, 0])
    out[2, 0] = x[0, 0]
    return out
```

Figure 5: A function which is incompatible with Autograd.

Autograd's code is heavily optimized for Graphic Processing Units and Tensor Processing Units rather than for CPUs; accordingly, running it on a system without a GPU or a TPU leads to very poor performance. Experiments run in Sec. 4.1 suggest that Autograd, when run only on a CPU, is 5 times slower than `auto_diff`, even after applying the JAX optimizer to the Autograd function.

Additionally, Autograd is now delivered as part of JAX, which is a large library with lots of features and dependencies; an installation of JAX is 51 times larger then the entirety of the `auto_diff` codebase. Our library is significantly smaller and better suited for experimental projects.

Finally, because of Autograd's functional API and GPU-centric design, it is not possible to differentiate in an interactive environment without wrapping code into functions or to view derivatives in a debugger in the middle of a function. In contrast, `auto_diff` allows users to enter and exit `AutoDiff` contexts in an interactive python shell. This allows users to experiment interactively. We also allow users to access the derivative of any intermediate `VecValDer` they may encounter while using a debugger.

Autograd also provides an implementation of computing Jacobians using the reverse-mode differentiation algorithm. In reverse-mode differentiation, instead of directly computing the Jacobian at intermediate steps, the algorithm builds a graph that describes the dependence of the outputs on the inputs. A walk of this graph allows direct computation of the gradient of one output of the function. Reverse-mode is often faster for computing gradients of a single output (or a few outputs); while forward-mode is typically faster for functions with many outputs, such as vector-valued functions, which are important in many applications. Reverse-mode requires a significantly more complex implementation, but can be much more memory efficient. We plan to augment `auto_diff` with a reverse-mode differentiation option in the future.

4 VALIDATION AND APPLICATIONS

In this section, we illustrate the use of `auto_diff` in electronic device, circuit and mechanical system modelling and simulation.

4.1 Electronic Device Modeling and Circuit Simulation

Here, we describe an electronic device model which we will use to validate `auto_diff`. This device model captures the behaviour of a Bipolar Junction Transistor (BJT), using the Ebers-Moll model (Ebers and Moll 1954). In essence, these models provide differential equations that describe the behaviour of a BJT. These models are used by "equation engines" that compile device models together, given a description of how different circuit components are connected, to generate a differential-algebraic equation system that models the circuit as a whole (Wang et al. 2015).

The Ebers-Moll model is

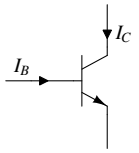
$$I_C = \frac{\mathrm{d}}{\mathrm{d}t}\left(C(V_{CE} - V_{BE})\right) - I_s(e^{(V_{BE}-V_{CE})/V_t} - 1) + \alpha_F I_s(e^{V_{BE}/V_t} - 1), \tag{1}$$

$$I_B = \frac{\mathrm{d}}{\mathrm{d}t}\left(CV_{BE}\right) + (1-\alpha_F)I_s(e^{V_{BE}/V_t} - 1) + (1-\alpha_R)I_s(e^{(V_{BE}-V_{CE})/V_t} - 1). \tag{2}$$

Useful values for the parameters are $I_s = 1\times10^{-12}\,\mathrm{A}$, $V_t = 2.6\times10^{-2}\,\mathrm{V}$, $\alpha_R = 0.5$, $\alpha_F = 0.99$, and $C = 1\times10^{-9}\,\mathrm{F}$. V_{CE} is defined as the voltage between the top node (where I_C is located) and the bottom node of the BJT. V_{BE} is the voltage between the left node and the bottom node of the BJT. I_B and I_C are the currents labeled on the diagram in Figure 6(a).

We rewrite this using two functions (Roychowdhury 2009): $\vec{q}(\vec{v})$, which takes in the two voltages and returns the portion of the right hand side of each equation which is being differentiated, and $\vec{f}(\vec{v})$, which captures the rest of each of the right hand sides. Using these, (1)–(2) can be rewritten as $\begin{bmatrix} I_C \\ I_B \end{bmatrix} = \frac{\mathrm{d}}{\mathrm{d}t}\vec{q}(\vec{v}) + \vec{f}(\vec{v})$. The code implementing $\vec{f}(\cdot)$ and $\vec{q}(\cdot)$ is shown in Figure 6. Numerical solvers require the derivatives of both $\vec{q}(\cdot)$ and $\vec{f}(\cdot)$; these derivatives are computed using `auto_diff` on the code in Figure 6.

In order to test `auto_diff`, we also hand-wrote code to compute the Jacobians of `EbersMoll_BJT.f` and `EbersMoll_BJT.q` and then compared the values of the Jacobian produced by our hand-written



(a) Schematic.

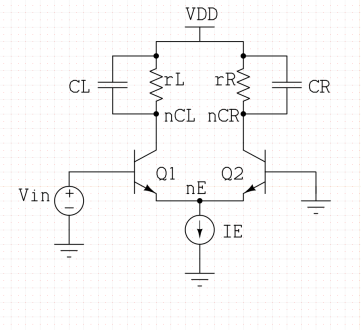
```
class EbersMoll_BJT:
    def __init__(self, Is=1e-12, Vt=0.026,
                 alphaF=0.99, alphaR = 0.5, C=1.0e-9):
        self._Is, self._Vt, self._C = Is, Vt, C
        self._alphaF, self._alphaR = alphaF, alphaR

    def f(self, vs):
        VBE, VCE = vs[0,0], vs[1,0]
        forward_diode_ID = self._Is * (np.exp(VBE/self._Vt) - 1)
        reverse_diode_ID = self._Is * (np.exp((VBE - VCE)/self._Vt) - 1)
        IC = forward_diode_ID - self._alphaF - reverse_diode_ID
        IB = forward_diode_ID * (1.0 - self._alphaF) + \
             reverse_diode_ID * (1.0 - self._alphaR)
        return np.array([[IC], [IB]])

    def q(self, vs):
        VBE, VCE = vs[0,0], vs[1,0]
        return np.array([[self._C * (VCE - VBE)], [self._C * VBE]])
```

(b) Code.

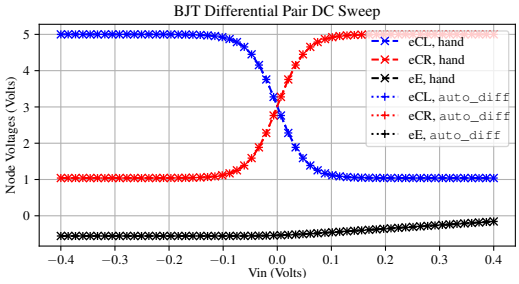
Figure 6: Ebers-Moll BJT code.

code and the Jacobian produced by `auto_diff`. We compared them on two hundred and fifty thousand different input values. The average absolute error for each element of the derivative of `f` was on the order of 10^{-20}, significantly smaller then the values in the Jacobian, which were on the order of 10^{-3}. Computing the derivatives with `auto_diff` takes 43.2 seconds. With our handwritten code, it took 3.11 seconds. With Autograd, running on a CPU instead of a GPU, it took 232 seconds.

For further validation, we inserted `auto_diff` into our implementation of Newton-Raphson, enabling it to compute Jacobians automatically from code for the function, rather than using user-provided code for derivatives. We then built, using the previous device model, a 3-dimensional ordinary differential equation of the BJT differential pair circuit shown in Figure 7(a).



(a) Schematic.



(b) DC sweep.

Figure 7: DC sweep on a BJT differential pair circuit.

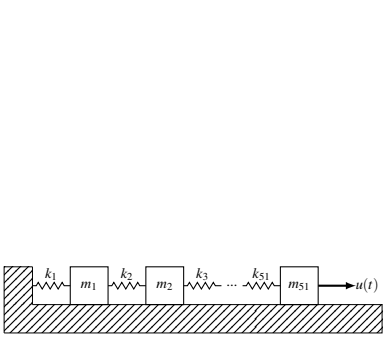
We then conducted a DC sweep analysis, *i.e.*, finding the DC steady state of the circuit for over a range of input voltages. As can be clearly seen from Figure 7(b), the DC steady states computed by `auto_diff` are identical to those from hand-generated code. The performance difference between the two was small: with hand-generated derivative code, it took 0.633 seconds to do the full DC sweep; with, `auto_diff`, it took 0.906 seconds. We did not attempt to use Autograd with our Newton-Raphson implementation due to the difficulty of porting all of our code to JAX's NumPy, with its immutable arrays and slow performance on CPU-bound tasks.

4.2 A 1-D Mechanical System Equation Engine

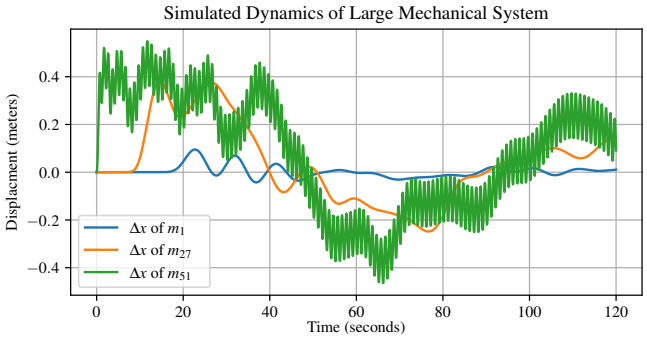
In order to validate `auto_diff` on a larger system, we developed a 1-dimensional Spring-Mass Equation Engine, which builds larger DAEs that model one dimensional systems of springs, masses, anchor points, and stiff rods. The schematic in Figure 8(a) shows our mechanical system. It is formed with an anchor on

the left, followed by 51 spring-mass pairs. The system is drawn horizontally with a floor as we include a friction term along the floor. Finally, we applied a sinusoidal force, labeled $u(t)$. Our DAE model for this system has 309 independent variables.

We used Backward Euler with Newton-Raphson to simulate 120 seconds of dynamics with a step size of 0.05 seconds. With hand-written derivative code, the simulation took 141 seconds; with `auto_diff` it took 461 seconds. We plot the displacement of the masses m_1, m_{27}, and m_{51} below.



(a) Diagram.



(b) Simulation results.

Figure 8: The large mechanical system.

5 CONCLUSION

We have introduced a new library, `auto_diff` that computes the Jacobians of almost arbitrary Python functions. We achieve this by using a Python context manager and NumPy's built-in support for overloading functions and operators to simplify the implementation. Unlike existing offerings, notably JAX's Autograd, we support mutable arrays and support using most libraries that were not designed to work with our library. We have validated the correctness of `auto_diff`, and evaluated its performance, via simulations of circuits and mechanical systems.

Future work on `auto_diff` will include adding support for computing Hessians and other higher-order derivatives. This will make `auto_diff` useful for optimization tasks. Additionally, we will add an implementation of reverse-mode differentiation. Further, we can optimize the codebase to improve performance. Switching from using NumPy arrays to SciPy's sparse arrays should reduce both the memory footprint and improve the speed of `auto_diff`. Other performance optimizations includes replacing the most commonly called functions with faster implementations in C or another compiled language. This should reduce the overhead of calling Python code and further improve performance.

ACKNOWLEDGMENTS

We thank Frank Liu of IBM Austin for bringing Martins, Sturdza, and Alonso (2003) to our attention. Support from the US National Science Foundation (via awards 1563812 and 1901004) is gratefully acknowledged.

REFERENCES

Andersson, J. A. E., J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl. 2019. "CasADi – A software framework for nonlinear optimization and optimal control". *Mathematical Programming Computation* vol. 11 (1), pp. 1–36.

Bradbury, J., R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne. 2018. "JAX: Composable Transformations of Python+NumPy Programs". Web link.

Corliss, G., and A. Griewank. 1993, January. "Operator Overloading as an Enabling Technology for Automatic Differentiation".

Ebers, J. J., and J. L. Moll. 1954, December. "Large-Signal Behavior of Junction Transistors". *Proceedings of the IRE* vol. 42 (12), pp. 1761–1772.

Feldmann, Melville, and Moinian. 1992, November. "Automatic differentiation in circuit simulation and device modeling". In *1992 IEEE/ACM International Conference on Computer-Aided Design*, pp. 248–253.

Forth, S. A. 2006, June. "An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB". *ACM Trans. Math. Softw.* vol. 32 (2), pp. 195–222.

Griewank, A., D. Juedes, and J. Utke. 1996, June. "Algorithm 755: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++". *ACM Trans. Math. Softw.* vol. 22 (2), pp. 131–167.

Guenter, B. 2007, August. "The D* Symbolic Differentiation Algorithm". ACM.

Abraham D. Lee 2013. "ad: a Python package for first- and second-order automatic differentation". Web link.

Martins, J., P. Sturdza, and J. Alonso. 2003, September. "The Complex-Step Derivative Approximation". *ACM Transactions on Mathematical Software* vol. 29, pp. 245–262.

Melville, R., S. Moinian, P. Feldmann, and L. Watson. 1993, May. "Sframe: An Efficient System for Detailed DC Simulation of Bipolar Analog Integrated Circuits Using Continuation Methods". *Analog Integr. Circuits Signal Process.* vol. 3 (3), pp. 163–180.

Neidinger, R. D. 2010. "Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming". *SIAM Review* vol. 52 (3), pp. 545–563.

Phipps, E. T., and D. M. Gay. 2006. "Automatic Differentation of C++ Codes With Sacado.". Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

Python Software Foundation 2019. "Data model: `__slots__`". Web link.

Roychowdhury, J. 2009, December. "Numerical Simulation and Modelling of Electronic and Biochemical Systems". *Foundations and Trends in Electronic Design Automation* vol. 3 (2-3), pp. 97–303.

SciPy 2019a. "NumPy Reference: Standard array subclasses, Special Attributes and Methods". Web link.

SciPy 2019b. "NumPy Reference: `numpy.lib.mixins.NDArrayOperatorsMixin`". Web link.

Wang, T., A. Karthik, B. Wu, J. Yao, and J. Roychowdhury. 2015, September. "MAPP: The Berkeley Model and Algorithm Prototyping Platform". In *Proc. IEEE CICC*, pp. 461–464.

AUTHOR BIOGRAPHIES

PARTH NOBEL is pursuing his Bachelors of Science in Electrical Engineering and Computer Science at UC Berkeley under Professor Jaijeet Roychowdhury. His email address is parthnobel@berkeley.edu.