Automatic Differentiation via Effects and Handlers

JESSE SIGAL, University of Edinburgh, United Kingdom

Machine learning, artificial intelligence, and scientific modelling have driven the demand for tools that enable derivative based optimization. Automatic differentiation (AD) is a family of algorithms used to calculate the derivatives of programs with only a constant factor slowdown. The utility of AD makes it worthwhile to implement it in as many languages as possible. Effects and handlers are a powerful programming language control flow construct based on delimited continuations. They are a structured method of including side effects into programs, and have found many uses including nondeterminism, state management, and concurrency. Mainstream programming languages are increasingly incorporating effects and handlers, notably OCaml 5.0.

We show that effects and handlers are a great match for implementing AD algorithms while maintaining asymptotic efficiency. In particular, effects and handlers allow for succinctness in the presence of the intrinsic complex control flow of AD. We implement four AD algorithms in OCaml 5.0 using effects and handlers. We provide benchmarks to empirically show that we can reach the correct asymptotic complexity for forward and reverse mode AD. Finally, we provide a real-world comparison by adding our implementation to a preexisting benchmark suite which includes systems such as TensorFlow and PyTorch, and show that our implementation is competitive with systems based on comparable algorithms.

1 INTRODUCTION

Machine learning, artificial intelligence, scientific modelling, information analysis, and other data heavy fields have driven the demand for tools which enable derivative based optimization. The family of algorithms known as automatic differentiation (AD) is the foundation of the tools which allow automated calculation of derivatives. The family can be coarsely divided into *forward mode* and *reverse mode*. Multiple modes exist because their asymptotics depend on different features of the differentiated programs. Forward mode AD was introduced in Wengert [1964], and reverse mode AD was created by Speelpenning [1980] in his thesis. It is not surprising that, given its long history, AD has been implemented in a variety of ways. The commonality between implementations is the preservation of the surprising efficiency of AD. Forward and reverse mode AD are only a constant multiple slower than the program being differentiated, with forward mode being 2 to 2.5 times slower than the original program, and reverse mode between 3 to 4 times slower. All in all, AD has experienced widespread adoption, either directly or through tools and systems based upon it.

Given the utility of AD, it is desirable to have implementations of it in as many languages as possible. However, the implementation strategy is heavily dependent on the language being used. Furthermore, the problem which AD is being applied to can necessitate the use of a particular mode of AD, and so the strategy employed must be flexible enough for many variations of AD. Thus, identifying a suitable set of features in a programming language that can cope with these varied demands is worthwhile.

Effects and handlers are a structured method of including side-effects into programs, and are themselves a structured form of delimited continuations. Algebraic effects were introduced in [G. Plotkin and Power 2001] and handlers for them were introduced in [G. Plotkin and Pretnar 2009]. Effects and handlers can be viewed as an extension of the common feature of catchable exceptions. Catching an exception terminates the program delimited by the exception handling code. In contrast, effect handlers can resume the handled code and pass a value to it. Effects and handlers can implement many common side effects such as state, exceptions, non-determinism,

---

Author's address: Jesse Sigal, University of Edinburgh, School of Informatics, Edinburgh, United Kingdom, jesse.sigal@ed.ac.uk.

2024. ACM 2475-1421/2024/3-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

logging, and input-output. They also support effect abstraction, composition, and program reuse through the ability of handlers to provide multiple interpretations of an effect. Furthermore, they provide a unified base in which to implement complex control flow constructs such as coroutines, generators, and async/await. In each instance, the control is non-local, an aspect in which effects and handlers excel. These use cases and others have motivated the inclusion of effects and handlers into mainstream projects such as OCaml [K. Sivaramakrishnan et al. 2021], culminating in their official inclusion in OCaml 5.0.

The ability of effects and handlers to capture non-local control flow and manage effects make them an ideal match for implementing AD. An effect can be defined where there is one operation for each primitive mathematical function, and a handler can be defined for each AD algorithm. The power of effect abstraction allows a program to be written once against a specified interface and later executed using any AD algorithm. Compositionality allows AD modes to be combined to create new modes. Furthermore, effects and handlers can provide the desired asymptotics for AD. Finally, they can also be competitive in raw performance with respect to comparable implementations in other languages using other methods.

Contributions. We make the following contributions in this paper:

- We implement four different AD modes in OCaml 5.0 using effects and handlers (section 3), and are the first to implement checkpointed reverse mode and tensor-valued operations with effects and handlers;
- We demonstrate how these modes are succinctly expressed using effects and handlers and that they are composable (section 3);
- We provide experimental evidence that our implementations, including forward and reverse mode, have the correct asymptotics (section 4.1); and
- We provide experimental evidence that our reverse mode implementation extended with tensor-valued operations is competitive with comparable implementations (section 4.2).

In summary, we show how to implement AD with effects and handlers in a modular, composable, simple, and performant way.

2 BACKGROUND ON AUTOMATIC DIFFERENTIATION

2.1 Deriving Forward and Reverse Modes

Forward and reverse mode AD can be easily derived for pure, straight-line programs. We will do so by example. We assume that the reader is familiar with partial derivatives of real-valued functions, as well as matrix-matrix and matrix-vector multiplication. Consider the algebraic definition

$$z = h(g(f(a), b), f(a))$$

where $a, b \in \mathbb{R}$, $f\colon \mathbb{R} \to \mathbb{R}$, $g, h\colon \mathbb{R}^2 \to \mathbb{R}$, and all functions are differentiable. We can rewrite this as a sequence of calculations using intermediate variables

$$\begin{aligned} x &= f(a) && (1) \\ y &= g(x, b) && (2) \\ z &= h(y, x) && (3) \end{aligned}$$

and consider the sequence as a pure, straight-line program where the variables a, b are inputs and the variables x, y, z are initialized to 0. We now regard the state of the program at each line as a five-tuple $(a, b, x, y, z) \in \mathbb{R}^5$ containing the values of our variables. Thus, each line (i) gives a function $F_i \colon \mathbb{R}^5 \longrightarrow \mathbb{R}^5$, i.e.

$$\begin{aligned}
F_1(v_0, v_1, v_2, v_3, v_4) &= (v_0, v_1, f(v_0), v_3, v_4) \\
F_2(v_0, v_1, v_2, v_3, v_4) &= (v_0, v_1, v_2, g(v_2, v_1), v_4) \\
F_3(v_0, v_1, v_2, v_3, v_4) &= (v_0, v_1, v_2, v_3, h(v_3, v_2)).
\end{aligned}$$

Our program can then be rewritten to

$$\begin{aligned}
\vec{x}_0 &= (a, b, 0, 0, 0) \\
\vec{x}_1 &= F_1(\vec{x}_0) \\
\vec{x}_2 &= F_2(\vec{x}_1) \\
\vec{x}_3 &= F_3(\vec{x}_2)
\end{aligned}$$

where $\vec{x}_3$ gives the final state. The multivariate version of differentiation is given by the Jacobian, which for a differentiable function $F \colon \mathbb{R}^n \longrightarrow \mathbb{R}^m$ and a point $\vec{x} \in \mathbb{R}^n$ we denote by $\nabla F(\vec{x})$. The Jacobian $\nabla F(\vec{x})$ is an $m \times n$ matrix containing all the partial derivatives of F at $\vec{x}$. Thus, writing $F(\vec{x})$ as $F(\vec{x}) = (f_1(\vec{x}), \ldots, f_m(\vec{x}))$ for differentiable functions $f_j \colon \mathbb{R}^n \longrightarrow \mathbb{R}$, the Jacobian $\nabla F(\vec{x})$ is

$$\nabla F(\vec{x}) \coloneqq \begin{pmatrix} \partial_1 f_1(\vec{x}) & \cdots & \partial_n f_1(\vec{x}) \\ \vdots & \ddots & \vdots \\ \partial_1 f_m(\vec{x}) & \cdots & \partial_n f_m(\vec{x}) \end{pmatrix}$$

where ∂_i is the i^{th} partial derivative operator. The Jacobian satisfies the multivariate chain rule $\nabla(G \circ F)(\vec{x}) = \nabla G(F(\vec{x})) \times \nabla F(\vec{x})$. Therefore, by viewing our program as a composition of state-transforming functions, namely $F_3 \circ F_2 \circ F_1$, we calculate

$$\nabla(F_3 \circ F_2 \circ F_1)(\vec{x}_0) = \nabla F_3(\vec{x}_2) \times \nabla F_2(\vec{x}_1) \times \nabla F_1(\vec{x}_0)$$

where × is matrix-matrix multiplication, and later matrix-vector multiplication as well. The crux of both forward and reverse mode AD is this calculation, which they each use differently.

For forward mode, we observe that the matrix product can be computed from right-to-left by

$$\begin{aligned}
X_1 &= \nabla F_1(\vec{x}_0) \\
X_2 &= \nabla F_2(\vec{x}_1) \times X_1 \\
X_3 &= \nabla F_3(\vec{x}_2) \times X_2.
\end{aligned}$$

It would be inefficient to materialize entire matrices in practice, and so we can pre-multiply by a vector $\vec{dx}_0$ to obtain

$$\nabla(F_3 \circ F_2 \circ F_1)(\vec{x}_0) \times \vec{dx}_0 = \nabla F_3(\vec{x}_2) \times \nabla F_2(\vec{x}_1) \times \nabla F_1(\vec{x}_0) \times \vec{dx}_0$$

giving the sequence of vectors

$$\begin{aligned}
\vec{dx}_1 &= \nabla F_1(\vec{x}_0) \times \vec{dx}_0 \\
\vec{dx}_2 &= \nabla F_2(\vec{x}_1) \times \vec{dx}_1 \\
\vec{dx}_3 &= \nabla F_3(\vec{x}_2) \times \vec{dx}_2.
\end{aligned}$$

Calculating the Jacobian of the function $F_1(v_0, v_1, v_2, v_3, v_4) = (v_0, v_1, f(v_0), v_3, v_4)$ at $\vec{x}_0$, we see

$$\nabla F_1(\vec{x}_0) = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ \partial f(a) & & 0 & & \\ & & & 1 & \\ & & & & 1 \end{pmatrix}$$

where ∂f is shorthand for the derivative of $f\colon \mathbb{R} \longrightarrow \mathbb{R}$ at a and empty entries are 0. Similarly,

$$\nabla F_2(\vec{x}_1) = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & \partial_R g(x,b) & \partial_L g(x,b) & 0 & \\ & & & & 1 \end{pmatrix} \qquad \nabla F_3(\vec{x}_2) = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & \partial_R h(y,x) & \partial_L h(y,x) & 0 \end{pmatrix}$$

where $\partial_R g$ is the partial derivative of g in the right argument and so on. Observe that the Jacobians are sparse due to each of the F_i's only changing one variable. We now calculate the vectors $\vec{dx}_i$. We use the notation $\vec{dx}_i[a]$, $\vec{dx}_i[b]$, $\vec{dx}_i[x]$, $\vec{dx}_i[y]$, and $\vec{dx}_i[z]$ for the 1st, 2nd, 3rd, 4th, and 5th components respectively. Pairing each vector with the matching line of our original program, we get

$$\begin{aligned} x &= f(a) & \vec{dx}_1 &= (\vec{dx}_0[a], \vec{dx}_0[b], \partial f(a) \cdot \vec{dx}_0[a], \vec{dx}_0[y], \vec{dx}_0[z]) \\ y &= g(x,b) & \vec{dx}_2 &= (\vec{dx}_1[a], \vec{dx}_1[b], \vec{dx}_1[x], \partial_R g(x,b) \cdot \vec{dx}_1[b] + \partial_L g(x,b) \cdot \vec{dx}_1[x], \vec{dx}_1[z]) \\ z &= h(y,x) & \vec{dx}_3 &= (\vec{dx}_2[a], \vec{dx}_2[b], \vec{dx}_2[x], \vec{dx}_2[y], \partial_R h(y,x) \cdot \vec{dx}_2[x] + \partial_L h(y,x) \cdot \vec{dx}_2[y]). \end{aligned}$$

Observe that $\vec{dx}_3[x] = \vec{dx}_2[x] = \vec{dx}_1[x]$ because the x components of the $\vec{dx}_i$'s are only changed when x is assigned to. Thus, we do not need to define a vector $\vec{dx}_i$ at each step, it is sufficient to only define one new scalar variable. We can therefore rewrite the above as

$$\begin{aligned} x &= f(a) & dx &= \partial f(a) \cdot da \\ y &= g(x,b) & dy &= \partial_R g(x,b) \cdot db + \partial_L g(x,b) \cdot dx \\ z &= h(y,x) & dz &= \partial_R h(y,x) \cdot dx + \partial_L h(y,x) \cdot dy \end{aligned}$$

which exactly captures the forward mode algorithm. Namely, each line is paired with a derivative calculation using the partial derivatives, i.e. $y = f(x_1, x_2, \ldots, x_n)$ is paired with

$$dy = \sum_{i=1}^{n} \partial_i f(x_1, x_2, \ldots, x_n) \cdot dx_i.$$

for a fresh variable dy. Forward mode AD can also be viewed as arithmetic in the ring of truncated Taylor series [Griewank and A. Walther 2008, Ch. 13].

For reverse mode, we observe that the matrix product can be transformed by transposition

$$\nabla(F_3 \circ F_2 \circ F_1)(\vec{x}_0)^\intercal = \nabla F_1(\vec{x}_0)^\intercal \times \nabla F_2(\vec{x}_1)^\intercal \times \nabla F_3(\vec{x}_2)^\intercal$$

and that this *reverses* the order of matrix multiplication. We can again calculate right-to-left,

$$\begin{aligned} X_3 &= \nabla F_3(\vec{x}_2)^\intercal \\ X_2 &= \nabla F_2(\vec{x}_1)^\intercal \times X_3 \\ X_1 &= \nabla F_1(\vec{x}_0)^\intercal \times X_2 \end{aligned}$$

and similarly opt for pre-multiplying by a vector $\vec{\delta x}_4$

$$\nabla(F_3 \circ F_2 \circ F_1)(\vec{x}_0)^\intercal \times \vec{\delta x}_4 = \nabla F_1(\vec{x}_0)^\intercal \times \nabla F_2(\vec{x}_1)^\intercal \times \nabla F_3(\vec{x}_2)^\intercal \times \vec{\delta x}_4$$

and thus we can define a sequence of intermediate vectors

$$\begin{aligned} \vec{\delta x}_3 &= \nabla F_3(\vec{x}_2)^\intercal \times \vec{\delta x}_4 \\ \vec{\delta x}_2 &= \nabla F_2(\vec{x}_1)^\intercal \times \vec{\delta x}_3 \\ \vec{\delta x}_1 &= \nabla F_1(\vec{x}_0)^\intercal \times \vec{\delta x}_2. \end{aligned}$$

The transposes of the Jacobians

$$\nabla F_1(\vec{x}_0)^\intercal = \begin{pmatrix} 1 & & \partial f(a) & & \\ & 1 & & & \\ & & 0 & & \\ & & & 1 & \\ & & & & 1 \end{pmatrix} \quad \nabla F_2(\vec{x}_1)^\intercal = \begin{pmatrix} 1 & & & & \\ & 1 & & \partial_R g(x,b) & \\ & & 1 & \partial_L g(x,b) & \\ & & & 0 & \\ & & & & 1 \end{pmatrix} \quad \nabla F_3(\vec{x}_2)^\intercal = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \partial_R h(y,x) \\ & & & 1 & \partial_L h(y,x) \\ & & & & 0 \end{pmatrix}$$

are also sparse. Let $\vec{\delta x}_i[a]$, $\vec{\delta x}_i[b]$, $\vec{\delta x}_i[x]$, $\vec{\delta x}_i[y]$, and $\vec{\delta x}_i[z]$ for the first, second, third, fourth, and fifth components of $\vec{\delta x}_i$ respectively. Calculating with components, we see

$$\begin{aligned}
\vec{\delta x}_3 &= (\vec{\delta x}_4[a], \vec{\delta x}_4[b], \vec{\delta x}_4[x] + \partial_R h(y,x) \cdot \vec{\delta x}_4[z], \vec{\delta x}_4[y] + \partial_L h(y,x) \cdot \vec{\delta x}_4[z], 0) \\
\vec{\delta x}_2 &= (\vec{\delta x}_3[a], \vec{\delta x}_3[b] + \partial_R g(x,b) \cdot \vec{\delta x}_3[y], \vec{\delta x}_3[x] + \partial_L g(x,b) \cdot \vec{\delta x}_3[y], 0, \vec{\delta x}_3[z]) \\
\vec{\delta x}_1 &= (\vec{\delta x}_2[a] + \partial f(a) \cdot \vec{\delta x}_2[x], \vec{\delta x}_2[b], 0, \vec{\delta x}_2[y], \vec{\delta x}_2[z])
\end{aligned}$$

and note that each line accumulates derivatives into the arguments of the function used based on the resulting variable. For example, $x = f(a)$ adds $f(a) \cdot \vec{\delta x}_2[x]$ to $\vec{\delta x}_2[a]$. We can use mutable variables δa, δb, δx, and δy initialized to 0 to perform the above calculation

$$\begin{aligned}
x &= f(a) \\
y &= g(x,b) \\
z &= h(y,x) \\
\delta y \mathrel{+}&= \partial_L h(y,x) \cdot \delta z, \quad \delta x \mathrel{+}= \partial_R h(y,x) \cdot \delta z \\
\delta x \mathrel{+}&= \partial_L g(x,b) \cdot \delta y, \quad \delta b \mathrel{+}= \partial_R g(x,b) \cdot \delta y \\
\delta a \mathrel{+}&= \partial f(a) \cdot \delta x
\end{aligned}$$

which is exactly reverse mode AD, modulo zeroing out mutable variables. Namely, each line has a corresponding stateful derivative update which accumulates into the mutable derivative associated with its arguments, i.e. $y = f(x_1, x_2, \dots, x_n)$ is paired with

$$\delta x_1 \mathrel{+}= \partial_i f(x_1, \dots, x_n) \cdot \delta y, \ \dots, \ \delta x_n \mathrel{+}= \partial_n f(x_1, \dots, x_n) \cdot \delta y$$

in the reverse order of the original program.

2.2 Automatic Differentiation in Practice

Automatic differentiation can be broadly categorized by mode (i.e. the specific algorithm) and implementation strategy. Some popular systems use a domain-specific language (DSL) strategy where the user specifies a computation graph which is then the main object from which derivatives are calculated. The computation graph and resulting derivative graph are often optimized after construction. The DSL can either be fine-grained (operator level), or coarse-grained (computational module or model level). The operator level encompasses basic scalar operations such as addition and multiplication and tensor operations such as summing along a dimension and taking slices. On the other hand, the module level includes examples such as fully-connected neural networks and convolutional layers. Examples of fine-grained systems are Theano [Theano Development Team 2016], CNTK [Seide and Amit Agarwal 2016], and TensorFlow [Abadi, Ashish Agarwal, et al. 2015] and examples of coarse-grained systems are Torch7 [Collobert and Kavukcuoglu 2011] and Caffe [Jia et al. 2014]. The computation graph approach, while useful, is usually limited to a subset of the host languages expressiveness. Thus, computation graph DSLs are generally considered to be *algorithmic* differentiation but not automatic differentiation, although this distinction is somewhat artificial.

Forward and reverse mode are the main categories of AD. There are also variations of these main modes; we list some examples.

- Sparse versions of forward and reverse mode take advantage the of structure of the program and requested results to perform less computation [Griewank and A. Walther 2008, Ch. 7].
- Reverse mode has a memory footprint which is linear in the length of the calculation, and so there exists a checkpointed form which re-runs portions of the original program in exchange for a lower memory footprint [Griewank and A. Walther 2008, Ch. 12][Hascoët and Araya-Polo 2006].
- Forward and reverse mode are in fact extreme choices on a spectrum. A given first-order program can be viewed as directed acyclic graph with mathematical operations as nodes and data dependencies as edges. AD can then be defined in terms of edge and face eliminations on this graph, with forward and reverse mode being extremal choices in the order of elimination. [Griewank and A. Walther 2008, Ch. 9]
- Forward mode can be derived from truncating Taylor series at their linear terms. Thus, truncating at higher-order terms allows for method similar to forward mode which calculates higher-order derivatives. [Griewank and A. Walther 2008, Ch. 13] [Barak A Pearlmutter and Jeffrey Mark Siskind n.d.]
- Forward and reverse mode can also be layered on top of each other in order to compute higher-order derivatives [Barak A Pearlmutter and Jeffrey M Siskind n.d.][Betancourt 2018].

Beyond the mode used, there are various non-DSL implementation strategies for AD. A useful categorization is into elemental, compiler-based, source transformations, and operator overloading [Baydin et al. 2018]. Elemental methods consist of programming with substitute mathematical functions defined by an AD library. Elemental AD is the simplest method to provide when the language does not support operator overloading. Examples include WCOMP and UCOMP [Lawson 1971]. Compiler-based AD uses special purpose compilers to generate derivative code during compilation. Examples include Stalingrad [Barak A. Pearlmutter and Jeffrey Mark Siskind 2008], Tangent [Merriënboer et al. 2017], SLANG [Thames 1969], and PROSE [Pfeiffer 1987]. Source transformation methods take program text and generates new program text containing the old code which also computes derivatives. Examples include ADIFOR [C. Bischof et al. 1996], ADIC [C. H. Bischof et al. 1997], and Tapenade [Hascoët and Pascual 2013; Pascual and Hascoët 2008]. Finally, operator overloading simply overloads the chosen mathematical functions to effectively perform the elemental method more ergonomically. Examples include ADOL-C [Andrea Walther 2009], the *ad* package for Python[1], the *ad* package for Haskell[2], and the DiffSharp package for F# and C# [Baydin et al. 2018].

The last distinction we make cuts across our other categorizations. Some AD systems are *define-then-run*, or static, whereby a the program written is statically analyzed and transformed into a new program. Static approaches include DSL and source transformation techniques, and are often the fastest methods due to optimization opportunities. Other AD systems are *define-by-run*, or dynamic, where the derivative is calculated as the defined program runs. Dynamic approaches are usually slower but more flexible and interactive, and includes methods such as elemental and operator overloading techniques.

---

[1] https://pypi.org/project/ad/

[2] https://hackage.haskell.org/package/ad

3 AUTOMATIC DIFFERENTIATION VIA EFFECTS AND HANDLERS BY EXAMPLE

3.1 Framework

We will assume the reader has a basic familiarity with algebraic effects and handlers. For a tutorial on algebraic effects we suggest [Bauer 2019] and for effect handlers we suggest [Pretnar 2015]. A function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ is called *smooth* when it has all partial derivative of all orders, meaning that the partial derivatives and Jacobian of such an f are also smooth. Thus, smooth functions are closed under differentiation, and are the natural class to consider when creating compositional AD algorithms. Of course, we cannot express all smooth functions, and so we choose a subset of smooth functions closed under differentiation as a family. The minimal collection of smooth functions is addition, multiplication, and constant functions taking values in the non-negative integers. For example, because we include the function $\sin(x)$, we also include the function $\cos(x)$ because ${}^{\partial}\!/_{\partial x}(\sin(x)) = \cos(x)$. Alternatively, because $\cos(x) = \sin(x + {}^{\pi}\!/_{2})$, we could leave $\cos(x)$ out, but in practice redundant functions are included for clarity and numerical considerations. A further practical choice is the inclusion of division, which is undefined when dividing by 0, but is smooth everywhere else.

We begin by opening the `Effect` module from the standard library to access effects and handlers. We then define data types enumerating what family of functions we wish to use, split by the number of arguments each function takes. Furthermore, we define a helper data type `arg` for specifying argument position of binary functions. Next, we define our effect and helper functions, collected into a module type named `SMOOTH`. The helper functions include overloaded operators, convenience functions for calling the smooth effect, and implementations of derivatives.

```
open Effect (* Access effect and handler interface *)

type nullary = Const of float                  (* Nullary functions *)
type unary = Negate | Sin | Cos | Exp          (* Unary functions   *)
type binary = Plus | Subtract | Times | Divide (* Binary functions  *)

type arg = L | R (* Left or right argument of a binary function *)

module type SMOOTH = sig (* Module type for smooth function effect *)
  type t (* Number type *)
  type _ Effect.t += Ap0 : nullary -> t Effect.t        (* Apply nullary *)
                   | Ap1 : unary * t -> t Effect.t      (* Apply unary   *)
                   | Ap2 : binary * t * t -> t Effect.t (* Apply binary  *)

  val c : float -> t              (* --- Begin helper functions --- *)
  val ( ~. ) : t -> t
  val sin_ : t -> t
  val cos_ : t -> t
  val exp_ : t -> t
  val ( +. ) : t -> t -> t
  val ( -. ) : t -> t -> t
  val ( *. ) : t -> t -> t
  val ( /. ) : t -> t -> t

  val ap0 : nullary -> t
  val ap1 : unary -> t -> t
  val ap2 : binary -> t -> t -> t (* ---- End helper functions ---- *)

```

```
29   val der1 : unary -> t -> t               (* Derivative of unary functions  *)
30   val der2 : binary -> arg -> t -> t -> t (* Derivative of binary functions *)
31 end
```

The `SMOOTH` module type includes a type `t` on line 10, which will allow for the composition of different modes of AD via different instantiations. For example, in forward mode it will be instantiated to a pair consisting of a value and a derivative, and in reverse mode a pair of a value and a reference to a mutable derivative. We define our effect on lines 11 to 13 by augmenting the `Effect.t` extensible variant with `Ap0`, `Ap1`, and `Ap2`. Each constructor corresponds to applying an n-ary operation. Lines 15 to 27 declare helper functions for using these effects. Finally, lines 29 and 30 declare the derivatives of unary and binary functions.

We can now define the implementation. Each helper function uses the `perform` function from the `Effect` module to perform the given effect.

```
33 module Smooth (T : sig type t end) : SMOOTH with type t = T.t = struct
34   type t = T.t (* Use the passed in type *)
35   type _ Effect.t += Ap0 : nullary -> t Effect.t
36                    | Ap1 : unary * t -> t Effect.t
37                    | Ap2 : binary * t * t -> t Effect.t
38
39   let c x = perform (Ap0 (Const x))
40   let ( ~. ) a = perform (Ap1 (Negate, a))
41   let sin_ a = perform (Ap1 (Sin, a))
42   let cos_ a = perform (Ap1 (Cos, a))
43   let exp_ a = perform (Ap1 (Exp, a))
44   let ( +. ) a b = perform (Ap2 (Plus, a, b))
45   let ( -. ) a b = perform (Ap2 (Subtract, a, b))
46   let ( *. ) a b = perform (Ap2 (Times, a, b))
47   let ( /. ) a b = perform (Ap2 (Divide, a, b))
48
49   let ap0 n =  perform (Ap0 n)
50   let ap1 u x = perform (Ap1 (u, x))
51   let ap2 b x y = perform (Ap2 (b, x, y))
52
53   let der1 u x = match u with (* ∂/∂x (u(x)) *)
54     | Negate -> ~. (c 1.0)     (* ∂/∂x(−x) = −1 *)
55     | Sin    -> cos_ x         (* ∂/∂x(sin(x)) = cos(x) *)
56     | Cos    -> ~. (sin_ x)    (* ∂/∂x(cos(x)) = − sin(x) *)
57     | Exp    -> exp_ x         (* ∂/∂x(e^x) = e^x *)
58   let der2 b arg x y = match b with (* ∂/∂x_arg (b(x_L, x_R)), for x_L = x, x_R = y *)
59     (* ∂/∂x(x + y) = 1,   ∂/∂y(x + y) = 1 *)
60     | Plus -> (match arg with L -> c 1.0 | R -> c 1.0)
61     (* ∂/∂x(x − y) = 1,   ∂/∂y(x − y) = −1 *)
62     | Subtract -> (match arg with L -> c 1.0 | R -> c (-1.0))
63     (* ∂/∂x(x · y) = y,   ∂/∂y(x · y) = x *)
64     | Times -> (match arg with L -> y | R -> x)
65     (* ∂/∂x(x/y) = 1/y,   ∂/∂y(x/y) = −x/y² *)
66     | Divide -> (match arg with L -> (c 1.0) /. y | R -> (~. x) /. (y *. y))
67 end
```

The helper function `c` on line 39 embeds any float into our numeric type. As a constant function, its derivative is 0, and so we have no `der0` function.

3.2 Evaluation Mode

Our first handler will interpret our smooth functions using floating point numbers. The `Evaluate` module is a super type of the `SMOOTH` module type by virtue of including the `Smooth` module with the number type `Smooth.t` instantiated to `float`. Later modules will be functors which accept `SMOOTH` type modules, allowing compositionality with `Evaluate` as the base case.

```
1  open Effect.Deep (* Access effect and (deep) handler interface  *)
2  open Float       (* Floating point operations                   *)
3  open Smooth      (* Smooth function effect and helper functions *)
4
5  module Evaluate = struct
6    (* Include smooth function effect with number type equal to `float` *)
7    include Smooth (struct type t = float end)
8
9    (* Handle a smooth function with the corresponding `float` operation *)
10   let (evaluate : ('a, 'a) handler) = {
11     retc = (fun x -> x); (* Do nothing with returned value  *)
12     exnc = raise;        (* Re-raise encountered exceptions *)
13     effc = (fun (type x) (eff : x Effect.t) ->
14       match eff with (* Match the intercepted effect *)
15       | Ap0 n -> Some (fun (k : (x, 'a) continuation) ->
16           match n with
17           | Const x -> continue (k : (float, 'a) continuation) x
18         )
19       | Ap1 (u, x) -> Some (fun k ->
20           match u with
21           | Negate -> continue k (neg x)
22           | Sin    -> continue k (sin x)
23           | Cos    -> continue k (cos x)
24           | Exp    -> continue k (exp x)
25         )
26       | Ap2 (b, x, y) -> Some (fun k ->
27           match b with
28           | Plus     -> continue k (add x y)
29           | Subtract -> continue k (sub x y)
30           | Times    -> continue k (mul x y)
31           | Divide   -> continue k (div x y)
32         )
33       | _ -> None (* Do not handle the effect if not a smooth effect *)
34     )
35   }
36 end
```

Line 7 includes the `Smooth` module, meaning that our effects and helper functions, with `t` set to `float`, are part of `Evaluate`. Line 10 defines the `evaluate` handler. The type `('a, 'a) handler` describes a handler which can handle a computation which returns an arbitrary type `'a` and returns a value of type `'a`. A handler in OCaml consists of three parts: the return clause `retc`, the exception clause `exnc`, and the effect clause `effc`. The return clause defines a transformation on the return value of the handled computation. Next, exception clause prescribes what occurs when an exception is encountered. Finally, the effect clause lets us match on on the effect being handled. In `evaluate`, the return and exception clauses are trivial.

Let us examine the effect clause of `evaluate`. It consists of a function with argument `eff`, the effect being handled, and possibly returns a function consuming a continuation. A value of `None` indicates the current handler does not wish to intercept the effect, while a value of `Some (fun k -> ...)` returns a function to consume the intercepted effect through use of the continuation `k`. We intercept each of `Ap0`, `Ap1`, and `Ap2`, consuming the continuation `k` with the `continue` function from `Effect.Deep` and passing to `k` the result applying the corresponding `float` function. By specifying a deep handler, all subsequent uses of the matched effects will also be handled by `evaluate`.

The `Evaluate` module can be used to create an effectful computation which can be handled by `evaluate`. Consider the following snippet.

```
let _ =
  let open Evaluate in
  let sqr x = x *. x in (* Square argument using effectful operation *)
  let res = (match_with : ('c -> 'a) -> 'c -> ('a, 'b) handler -> 'b)
    (* Effectful computation to handle *)
    (fun (twice, x) -> if twice then sqr (sqr x) else sqr x)
    (true, 5.0) (* Argument to computation *)
    evaluate    (* Handler for computation *)
  in
  Printf.printf "%f\n" res (* Prints "625.000000"=5^4 *)
```

We begin by opening the `Evaluate` module and defining an effectful function `sqr` which squares its argument. In order to compute with `sqr`, we must run it in the context of `evaluate`, which is achieved with the `match_with` function from `Effect.Deep`. The first argument is the computation to be handled, in which we choose to use `sqr` once or twice, the second argument gives the input to the computation, and the third specifies which handler to use. The result `res` is `625.0` (= 5^4) as expected. Thus, the `evaluate` handler has dynamically interpreted the `*.` operation as multiplication on floats. We have now seen how to add a new effect, create a handler for effects, and run a computation using a specified handler in OCaml 5.0.

3.3 Forward Mode

Our next example is forward mode AD. The forward mode implementation will take the form of an OCaml functor, i.e. a module-level function, taking as input a `SMOOTH` module and producing a module which is a `SMOOTH` super-type. As illustrated in section 2.1 in the derivation of forward mode, each smooth function will now operate on a pair of values, the original value and its derivative. Thus, we define a data type of paired numbers, and make it parameterized to allow nesting of AD. The implementation of the forward mode handler is then a straightforward transcription of the algorithm.

```
open Effect.Deep (* Access effect and (deep) handler interface  *)
open Smooth      (* Smooth function effect and helper functions *)

type 't paired = {v : 't; dv : 't} (* A value paired with its derivative *)

(* Perform forward mode w.r.t. an interpretation of reals given by T *)
module Forward (T : SMOOTH) = struct
  (* Include helper functions and effects instantiated with paired numbers *)
  include Smooth (struct type t = T.t paired end)

  (* Handler for forward mode *)
  let (forward : ('a, 'a) handler) = {
    retc = (fun x -> x); (* Do nothing with returned value  *)
```

```
14      exnc = raise;          (* Re-raise encountered exceptions *)
15      effc = (fun (type a) (eff : a Effect.t) ->
16        match eff with
17        | Ap0 n -> Some (fun (k : (a, _) continuation) -> let open T in
18            (* v = n,   dv = 0 *)
19            continue k {v = ap0 n; dv = c 0.0}
20          )
21        | Ap1 (u, x) -> Some (fun k -> let open T in
22            (* v = u(x),   dv = ∂u(x) · dx *)
23            continue k {v = ap1 u x.v; dv = der1 u x.v *. x.dv}
24          )
25        | Ap2 (b, x, y) -> Some (fun k -> let open T in
26            (* v = b(x, y),   dv = ∂_L b(x, y) · dx + ∂_R b(x, y) · dy *)
27            continue k {v = ap2 b x.v y.v; dv = (der2 b L x.v y.v *. x.dv) +.
28                                                 (der2 b R x.v y.v *. y.dv)}
29          )
30        | _ -> None
31      )
32    }
33
34    (* diff f x = ∂f(z)/∂z (x) *)
35    let diff (f : T.t paired -> T.t paired) (x : T.t) =
36      let res = match_with f {v = x; dv = T.c 1.0} forward in res.dv
37  end
```

Line 4 defines the paired data type. Line 7 defines the `Forward` module, which now takes a `SMOOTH` module `T`. As before, we include the `Smooth` module (line 9), this time instantiating the number type with paired numbers based on `T`'s number type to allow nesting. We define the `forward` handler from line 12. Each case in the effect clause implements the forward mode rule; note how we **`open`** `T` each time so that the calculations are with respect to `T`. Finally, we define a helper function `diff` starting on line 35 which uses `forward` to calculate the derivative of a function `f` which operates on paired numbers. Note that `f` must be defined only using the combinators provided by `Smooth`, e.g. `sin_` and `+.`, and not by destructuring the `paired` data type or else an invalid derivative may be calculated.

The following is an example of how to use `Forward` by composing the `forward` handler inside `diff` with our previously defined `evaluate` handler.

```
1   let _ =
2     let module E = Evaluate in
3     let module F = Forward(E) in (* Instantiate forward mode with floats   *)
4     let sqr x = F.(x *. x) in    (* Square argument using operation from F *)
5     let res = match_with
6       (fun (twice, y) -> F.diff (fun x -> if twice then sqr (sqr x) else sqr x) y)
7       (true, 5.0)
8       E.evaluate
9     in
10    Printf.printf "%f\n" res (* Prints "500.000000"= 4 · 5^3 = ∂(x^4)/∂x (5)) *)
```

Line 3 instantiates `Forward` with `Evaluate`, allowing the `diff` function to be handled with `evaluate` and produce a `float` result. Note that the `sqr` function is defined using operations from `F`, allowing it to be used as an argument to `F.diff`. Our next example shows that we can also instantiate `Forward` with itself to calculate second derivatives.

```
let _ =
  let module E = Evaluate in
  let module F = Forward(E) in  (* Instantiate forward mode with floats       *)
  let module FF = Forward(F) in (* Instantiate forward mode with forward mode *)
  let sqr x = FF.(x *. x) in    (* Square argument using operation from FF    *)
  let res = match_with (fun (twice, z) ->
      F.diff (fun y ->
        FF.diff (fun x -> if twice then sqr (sqr x) else sqr x) y
      ) z
    ) (true, 5.0) E.evaluate
  in
  Printf.printf "%f\n" res (* Prints "300.000000"= $12 \cdot 5^2 = \frac{\partial^2(x^4)}{\partial x^2}(5)$ *)
```

Note that when we define `sqr` here, we must use operations from `FF`. To avoid redefining `sqr` every time, we can create a functor

```
module Sqr (T : SMOOTH) = struct
  let sqr x = T.(x *. x)
end
```

and instantiate it as needed.

3.4 Reverse Mode

Recall our example from section 2.1:

$$\begin{aligned} x &= f(a) && (1)\\ y &= g(x,b) && (2)\\ z &= h(y,x) && (3)\end{aligned}$$

The reverse mode algorithm applied to this program can be viewed as being applied recursively from the first line onwards, where the lines responsible for derivative accumulation are prepended

$$\begin{aligned} x &= f(a) && (1a)\\ &\cdots \\ \delta a &\mathrel{+}= \partial f(a)\cdot\delta x && (1b)\end{aligned} \quad\rightarrow\quad \begin{aligned} x &= f(a) && (1a)\\ y &= g(x,b) && (2a)\\ &\cdots \\ \delta x &\mathrel{+}= \partial_L g(x,b)\cdot\delta y && (2b)\\ \delta b &\mathrel{+}= \partial_R g(x,b)\cdot\delta y && (2b)\\ \delta a &\mathrel{+}= \partial f(a)\cdot\delta x && (1b)\end{aligned} \quad\rightarrow\quad \begin{aligned} x &= f(a) && (1a)\\ y &= g(x,b) && (2a)\\ z &= h(y,x) && (3a)\\ \delta y &\mathrel{+}= \partial_L h(y,x)\cdot\delta z && (3b)\\ \delta x &\mathrel{+}= \partial_R h(y,x)\cdot\delta z && (3b)\\ \delta x &\mathrel{+}= \partial_L g(x,b)\cdot\delta y && (2b)\\ \delta b &\mathrel{+}= \partial_R g(x,b)\cdot\delta y && (2b)\\ \delta a &\mathrel{+}= \partial f(a)\cdot\delta x && (1b)\end{aligned}$$

where the ellipsis represents the program yet to be consumed. This formulation can be used to write a reverse mode handler, and we believe was first recorded by [K. C. Sivaramakrishnan 2018], which itself was inspired by the approach of [F. Wang, Zheng, et al. 2019] based on delimited control operators. Our implementation is essentially that of K. C. Sivaramakrishnan with the addition of more operations and the modular approach using functors we are instituting.

We begin by defining a new data type of paired numbers where the derivative is mutable. Next, we define a functor which takes a `SMOOTH` module, which includes the `Smooth` modules as before. The handler we define dynamically creates the reverse pass while handling through control flow by running code after resuming the captured continuation. Finally, we define a helper function to calculate derivatives.

```
1  open Effect.Deep (* Access effect and (deep) handler interface  *)
2  open Smooth      (* Smooth function effect and helper functions *)
3
4  type 't mpaired = {v : 't; mutable dv : 't} (* Value with mutable derivative *)
5
6  (* Perform reverse mode w.r.t. an interpretation of reals given by T *)
7  module Reverse (T : SMOOTH) = struct
8    include Smooth (struct type t = T.t mpaired end)
9
10   (* Handler for reverse mode *)
11   let (reverse : (unit, unit) handler) = {
12     retc = (fun x -> x); (* Do nothing with returned value  *)
13     exnc = raise;        (* Re-raise encountered exceptions *)
14     effc = (fun (type a) (eff : a Effect.t) ->
15       match eff with
16       | Ap0 n -> Some (fun (k : (a, _) continuation) -> let open T in
17           continue k {v = ap0 n; dv = c 0.0} (* r = n,  δr = 0 *)
18         )
19       | Ap1 (u, x) -> Some (fun k -> let open T in
20           let r = {v = ap1 u x.v; dv = c 0.0} in (* r = u(x),   δr = 0 *)
21           continue k r;                          (* Rest of the program *)
22           x.dv <- x.dv +. (der1 u x.v *. r.dv)   (* δx += ∂u(x) · δr  *)
23         )
24       | Ap2 (b, x, y) -> Some (fun k -> let open T in
25           let r = {v = ap2 b x.v y.v; dv = c 0.0} in  (* r = b(x, y),   δr = 0 *)
26           continue k r;                               (* Rest of the program *)
27           x.dv <- x.dv +. (der2 b L x.v y.v *. r.dv); (* δx += ∂_L b(x, y) · δr  *)
28           y.dv <- y.dv +. (der2 b R x.v y.v *. r.dv)  (* δy += ∂_R b(x, y) · δr  *)
29         )
30       | _ -> None
31     )
32   }
33
34   (* grad f x = ∂f(z)/∂z(x)  *)
35   let grad (f : T.t mpaired -> T.t mpaired) (x : T.t) =
36     let r = {v = x; dv = T.c 0.0} in
37     (* Set the output derivative to 1 to get derivative of f *)
38     match_with (fun x -> (f x).dv <- T.c 1.0) r reverse;
39     r.dv
40 end
```

Line 4 defines the paired data type, line 7 defines the `Reverse` module, which takes a `SMOOTH` module. We define the `reverse` handler from line 11. Each case in the effect clause implements the reverse mode rule. The calls to `continue` on line 21 and line 26 run the remainder of the program, the ellipsis in our example. Finally, we define a helper function `grad` on line 35. To calculate the derivative of `f`, we must set its derivative to 1 on line 37. Note again that `f` must be defined only using the combinators provided by `Smooth` and not by destructuring the `mpaired` data type.

The following is an example of how to use `Reverse`.

```
let _ =
  let module E = Evaluate in
  let module R = Reverse(E) in (* Instantiate reverse mode with floats   *)
```

```
4   let sqr x = R.(x *. x) in     (* Square argument using operation from R *)
5   let res = match_with
6     (fun (twice, y) -> R.grad (fun x -> if twice then sqr (sqr x) else sqr x) y)
7     (true, 5.0)
8     E.evaluate
9   in
10  Printf.printf "%f\n" res (* Prints "500.000000"= $4 \cdot 5^3 = \frac{\partial(x^4)}{\partial x}(5)$) *)
```

Line 3 instantiates `Reverse` with `Evaluate`, allowing the `grad` function to be handled with `evaluate` and produce a `float` result. Our next example shows that we can also instantiate `Reverse` with `Forward` calculate second derivatives.

```
let _ =
  let module E = Evaluate in
  let module F = Forward(E) in  (* Instantiate forward mode with floats       *)
  let module RF = Reverse(F) in (* Instantiate reverse mode with forward mode *)
  let sqr x = RF.(x *. x) in    (* Square argument using operation from RF    *)
  let res = match_with (fun (twice, z) ->
      F.diff (fun y ->
        RF.grad (fun x -> if twice then sqr (sqr x) else sqr x) y
      ) z
    ) (true, 5.0) E.evaluate
  in
  Printf.printf "%f\n" res (* Prints "300.000000"= $12 \cdot 5^2 = \frac{\partial^2(x^4)}{\partial x^2}(5)$ *)
```

The module `RF` implements the so-called *forward-over-reverse* mode. It is often used to calculated the product of Hessian matrices with vectors; the Hessian is as to second derivatives as the Jacobian is to first derivatives. Finally, it is of course possible to do arbitrary compositions of forward and reverse mode together.

3.5 Taped Reverse Mode

Our reverse mode handler accumulates to derivatives after continuing the computation. The structure of the accumulations is the same each time, and so easily transformed into data. Thus, we can defer these accumulations during handling by recording their need in a data structure and then run them in the correct order after handling the effectful program. The data structure is called the *tape* and method of explicitly recording deferred derivative accumulations into a tape is called *taping*. Thus, the tape records the data dependency of operations, and so is essentially a directed acyclic graph recorded as a list of nodes in a topological sort defined by execution. We will see that our taped reverse mode handler calls `continue` in the tail position, which can enable optimizations. Furthermore, there are special cases when the tape from a particular execution can be re-used, thereby saving computation.

In order to record the dependencies in the derivative accumulations, we define a new effect for fresh name generation. A name is merely a wrapper around an `int`, and we will use these integers to index into array when calculating the deferred accumulations.

```
open Effect.Deep (* Access effect and (deep) handler interface  *)
open Effect      (* Ditto, contains `perform`                   *)
open Smooth      (* Smooth function effect and helper functions *)
open Array       (* Mutable arrays                              *)

type name = {get_value : int} (* Name data type for fresh names *)

```

```
module type FRESH = sig (* Module type for fresh name effect *)
  type _ Effect.t += Fresh : unit -> name Effect.t (* Generate fresh name *)
  val fresh : unit -> name (* Helper function *)
end

module Fresh : FRESH = struct
  type _ Effect.t += Fresh : unit -> name Effect.t
  let fresh () = perform (Fresh ())
end
```

Next, we define a new paired type of a value with a named derivative. The name will be used to record dependencies on the tape. As an optimization, we make this name `optional`, where a value of none is used for constants and values which transitively depend only on constants. We do not need to calculate their derivatives as they are always 0.

```
type 't npaired = {v : 't; dv : name option} (* Value with named derivative *)
```

The tape itself will be a list of deferred accumulations, and so we define a `defer` data type to be its elements. Because we have unary and binary operations, we either defer a single or doubles dependency; binary operations can depend on one non-constant derivative and so can also have a single dependency.

```
type 't defer (* Defer an accumulation while recording dependency and value *)
  = Single of name * 't             (* Single dependency, save derivative  *)
  | Double of name * name * 't * 't (* Double dependency, save derivatives *)
```

We can now begin to define taped reverse mode. We define a simple handler for generating fresh names that maintains a mutable `int` counter, incrementing it each time a new name is generated, and returning its final value when the handled computation returns.

```
(* Perform taped reverse mode w.r.t. an interpretation of reals given by T *)
module Reverse_tape (T : SMOOTH) = struct
  include Smooth (struct type t = T.t npaired end)
  include Fresh (* Access the fresh effect *)

  let increment_name (init : int) = (* Handle fresh names by incrementing *)
    let i = ref init in { (* Create counter to track generated names *)
      retc = (fun x -> (!i, x)); (* Return updated counter with value *)
      exnc = raise;              (* Re-raise encountered exceptions   *)
      effc = (fun (type a) (eff : a Effect.t) ->
        match eff with
        | Fresh () -> Some (fun (k : (a, _) continuation) ->
            let t = !i in              (* Get fresh value  *)
            i := !i + 1;               (* Update value     *)
            continue k {get_value = t} (* Return fresh name*)
          )
        | _ -> None
      )
  }
```

The taped reverse mode handler begins by allocating the tape, which will be returned when the handled computation completes. For each operation being handled, we check if each argument it transitively dependant solely on constants, and for those which are not, we prepend the operations dependency on said arguments along with the pertinent derivative. We then continue the computation after creating a fresh derivative for the result of the operation.

```
44 let reverse () = (* Handler for taped reverse mode *)
45   (* Initialize a mutable tape, i.e. list of dependencies via `defer`s *)
46   let tape : T.t defer list ref = ref [] in let open Fresh in {
47     retc = (fun x -> (!tape, x)); (* Return updated tape with value  *)
48     exnc = raise;                 (* Re-raise encountered exceptions *)
49     effc = (fun (type a) (eff : a Effect.t) ->
50       match eff with
51       | Ap0 n -> Some (fun (k : (a, _) continuation) -> let open T in
52           continue k {v = ap0 n; dv = None} (* Calculate value, no dep. *)
53         )
54       | Ap1 (u, x) -> Some (fun k -> let open T in
55           let res = ap1 u x.v in (* Calculate value *)
56           match x.dv with
57           | None -> continue k {v = res; dv = None} (* No dependency *)
58           | Some nx ->
59             (* Do $\delta x \mathrel{+}= \partial u(x) \cdot \delta r$ later *)
60             tape := Single (nx, der1 u x.v) :: (!tape);
61             continue k {v = res; dv = Some (fresh ())} (* New derivative *)
62         )
63       | Ap2 (b, x, y) -> Some (fun k -> let open T in
64           let res = ap2 b x.v y.v in (* Calculate value *)
65           match (x.dv, y.dv) with
66           | (None, None) -> continue k {v = res; dv = None} (* No dep. *)
67           | (Some nx, None) ->
68             (*  Do  $\delta x \mathrel{+}= \partial_L b(x, y) \cdot \delta r$ later *)
69             tape := Single (nx, der2 b L x.v y.v) :: (!tape);
70             continue k {v = res; dv = Some (fresh ())} (* New derivative *)
71           | (None, Some ny) ->
72             (* Do $\delta y \mathrel{+}= \partial_R b(x, y) \cdot \delta r$ later *)
73             tape := Single (ny, der2 b R x.v y.v) :: (!tape);
74             continue k {v = res; dv = Some (fresh ())} (* New derivative *)
75           | (Some nx, Some ny) ->
76             tape := (* Do $\delta y \mathrel{+}= \partial_R b(x, y) \cdot \delta r$ and $\delta y \mathrel{+}= \partial_R b(x, y) \cdot \delta r$ later *)
77               Double (nx, ny, der2 b L x.v y.v, der2 b R x.v y.v) :: (!tape);
78             continue k {v = res; dv = Some (fresh ())} (* New derivative *)
79         )
80       | _ -> None
81     )
82 }
```

We begin by allocating reference cell containing a new tape on line 46 for each use of the handler. The tape is returned on line 47 in the return clause. Handling nullary operations just calculates the necessary value. For unary operations, we have two cases, either no dependency or a single dependency. In the later case, we record the deferred accumulation on line 60 and return the result paired with a fresh named derivative. For binary operations, we have four cases for dependencies. We record deferred accumulations as necessary, for example on line line 77 we record a double dependency.

The taped reverse mode handler requires more from its helper function than our previous modes. For a given computation, we will run it to produce a tape and count m of derivatives created, and then execute the recorded accumulations. To do so, we use a mutable array of size m initialized to 0 in which to accumulate. We then set the final derivative to 1 as in reverse mode and iterate over the deferred accumulations, performing them.

```
84    (* grad f x = $\frac{\partial f(z)}{\partial z}$(x) *)
85    let grad (f : T.t npaired -> T.t npaired) (x : T.t) =
86      let (m, (tape, _)) = (* Get number of derivatives and deferred operations *)
87        match_with (fun () -> (* Fresh name handler by incrementing from 0 *)
88          match_with f {v = x; dv = Some (Fresh.fresh ())} (reverse ())
89        ) () (increment_name 0)
90      in
91      (* Initialize array of derivatives to 0 for each *)
92      let ds = init (m : int) (fun _ -> T.c 0.0) in
93      ds.(m - 1) <- T.c 1.0; (* Set derivative of $f(x)$ to 1 *)
94      (* Iterate through the tape with index and perform deferred operations *)
95      List.iteri (fun (k : int) (p : T.t defer) -> let open T in
96        match p with (* Account for the effect of the k-th derivative *)
97        | Single (nu, vu) -> (* Do  $\delta u \mathrel{+}= v_u \cdot \delta k$ *)
98          let dk = ds.(m - (k + 1)) in (* Tape is in reverse, `ds` is not *)
99          let du = ds.(nu.get_value) in
100         ds.(nu.get_value) <- (du +. (vu *. dk))
101       | Double (nl, nr, vl, vr) -> (* Do  $\delta l \mathrel{+}= v_l \cdot \delta k$ and $\delta r \mathrel{+}= v_r \cdot \delta k$ *)
102         let dk = ds.(m - (k + 1)) in (* Tape is in reverse, `ds` is not *)
103         let dl = ds.(nl.get_value) in
104         ds.(nl.get_value) <- (dl +. (vl *. dk));
105         let dr = ds.(nr.get_value) in
106         ds.(nr.get_value) <- (dr +. (vr *. dk))
107     ) (tape : T.t defer list);
108     ds.(0) (* Derivative of $x$, was the first `fresh` *)
109 end
```

We execute the given function on line 88 using our defined handlers, producing the count m and tape. Line 92 initializes the mutable array of derivatives to 0, directly followed by the setting of the output derivative to 1. Next, we iterate over the tape on line 95, using the saved names and values to accumulate into the array of derivatives. Finally, on line 108 we return the derivative of the input variable.

We can use the `Reverse_tape` functor just as before. One useful aspect of taped reverse mode is that it makes clear the dependence of memory allocation with respect to the number of smooth operations. Namely, for n handled operations we must allocate $O(n)$ memory for the derivatives, both via the tape and via derivative array. Thus, one is limited by the available memory of the system on which the computation is being run. A solution to this issue is to not create the entire tape simultaneously, and to instead create portions of it. To do so, portions of the computation must be executed multiple times. The resulting algorithm, called *checkpointed reverse mode*, lowers the maximum memory needed at the cost of increased computation.

3.6 Checkpointed Reverse Mode

We will focus on user specified checkpointing, i.e. the user must choose what portion of the program should be recomputed in order to save memory. For an in-depth explanation, we recommend [Hascoët and Araya-Polo 2006]. Checkpointing without user annotation is possible, see [Jeffrey Mark Siskind and Barak A. Pearlmutter 2018], and we leave it as future work. Furthermore, we will implement a checkpointed reverse mode with an implicit reverse pass as in section 3.4, as we believe is it more succinct and clear.

Our implementation begins with the definition of a new pair type.

```
open Effect.Deep (* Access effect and (deep) handler interface  *)
open Effect      (* Ditto, contains `perform`                   *)
open Smooth      (* Smooth function effect and helper functions *)

type 't rpaired = {v : 't; dv : 't ref} (* Value with ref. of derivative *)
```

In order to make clear when memory is being allocated, the `rpaired` type stores the mutable derivative as a `ref`. Next, we define a new checkpoint effect and helper function.

```
module type CHECKPOINT = sig (* Module type for checkpoint effect *)
  type t
  type _ Effect.t += Checkpoint : (unit -> t rpaired) -> t rpaired Effect.t
  val checkpoint : (unit -> t rpaired) -> t rpaired
end

module Checkpoint (T : sig type t end) : CHECKPOINT with type t = T.t = struct
  type t = T.t
  type _ Effect.t += Checkpoint : (unit -> t rpaired) -> t rpaired Effect.t
  let checkpoint p = perform (Checkpoint p)
end
```

Note that the argument to `checkpoint` is a computation. The intended semantics is that `checkpoint p` produces the same result as `p ()`, so that the use of `checkpoint` only changes behavior related to the reverse pass.

We now define checkpointed reverse mode, which will consist of two handlers, one which does not generate a reverse pass and one which does. The first handler is essentially the evaluate handler of section 3.2.

```
(* Perform checkpointed reverse mode w.r.t. T *)
module Reverse_checkpoint (T : SMOOTH) = struct
  include Smooth (struct type t = T.t rpaired end)
  include Checkpoint (struct type t = T.t end)

  let rec evaluate (s : t ref) = { (* Handle checkpoint without reverse pass *)
    retc = (fun x -> x); (* Do nothing with returned value  *)
    exnc = raise;        (* Re-raise encountered exceptions *)
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | Ap0 n -> Some (fun (k : (a, _) continuation) -> let open T in
          continue k {v = ap0 n; dv = s}
        )
      | Ap1 (u, x) -> Some (fun k -> let open T in
          continue k {v = ap1 u x.v; dv = s}
        )
      | Ap2 (b, x, y) -> Some (fun k -> let open T in
          continue k {v = ap2 b x.v y.v; dv = s}
        )
      | Checkpoint p -> Some (fun k ->
          (* Recursively run other checkpoints without reverse pass *)
          let {v = res; dv = _} = match_with p () (evaluate s) in
          continue k {v = res; dv = s}
        )
```

```
43        | _ -> None
44      )
45    }
```

Unlike previous handlers, `evaluate` recursively calls itself (line 40) due to the need of handling smooth operations and nested checkpoint effects in checkpointed code. Deep handlers only recursively handle effects encountered through resuming the caught continuation `k`. Other kinds of effect and handler systems, such as the scoped effects of [Wu et al. 2014; Yang et al. 2022] or the higher-order effects of [B. v. d. Berg and Schrijvers 2023], may be able to express the requirement of `evaluate` handling the effects of `checkpoint`'s argument. We also pass in a reference `s` to act as a dummy value. A more verbose alternative would be to use an `option` type in `rpaired`.

We now define the checkpointed reverse mode handler, which is also recursive. The handling of smooth functions is the same as in reverse mode except for being adapted for references. The handling of the checkpoint effect has the same general structure as smooth functions: calculate the value of the function and allocate a derivative, run the rest of the program, and finally accumulate into the derivatives of the inputs.

```
47    let rec reverse () = { (* Handler for checkpointed reverse mode *)
48      retc = (fun x -> x); (* Do nothing with returned value  *)
49      exnc = raise;        (* Re-raise encountered exceptions *)
50      effc = (fun (type a) (eff : a Effect.t) ->
51        match eff with
52        | Ap0 n -> Some (fun (k : (a, _) continuation) -> let open T in
53            continue k {v = ap0 n; dv = ref (c 0.0)}
54          )
55        | Ap1 (u, x) -> Some (fun k -> let open T in
56            let r = {v = ap1 u x.v; dv = ref (c 0.0)} in
57            continue k r;
58            x.dv := !(x.dv) +. (der1 u x.v *. !(r.dv))
59          )
60        | Ap2 (b, x, y) -> Some (fun k -> let open T in
61            let r = {v = ap2 b x.v y.v; dv = ref (c 0.0)} in
62            continue k r;
63            x.dv := !(x.dv) +. (der2 b L x.v y.v *. !(r.dv));
64            y.dv := !(y.dv) +. (der2 b R x.v y.v *. !(r.dv))
65          )
66        | Checkpoint p -> Some (fun k -> let open T in
67            let s = ref (c 0.0) in
68            (* Get result of checkpoint without creating reverse pass *)
69            let res = match_with (p : unit -> t rpaired) () (evaluate s) in
70            let r = {v = res.v; dv = ref (c 0.0)} in
71            continue k r; (* Rest of the program *)
72            match_with (fun () -> (* Create and run reverse pass for checkoint *)
73              let {v = _; dv = dres} = p () in
74              dres := !(r.dv) (* Propagate result of `checkpoint` reverse pass *)
75            ) () (reverse ())
76          )
77        | _ -> None
78      )
79    }
```

Let us focus on the `Checkpoint` case. Line 69 calculates the value result of executing `p` by using the `evaluate` handler. The subsequent line then allocates a derivative for said result. We then continue the rest of the program, which as we have seen generates a portion of the reverse pass. After the remainder of the program has been handled, we call p again on line 73 to generate the reverse pass through a recursive use of `reverse`. Importantly, on line 74, we propagate the accumulations of this reverse pass by setting the derivative created for p's reverse pass to the previously allocated derivative.

Finally, the helper function is analogous to the standard reverse mode function.

```
81    (* grad f x = $\frac{\partial f(z)}{\partial z}$(x) *)
82    let grad (f : t rpaired -> t rpaired) (x : t) =
83      let r = {v = x; dv = ref (T.c 0.0)} in
84      match_with (fun x -> (f x).dv := T.c 1.0) r (reverse ());
85      !(r.dv)
86  end
```

The following is an example of how to use checkpointed reverse mode. We have written a program which should be equivalent to $x^2 + 3x + 2$, which has derivative $2x + 3$.

```
let _ =
  let module E = Evaluate in
  let module R = Reverse_checkpoint(E) in
  let res = match_with (R.grad (fun x -> let open R in
    let y = c 2.0 in                            (* $y = 2$ *)
    let z = checkpoint (fun () -> x +. y) in    (* $z = x + 2$ *)
    let a = checkpoint (fun () ->
      let w = checkpoint (fun () -> x *. z) in (* $w = x^2 + 2x$ *)
      w +. y                                    (* $a = x^2 + 2x + 2$ *)
    ) in
    a +. x                                      (* $x^2 + 3x + 2$ *)
  )) 5.0 E.evaluate in
  Printf.printf "%f\n" res (* Prints "13.000000" $= 2(5) + 3 = \frac{\partial(x^2+3x+2)}{\partial x}(5)$ *)
```

Values are passed into checkpointed code via closures. Furthermore, we see that the algorithm supports nested checkpointing. We also note that like previous modes, checkpointed reverse mode can be combined with other modes.

4 BENCHMARKS

All benchmarks have been run on a Dell Precision T3600 with a quad core (3.60 GHz boost) Intel Xeon E5-1620, 4 × 8GB = 32GB 1600 MHz DDR4, and 256 GB SATA 6 Gb/s SSD. The operating system used is headless Debian 12 (bookworm) with Linux kernel release 6.1.0-18-amd64.

4.1 Asymptotic Benchmarks

An important aspect of AD is the asymptotic behavior. [Griewank and A. Walther 2008, Sec. 4.4] show that for a composite measure of "work", both forward and reverse mode only need perform bounded constant multiple more work than the original program. Their measure of work accounts for four categories: memory fetches and stores, additions and subtractions, multiplications, and non-linear operations. Paired with reasonable assumptions, they then prove that forward mode applied to a program should be between 2 to 2.5 times slower than the original program, and reverse mode should be between 3 to 4 times slower. The behavior of checkpointed reverse mode is more complicated due to its ability to trade space for time. We will thus examine forward mode, reverse mode, and taped reverse mode for correct performance.

Thus, we would like to show that our implementations differentiate with only a constant multiple slowdown, and that this holds across different problem sizes. To do so, we create a simple program with an input n such that the number of smooth operations invoked is directly proportional to n. Thus, graphing time t against n should produce a line, and if another program takes time $c \cdot t$, then it is also a line when graphed against n. Our simple program will approximate the the Taylor series of $\frac{1}{x}$ around 1, i.e. it will approximate the right-hand side of

$$\frac{1}{x} = \sum_{n=0}^{\infty} (-1)^n (x-1)^n$$

which converges when $|x - 1| < 1$. Let a_n denote the n^{th} term of the above series. Then we have the recurrence

$$a_0 = 1, \qquad a_n = -(x-1) \cdot a_{n-1}$$

and so we can iteratively generate a_n as shown below:

```
open Smooth

module Taylor_Recip_Benchmark (T : SMOOTH) = struct
  let approx_recip iters x = let open T in
    let prev = ref (c 1.0) in (* $a_0$ *)
    let acc = ref (c 1.0) in  (* $\sum_{n=0}^{0} a_n$ *)
    for _i = 1 to iters do
      prev := !prev *. (~. (x -. (c 1.0))); (* $a_{\_i} = -(x - 1) \cdot a_{\_i-1}$ *)
      acc := !prev +. !acc                  (* $\sum_{n=0}^{\_i} a_n = a_{\_i} + \sum_{n=0}^{\_i-1} a_n$ *)
    done;
    !acc (* $\sum_{n=0}^{\mathtt{iters}} a_n$ *)
end
```

Each iteration of the loop executes five smooth operations. Therefore, the number of operations and thus the time to execute should be directly proportional to `iters`, and thus each algorithm applied to `apporx_recip` should be directly proportional if our implementations have the correct behavior. We then create an executable which takes `iters` as a command line argument, e.g.

```
let _ =
  (* Increase the minor heap size to 500MiB to stop quadratic behaviour in
     reverse mode due to deep callstack. 1MiB = 1048576. *)
  Gc.set { (Gc.get ()) with Gc.minor_heap_size = (500 * 1048576)};
  let iters = int_of_string Sys.argv.(1) in
  let module E = Evaluate in
  let module R = Reverse(E) in
  let module T = Taylor_Recip_Benchmark(R) in
  let res = match_with (R.grad (T.approx_recip iters)) 0.5 E.evaluate in
  Printf.printf "%f\n" res
```

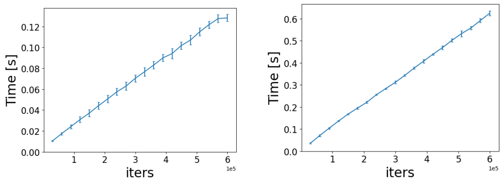
The above example is straightforward except for the change in the garbage collector (GC) minor heap size parameter. Reverse mode creates a deep call stack which is long-lived (the length of the entire program), causing stack scans by the GC to add a linear overhead. By increasing the minor heap size, this issue is alleviated. The needed minor heap size increases proportional to `iters`, and we have chosen a suitable value for our tested range. It is also possible to change the behavior of OCaml 5.0's GC, but that is out of our scope here[3].

To analyze the runtime of each of the created programs, we execute the program with values of `iters` from $30,000$ to $600,000$ in increments of $30,000$. For each value of `iters`, we first run the program ten times as a warmup, and then at least a further ten times to collect timing data. We then

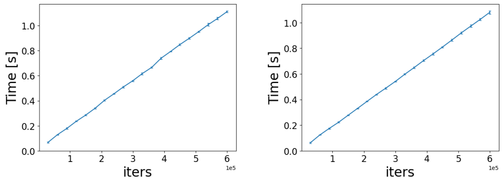
---

[3] Thanks to REDACTED FOR PEER REVIEW who diagnosed the issue and suggested the used solution.

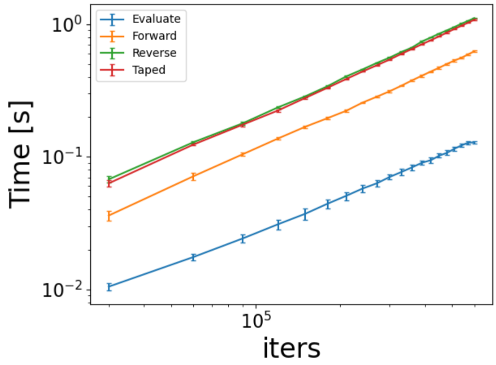
plot the mean of the collected times with symmetric error bars showing the standard deviation. Furthermore, to aid comparison across modes, we graph all four modes together using logarithmic scales on both axes. The results of this process are collected in fig. 1.



(a) Evaluation mode (b) Forward mode



(c) Reverse mode (d) Taped reverse mode



(e) Log-log comparison of all modes

Fig. 1. Benchmark results

Figures 1a to 1d show the results of each individual mode. As desired, fig. 1a shows the execution time of evaluation mode is directly proportional to `iters`. Furthermore, figs. 1b to 1d show that the other have execution time directly proportional to `iters`, meaning that each mode is only a constant time slower than evaluation across all values of `iters`. Finally, fig. 1e shows that forward and both reverse modes are within one order of magnitude slower than evaluation mode, forward mode is approximately 4.6× slower than evaluation and each reverse mode is approximately 8.3× slower. We have not reached the theoretical optimal bounds derived by [Griewank and A. Walther 2008] of 2.5× and 4×, but we are not far off. Therefore, we claim that our AD modes are performant enough to be practical. We will now strengthen this claim with a real world example.

4.2 Real World Benchmarks

We claim that our implementation of AD via effects and handlers is performant with respect to comparable implementations. By comparable, we mean CPU based, as we do not use GPU based computation, and dynamic, as static approaches are almost always faster through code generation and optimization. The dynamic approach is often referred to as *eager mode*, for example by PyTorch and TensorFlow. To substantiate our claim, we will use the benchmark suite of [Šrajer et al. 2018b][4].

The suite of Šrajer et al. is reproducible, extensible, realistic, and expansive. It is reproducible through the use of containerization, ensuring that the same version of each tool is used across runs and compilations. Extensibility is achieved through a documented test harness and modular design. The four computations which they benchmark are real world functions which are optimized against in machine learning and computer vision. Additionally, the current iteration of their system supports thirteen different implementations across five languages, including a baseline of finite differences[5] and manually implemented derivatives. Finally, the computed derivatives are checked for correctness against a known correct implementation.

---

[4] A longer preprint is available [Šrajer et al. 2018a] and the base suite is available at https://github.com/microsoft/ADBench.

[5] Finite differences approximate the derivative via, for example, ${}^{\partial f(x)}/_{\partial x}(y) \cong {}^{f(y+\epsilon)-f(y)}/_{\epsilon}$, which holds for small ϵ.

The full methodology can be found in their paper and the repository[6], we will highlight the important aspects here. For each implementation and set of parameters, essentially the following is carried out:

- Read the input data and convert it into a consumable format.
- Run any needed preparation code which is not AD.
- For both computation of the objective function and its gradient:
  - Find the number of times r needed to run to reach a prescribed minium time.
  - Run n lots of r computations, find the average time for each lot.
  - Pick the minium average time of from the n lots to alleviate noise.
- Save the times recorded and the numerical results to check correctness.

We have chosen one of their four computations to implement, namely the objective function used for the fitting of Gaussian mixture models. Specifically, let $m, N, K, D \in \mathbb{N}$ and let $1 \leq i \leq N$ and $1 \leq k \leq K$. We use $|| \cdot ||$ for Euclidean norm, and use an unspecified function $Q \colon \mathbb{R}^D \times \mathbb{R}^{D(D-1)/2} \longrightarrow \mathbb{R}^{D \times D}$ which creates a $D \times D$ lower triangular matrix. Then for vectors $\boldsymbol{x}_i \in \mathbb{R}^D$, $\boldsymbol{q}_k \in \mathbb{R}^D$, $\boldsymbol{l}_k \in \mathbb{R}^{D(D-1)/2}$, $\boldsymbol{\mu}_k \in \mathbb{R}^D$, and $\boldsymbol{\alpha} \in \mathbb{R}^K$, we define

$$
\begin{aligned}
L(\boldsymbol{\alpha}, \mathbf{M}, \mathbf{Q}, \mathbf{L}) \coloneqq \sum_{i=1}^{N} \operatorname{logsumexp}\left(\left[\alpha_k + \operatorname{sum}(\boldsymbol{q}_k) - \frac{1}{2} ||Q(\boldsymbol{q}_k, \boldsymbol{l}_k)(\boldsymbol{x}_i - \boldsymbol{\mu}_k)||^2 \right]_{k=1}^{K} \right) \\
- N \operatorname{logsumexp}\left([\alpha_k]_{k=1}^{K} \right) \qquad (1) \\
+ \frac{1}{2} \sum_{k=1}^{K} \left(|| \exp(\boldsymbol{q}_k)||^2 + ||\boldsymbol{l}_k||^2 \right) - m \operatorname{sum}(\boldsymbol{q}_k)
\end{aligned}
$$

where we have matrices $\mathbf{M} \coloneqq [\boldsymbol{\mu}_k]_{k=1}^{K}$, $\mathbf{Q} \coloneqq [\boldsymbol{q}_k]_{k=1}^{K}$, and $\mathbf{L} \coloneqq [\boldsymbol{l}_k]_{k=1}^{K}$. The derivation of this objective function and the definition of Q can be found in [Šrajer et al. 2018b]. The variables $\boldsymbol{\alpha}$, $\mathbf{M}$, $\mathbf{Q}$, and $\mathbf{L}$ are the independent variables which we must find the derivatives of, where the $\boldsymbol{x}_i$'s have a fixed value. The dimensions of the independent variables will change depending on N, K, and D and the total number of independent variables will be the increasing parameter which we measure time against.

We implement the above function using the Owl scientific computing library [L. Wang et al. 2022][7]. Doing so gives us access to primitive operations such as summation and transposition on tensors (n-dimensional arrays). Thus, our family of smooth functions can now include tensor-valued operations. Owl itself can perform AD, but we do not use this feature. The new version of `Smooth` can be found in appendix B.1. Consequently, the number of effectful operations greatly decreases, e.g. 999 uses of binary addition for a 1000 element vector versus 1 summation operation, which reduces the overhead of effect handling. Besides the change to operations involving tensors, the structure of reverse mode is the same, which can be seen in appendix B.2.

The results of our implementation along with the other systems is summarized in fig. 2 (and fig. 3 in appendix A) where the x-axis is the number of independent variables and the y-axis is the amount of time to compute the Jacobian, with each axis logarithmic scale. The input data for fig. 2 always has $N = 1,000$, while fig. 3 always has $N = 10,000$, and we note that this does not effect the number of independent variables. In both figures, our implementation is more performant in the long run than: finite differences (C++), Autograd (Python), Zygote (Julia), and pure Julia (Julia). For $N = 1,000$ we are competitive with eager TensorFlow 2.0 (Python), although we are not for $N = 10,000$. Finally, we are competitive with DiffSharp (F#) in both instances. Of the

---

[6] https://github.com/microsoft/ADBench/blob/master/docs/Methodology.md

[7] https://ocaml.xyz/

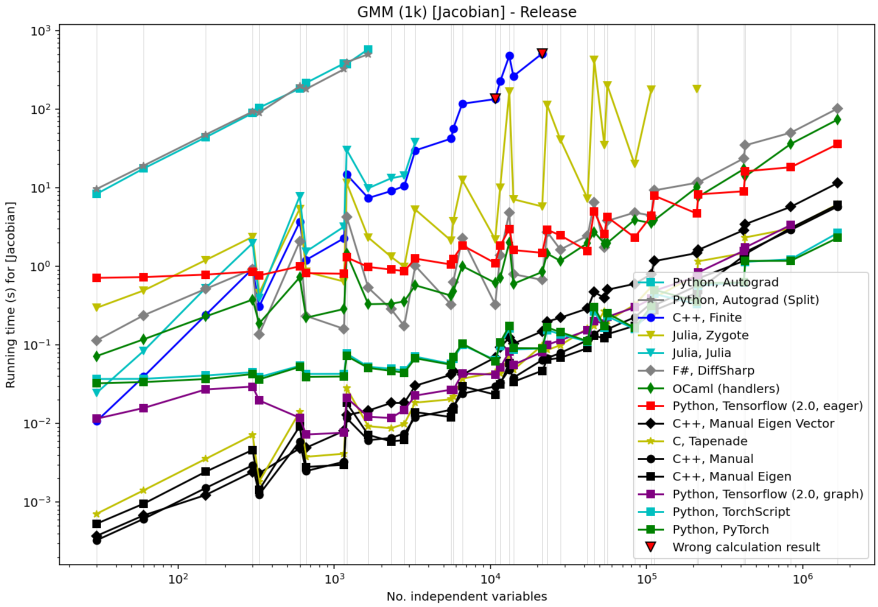


Fig. 2. GMM results, N = 1,000

previous systems, the only define-by-run system we do not out perform is eager Tensorflow. Furthermore, the remaining seven implementations which outperform us are either handcrafted, source transformations, or define-then-run systems. Therefore, we substantiate our claim that we are a competitive define-by-run system.

5 RELATED WORK

AD with Effects and Handlers. There is previous work in implementing AD with effects and handlers as well as proving said implementations correct. The first implementation we are aware of is by [K. C. Sivaramakrishnan 2018], and is of reverse mode AD, which itself was adapted from [F. Wang and Rompf 2018] which used delimited continuations. F. Wang, Zheng, et al. also extended their delimited continuation AD approach in [F. Wang, Zheng, et al. 2019]. Finally, [de Vilhena and Pottier 2023] prove the correctness of an implementation analogous to that of [K. C. Sivaramakrishnan 2018]. They use their separation logic for effects and handlers to prove reverse mode correct with respect to an operational semantics. Another combination of AD with effects and handlers is [Tan et al. 2023], which implements effect handlers for the JAX language [Bradbury et al. 2024]. JAX supports AD and Tan et al. use AD to implement handlers for choice-based learning.

AD and the Programming Language Community. As seen in section 2.2, there is no shortage of AD systems stretching back decades. What is more recent is the interest of the programming language community in AD, catalyzed by [Barak A. Pearlmutter and Jeffrey Mark Siskind 2008] who showed how to implement reverse mode in a functional framework. Elliott [2018] provided a correct-by-construction, categorical combinator based approach to various modes. Another combinator-like approach is described in the string diagram formalism of [Alvarez-Picallo et al. 2021]. Other works have shown how to derive AD modes based on a sequence of program transformations [Krawiec et al. 2022; Radul et al. 2022; T. J. Smeding and M. I. L. Vákár 2023] and algebraic reasoning [B. v. d. Berg et al. 2024]. Much work has been directed towards ensuring efficiency of reverse mode in general purpose languages [Brunel et al. 2019; Krawiec et al. 2022; Radul et al. 2022; T. J. Smeding and M. I. L. Vákár 2024, 2023] as well as in array focused languages [Shaikhha, Fitzgibbon, et al. 2019; Shaikhha, Huot, et al. 2022]. Correctness has also been a focus, for example the sound and complete semantics of [Abadi and G. D. Plotkin 2020] of a first-order language. The inclusion of higher-order functions has been achieved in a number of works [Alvarez-Picallo et al. 2021; Huot et al. 2020, 2022; Sherman et al. 2021; M. Vákár and T. Smeding 2022]. Finally, attention has also been paid to non-differentiable functions. Mazza and Pagani [2021] show that PCF, with allowed primitives, gives the correct derivative with probability 1, and Sherman et al. [2021] describe a language for Lipschitz but nondifferentiable functions with a computable semantics.

6 CONCLUSION AND FUTURE WORK

We have shown how to implement four different AD modes in OCaml 5.0 using effects and handlers, namely forward mode, reverse mode, taped reverse mode, and checkpointed reverse mode. Reverse mode took advantage of the complex control flow that effects and handlers afford to dynamically build a reverse pass. Checkpointed reverse mode made use of the ability of handlers to provide different interpretations of the same program by running checkpointed code in two different manners. Additionally, by structuring our modes as OCaml functors, they composed together to form new modes and compute higher-order derivatives due to the compositionality of handlers. Overall, we provided a framework for modularly defining AD algorithms using effects and handlers.

Importantly, we also analyzed the execution time characteristics of forward mode, reverse mode, and taped reverse mode. The linchpin result of AD is that there is only a constant multiple overheard for computing the derivative compared to the original program. By creating a sample program which performed a variable amount of work, we showed that our implementations satisfied this requirement across problem sizes. Finally, we provided a real world test of absolute performance of reverse mode, and showed that our implementation was competitive with other define-by-run systems.

Future work. We see various avenues for future work:

- The checkpointed reverse mode we implemented is user-driven; only the code explicitly annotated by the user is checkpointed. [Jeffrey Mark Siskind and Barak A. Pearlmutter 2018] describe a checkpointed reverse mode which does not require user annotation. Their divide-and-conquer algorithm requires “splitting a program in half” with respect to execution cost, and then recursing on each half. OCaml 5.0 handlers can only resume a continuation once[8], but so-called *multi-shot* handlers (which can resume multiple times) exist in other languages. Thus, their algorithm is a perfect fit for multi-shot handlers which can run the program once to split it in half with a measurement handler, and then run it again to recurse using a different handler.
- Another interesting algorithm we believe is well suited to effects and handlers is the ADEV algorithm of [Lew et al. 2023], which enables AD to differentiate through the expectation of probabilistic processes. In particular, the same syntactic sampling operation can be interpreted in different ways to achieve different statistical guarantees, a great match for effects and handlers.

---

[8] The `multicont` library (https://github.com/dhil/ocaml-multicont) of Daniel Hillerström allows multi-shot handlers in OCaml 5.0, but with various hazards.

- Our implementations provide a suitable base for studying the interaction of AD and other effects. For example, what interactions should there be between checkpointed reverse mode and non-determinism, or probabilistic sampling?
- Though [de Vilhena and Pottier 2023] have proved handler based reverse mode correct, we believe there is more room for semantic proofs of handler based AD algorithms, which is ongoing work.

REFERENCES

Martín Abadi, Ashish Agarwal, et al.. 2015. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* https://www.tensorflow.org/.

Martín Abadi and Gordon D. Plotkin. Jan. 2020. "A simple differentiable programming language." en. *Proceedings of the ACM on Programming Languages*, 4, POPL, (Jan. 2020), 1–28. DOI: 10.1145/3371106.

Mario Alvarez-Picallo, Dan R. Ghica, David Sprunger, and Fabio Zanasi. July 2021. "Functorial String Diagrams for Reverse-Mode Automatic Differentiation." *arXiv:2107.13433 [cs]*, (July 2021). arXiv: 2107.13433. Retrieved July 29, 2021 from http://arxiv.org/abs/2107.13433.

Andrej Bauer. Mar. 2019. "What is algebraic about algebraic effects and handlers?" en. *arXiv:1807.05923 [cs]*, (Mar. 2019). arXiv: 1807.05923. Retrieved Jan. 31, 2020 from http://arxiv.org/abs/1807.05923.

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Feb. 2018. "Automatic differentiation in machine learning: a survey." en. *arXiv:1502.05767 [cs, stat]*, (Feb. 2018). arXiv: 1502.05767. Retrieved June 26, 2020 from http://arxiv.org/abs/1502.05767.

Birthe van den Berg, Tom Schrijvers, James McKinna, and Alexander Vandenbroucke. Jan. 2024. "Forward- or reverse-mode automatic differentiation: What's the difference?" *Science of Computer Programming*, 231, (Jan. 2024), 103010. DOI: 10.1016/j.scico.2023.103010.

Birthe van den Berg and Tom Schrijvers. Feb. 2023. *A Framework for Higher-Order Effects & Handlers.* arXiv:2302.01415 [cs]. (Feb. 2023). DOI: 10.48550/arXiv.2302.01415.

Michael Betancourt. Dec. 2018. "A Geometric Theory of Higher-Order Automatic Differentiation." en. *arXiv:1812.11592 [stat]*, (Dec. 2018). arXiv: 1812.11592. Retrieved May 22, 2019 from http://arxiv.org/abs/1812.11592.

C. H. Bischof, L. Roh, and A. J. Mauer-Oats. 1997. "ADIC: an extensible automatic differentiation tool for ANSI-C." en. *Software: Practice and Experience*, 27, 12, 1427–1456. DOI: 10.1002/(SICI)1097-024X(199712)27:12<1427::AID-SPE138>3.0.CO;2-Q.

Christian Bischof, Peyvand Khademi, Andrew Mauer, and Alan Carle. Sept. 1996. "Adifor 2.0: Automatic Differentiation of Fortran 77 Programs." *IEEE Computational Science & Engineering*, 3, 3, (Sept. 1996), 18–32. DOI: 10.1109/99.537089.

[SW] James Bradbury et al., *JAX: composable transformations of Python+NumPy programs* version 0.4.25, 2024. URL: http://github.com/google/jax.

Aloïs Brunel, Damiano Mazza, and Michele Pagani. Dec. 2019. "Backpropagation in the simply typed lambda-calculus with linear negation." *Proceedings of the ACM on Programming Languages*, 4, POPL, (Dec. 2019), 64:1–64:27. DOI: 10.1145/3371132.

Ronan Collobert and Koray Kavukcuoglu. 2011. "Torch7: A matlab-like environment for machine learning." In: *In BigLearn, NIPS Workshop*.

Paulo Emílio de Vilhena and François Pottier. Aug. 2023. *Verifying an Effect-Handler-Based Define-By-Run Reverse-Mode AD Library.* arXiv:2112.07292 [cs]. (Aug. 2023). DOI: 10.48550/arXiv.2112.07292.

Conal Elliott. July 2018. "The Simple Essence of Automatic Differentiation." *Proc. ACM Program. Lang.*, 2, ICFP, (July 2018), 70:1–70:29. Number: ICFP. DOI: 10.1145/3236765.

A. Griewank and A. Walther. Jan. 2008. *Evaluating Derivatives.* Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, (Jan. 2008). ISBN: 978-0-89871-659-7. DOI: 10.1137/1.9780898717761.

Laurent Hascoët and Mauricio Araya-Polo. June 2006. "Enabling user-driven Checkpointing strategies in Reverse-mode Automatic Differentiation." *arXiv:cs/0606042*, (June 2006). arXiv: cs/0606042. Retrieved Feb. 28, 2020 from http://arxiv.org/abs/cs/0606042.

Laurent Hascoët and Valérie Pascual. May 2013. "The Tapenade automatic differentiation tool: Principles, model, and specification." *ACM Transactions on Mathematical Software*, 39, 3, (May 2013), 20:1–20:43. DOI: 10.1145/2450153.2450158.

Mathieu Huot, Sam Staton, and Matthijs Vákár. Jan. 2020. "Correctness of Automatic Differentiation via Diffeologies and Categorical Gluing." *arXiv:2001.02209 [cs]*, (Jan. 2020). arXiv: 2001.02209. Retrieved Feb. 19, 2020 from http://arxiv.org/abs/2001.02209.

Mathieu Huot, Sam Staton, and Matthijs Vákár. Mar. 2022. "Higher Order Automatic Differentiation of Higher Order Functions." *Logical Methods in Computer Science*, Volume 18, Issue 1, (Mar. 2022). Publisher: Episciences.org. DOI: 10.46298/lmcs-18(1:41)2022.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. June 2014. “Caffe: Convolutional Architecture for Fast Feature Embedding.” *arXiv:1408.5093 [cs]*, (June 2014). arXiv: 1408.5093. Retrieved July 2, 2020 from http://arxiv.org/abs/1408.5093.

Faustyna Krawiec, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A. Eisenberg, and Andrew Fitzgibbon. Jan. 2022. “Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation.” *Proceedings of the ACM on Programming Languages*, 6, POPL, (Jan. 2022), 48:1–48:30. DOI: 10.1145/3498710.

Charles L. Lawson. Sept. 1971. *Computing Derivatives Using W-Arithmetic and U-Arithmetic*. Internal Computing Memorandum CM–286. Jet Propulsion Laboratory, Pasadena, Calif., (Sept. 1971).

Alexander K. Lew, Mathieu Huot, Sam Staton, and Vikash K. Mansinghka. Jan. 2023. “ADEV: Sound Automatic Differentiation of Expected Values of Probabilistic Programs.” en. *Proceedings of the ACM on Programming Languages*, 7, POPL, (Jan. 2023), 121–153. DOI: 10.1145/3571198.

Damiano Mazza and Michele Pagani. Jan. 2021. “Automatic differentiation in PCF.” *Proceedings of the ACM on Programming Languages*, 5, POPL, (Jan. 2021), 28:1–28:27. DOI: 10.1145/3434309.

Bart van Merriënboer, Alexander B. Wiltschko, and Dan Moldovan. Nov. 2017. “Tangent: Automatic Differentiation Using Source Code Transformation in Python.” *arXiv:1711.02712 [cs, stat]*, (Nov. 2017). arXiv: 1711.02712. Retrieved July 2, 2020 from http://arxiv.org/abs/1711.02712.

Valérie Pascual and Laurent Hascoët. 2008. “TAPENADE for C.” en. In: *Advances in Automatic Differentiation* (Lecture Notes in Computational Science and Engineering). Ed. by Christian H. Bischof, H. Martin Bücker, Paul Hovland, Uwe Naumann, and Jean Utke. Springer, Berlin, Heidelberg, 199–209. ISBN: 978-3-540-68942-3. DOI: 10.1007/978-3-540-68942-3_18.

Barak A Pearlmutter and Jeffrey M Siskind. N.d. “Putting the Automatic Back into AD: Part II, Dynamic, Automatic, Nestable, and Fast (CVS: 1.1).” en, 11.

Barak A Pearlmutter and Jeffrey Mark Siskind. N.d. “Lazy Multivariate Higher-Order Forward-Mode AD.” en, 6.

Barak A. Pearlmutter and Jeffrey Mark Siskind. Mar. 2008. “Reverse-mode AD in a Functional Framework: Lambda the Ultimate Backpropagator.” *ACM Trans. Program. Lang. Syst.*, 30, 2, (Mar. 2008), 7:1–7:36. Number: 2. DOI: 10.1145/1330017.1330018.

F W Pfeiffer. Jan. 1987. “Automatic differentiation in prose.” *ACM SIGNUM Newsletter*, 22, 1, (Jan. 1987), 2–8. DOI: 10.1145/24680.24681.

Gordon Plotkin and John Power. 2001. “Adequacy for Algebraic Effects.” en. In: *Foundations of Software Science and Computation Structures*. Vol. 2030. Ed. by Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Furio Honsell, and Marino Miculan. Series Title: Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–24. ISBN: 978-3-540-41864-1 978-3-540-45315-4. DOI: 10.1007/3-540-45315-6_1.

Gordon Plotkin and Matija Pretnar. 2009. “Handlers of Algebraic Effects.” en. In: *Programming Languages and Systems*. Vol. 5502. Ed. by Giuseppe Castagna. Series Title: Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 80–94. ISBN: 978-3-642-00589-3 978-3-642-00590-9. DOI: 10.1007/978-3-642-00590-9_7.

Matija Pretnar. Dec. 2015. “An Introduction to Algebraic Effects and Handlers. Invited tutorial paper.” en. *Electronic Notes in Theoretical Computer Science*, 319, (Dec. 2015), 19–35. DOI: 10.1016/j.entcs.2015.12.003.

Alexey Radul, Adam Paszke, Roy Frostig, Matthew Johnson, and Dougal Maclaurin. Apr. 2022. *You Only Linearize Once: Tangents Transpose to Gradients*. en. arXiv:2204.10923 [cs]. (Apr. 2022). Retrieved Aug. 2, 2022 from http://arxiv.org/abs/2204.10923.

Frank Seide and Amit Agarwal. Aug. 2016. “CNTK: Microsoft’s Open-Source Deep-Learning Toolkit.” In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (KDD ’16). Association for Computing Machinery, San Francisco, California, USA, (Aug. 2016), 2135. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2945397.

Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. July 2019. “Efficient differentiable programming in a functional array-processing language.” *Proceedings of the ACM on Programming Languages*, 3, ICFP, (July 2019), 97:1–97:30. DOI: 10.1145/3341701.

Amir Shaikhha, Mathieu Huot, Shabnam Ghasemirad, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. Dec. 2022. *Efficient and Sound Differentiable Programming in a Functional Array-Processing Language*. en. arXiv:2212.10307 [cs]. (Dec. 2022). Retrieved Mar. 1, 2023 from http://arxiv.org/abs/2212.10307.

Benjamin Sherman, Jesse Michel, and Michael Carbin. Jan. 2021. “λ_s: computable semantics for differentiable programming with higher-order functions and datatypes.” *Proceedings of the ACM on Programming Languages*, 5, POPL, (Jan. 2021), 3:1–3:31. DOI: 10.1145/3434284.

Jeffrey Mark Siskind and Barak A. Pearlmutter. Nov. 2018. “Divide-and-conquer checkpointing for arbitrary programs with no user annotation.” *Optimization Methods and Software*, 33, 4-6, (Nov. 2018), 1288–1330. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/10556788.2018.1459621. DOI: 10.1080/10556788.2018.1459621.

K. C. Sivaramakrishnan. Feb. 2018. *Reverse-mode Algorithmic differentiation using effect handlers*. en. (Feb. 2018). Retrieved Aug. 29, 2023 from https://github.com/ocaml-multicore/effects-examples/blob/master/algorithmic_differentiation.ml.

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. June 2021. “Retrofitting effect handlers onto OCaml.” In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (PLDI 2021). Association for Computing Machinery, New York, NY, USA, (June 2021), 206–221. ISBN: 978-1-4503-8391-2. DOI: 10.1145/3453483.3454039.

Tom J. Smeding and Matthijs I. L. Vákár. Jan. 2024. “Efficient CHAD.” *Proceedings of the ACM on Programming Languages*, 8, POPL, (Jan. 2024), 36:1060–36:1088. DOI: 10.1145/3632878.

Tom J. Smeding and Matthijs I. L. Vákár. Jan. 2023. “Efficient Dual-Numbers Reverse AD via Well-Known Program Transformations.” *Proceedings of the ACM on Programming Languages*, 7, POPL, (Jan. 2023), 54:1573–54:1600. DOI: 10.1145/3571247.

Bert Speelpenning. 1980. “Compiling fast partial derivatives of functions given by algorithms.” Ph.D. University of Illinois at Urbana-Champaign, USA. AAI8017989.

Filip Šrajer, Zuzana Kukelova, and Andrew Fitzgibbon. July 2018a. *A Benchmark of Selected Algorithmic Differentiation Tools on Some Problems in Computer Vision and Machine Learning*. arXiv:1807.10129 [cs]. (July 2018). DOI: 10.48550/arXiv.1807.10129.

Filip Šrajer, Zuzana Kukelova, and Andrew Fitzgibbon. Nov. 2018b. “A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning.” *Optimization Methods and Software*, 33, 4-6, (Nov. 2018), 889–906. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/10556788.2018.1435651. DOI: 10.1080/10556788.2018.1435651.

Shangyin Tan, Dan Zheng, Gordon Plotkin, and Ningning Xie. Dec. 2023. “Choice-Based Learning in JAX.” en. In: (Dec. 2023). Retrieved Feb. 29, 2024 from https://openreview.net/forum?id=wkAFNdzhli.

Joe M. Thames. Aug. 1969. “SLANG a problem solving language for continuous-model simulation and optimization.” In: *Proceedings of the 1969 24th national conference* (ACM ’69). Association for Computing Machinery, New York, NY, USA, (Aug. 1969), 23–41. ISBN: 978-1-4503-7493-4. DOI: 10.1145/800195.805913.

Theano Development Team. May 2016. “Theano: A Python framework for fast computation of mathematical expressions.” *arXiv e-prints*, abs/1605.02688, (May 2016). http://arxiv.org/abs/1605.02688.

Matthijs Vákár and Tom Smeding. June 2022. *CHAD: Combinatory Homomorphic Automatic Differentiation*. en. arXiv:2103.15776 [cs]. (June 2022). Retrieved July 25, 2022 from http://arxiv.org/abs/2103.15776.

Andrea Walther. 2009. “Getting Started with ADOL-C.” In: *Combinatorial Scientific Computing* (Dagstuhl Seminar Proceedings). Ed. by Uwe Naumann, Olaf Schenk, Horst D. Simon, and Sivan Toledo. ISSN: 1862-4405 Issue: 09061. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany. Retrieved July 2, 2020 from http://drops.dagstuhl.de/opus/volltexte/2009/2084.

Fei Wang and Tiark Rompf. June 2018. “A Language and Compiler View on Differentiable Programming.” en, (June 2018). Retrieved Aug. 29, 2023 from https://openreview.net/forum?id=SJxJtYkPG.

Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. July 2019. “Demystifying differentiable programming: shift/reset the penultimate backpropagator.” *Proceedings of the ACM on Programming Languages*, 3, ICFP, (July 2019), 96:1–96:31. DOI: 10.1145/3341700.

Liang Wang, Jianxin Zhao, and Richard Mortier. 2022. *OCaml Scientific Computing: Functional Programming in Data Science and Artificial Intelligence*. en. Undergraduate Topics in Computer Science. Springer International Publishing, Cham. ISBN: 978-3-030-97644-6 978-3-030-97645-3. DOI: 10.1007/978-3-030-97645-3.

R. E. Wengert. Aug. 1964. “A simple automatic derivative evaluation program.” *Communications of the ACM*, 7, 8, (Aug. 1964), 463–464. DOI: 10.1145/355586.364791.

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Sept. 2014. “Effect handlers in scope.” In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell* (Haskell ’14). Association for Computing Machinery, Gothenburg, Sweden, (Sept. 2014), 1–12. ISBN: 978-1-4503-3041-1. DOI: 10.1145/2633357.2633358.

Zhixuan Yang, Marco Paviotti, Nicolas Wu, Birthe van den Berg, and Tom Schrijvers. 2022. “Structured Handling of Scoped Effects.” en. In: *Programming Languages and Systems*. Vol. 13240. Ed. by Ilya Sergey. Series Title: Lecture Notes in Computer Science. Springer International Publishing, Cham, 462–491. ISBN: 978-3-030-99335-1 978-3-030-99336-8. DOI: 10.1007/978-3-030-99336-8_17.

A GMM GRAPH FOR N = 10,000

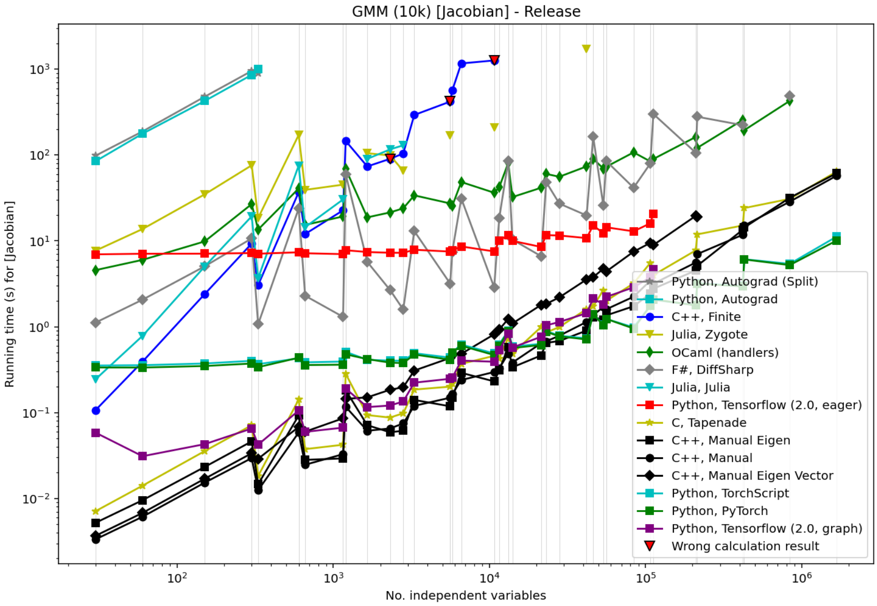


Fig. 3. GMM results, N = 10,000

B ADDITIONAL CODE FOR TENSOR VALUED OPERATIONS

B.1 Smooth Effect with Tensors

```
open Effect

type u_to_s = Const of float
type s_to_s = Negate | Log
type s's_to_s = Add | Subtract | Multiply | Divide

type u_to_t = Zeros of int array | Create of int array * float
type t_to_t
  = Squeeze of int array option
  | Reshape of int array
  | GetSlice of int list list
  | SliceLeft of int array
  | Transpose of int array option
  | Exp
  | Negate
  | PowerConst of float
  | SumReduce of int array option
  | LogSumExp of int option * bool option
  | Softmax of int option
type t't_to_t
  = Add
  | Subtract
  | Multiply
  | Divide
  | Einsum_ijk_mik_to_mij
  | Einsum_ijk_mij_to_mik
  | Einsum_mij_mik_to_ijk
  | SetSlice of int list list

type t_to_s = Get of int array | Sum
type s't_to_t = ScalarMultiply | SubtractScalar
type ta_to_t = Concatenate of int option | Stack of int option
type t_to_ta = Split of int option * int array

type arg = L | R

module type SMOOTH = sig
  type scalar
  type tensor
  type _ Effect.t +=
      Ap_u_to_s : u_to_s -> scalar Effect.t
    | Ap_s_to_s : s_to_s * scalar -> scalar Effect.t
    | Ap_s's_to_s : s's_to_s * scalar * scalar -> scalar Effect.t
    | Ap_u_to_t : u_to_t -> tensor Effect.t
    | Ap_t_to_t : t_to_t * tensor -> tensor Effect.t
    | Ap_t't_to_t : t't_to_t * tensor * tensor -> tensor Effect.t
    | Ap_t_to_s : t_to_s * tensor -> scalar Effect.t
    | Ap_s't_to_t : s't_to_t * scalar * tensor -> tensor Effect.t
    | Ap_ta_to_t : ta_to_t * tensor array -> tensor Effect.t
```

```
  | Ap_t_to_ta : t_to_ta * tensor -> tensor array Effect.t

val c : float -> scalar
val ( ~. ) : scalar -> scalar
val log : scalar -> scalar
val ( +. ) : scalar -> scalar -> scalar
val ( -. ) : scalar -> scalar -> scalar
val ( *. ) : scalar -> scalar -> scalar
val ( /. ) : scalar -> scalar -> scalar

(* Non-differentiable operations *)
val shape : tensor -> int array
val add_ : tensor -> tensor -> unit

(* Creating constant tensors *)
val zeros : int array -> tensor
val create : int array -> float -> tensor

(* Combining tensors *)
val concatenate : ?axis:int -> tensor array -> tensor
val stack : ?axis:int -> tensor array -> tensor

(* Splitting tensors *)
val split : ?axis:int -> int array -> tensor -> tensor array

(* Changing tensor shape *)
val transpose : ?axis:int array -> tensor -> tensor
val reshape : tensor -> int array -> tensor

(* Shrinking and slicing tensors *)
val squeeze : ?axis:int array -> tensor -> tensor
val get_slice : int list list -> tensor -> tensor
val slice_left : tensor -> int array -> tensor
val get : tensor -> int array -> scalar
val set_slice : int list list -> tensor -> tensor -> tensor

(* Einsum operations *)
val einsum_ijk_mik_to_mij : tensor -> tensor -> tensor
val einsum_ijk_mij_to_mik : tensor -> tensor -> tensor
val einsum_mij_mik_to_ijk : tensor -> tensor -> tensor

(* Pointwise tensor operations *)
val exp : tensor -> tensor
val pow_const : tensor -> float -> tensor
val ( ~- ) : tensor -> tensor
val ( + ) : tensor -> tensor -> tensor
val ( - ) : tensor -> tensor -> tensor
val ( * ) : tensor -> tensor -> tensor
val ( / ) : tensor -> tensor -> tensor

(* Reduction operations *)
val sum : tensor -> scalar
```

```
  val sum_reduce : ?axis:int array -> tensor -> tensor
  val log_sum_exp : ?axis:int -> ?keep_dims:bool -> tensor -> tensor
  val softmax : ?axis:int -> tensor -> tensor

  (* Scalar-tensor operations *)
  val scalar_mul : scalar -> tensor -> tensor
  val sub_scalar : tensor -> scalar -> tensor

  val op_u_to_s: u_to_s -> scalar
  val op_s_to_s: s_to_s -> scalar -> scalar
  val op_s's_to_s : s's_to_s -> scalar -> scalar -> scalar

  val op_u_to_t : u_to_t -> tensor
  val op_t_to_t : t_to_t -> tensor -> tensor
  val op_t't_to_t : t't_to_t -> tensor -> tensor -> tensor

  val op_t_to_s : t_to_s -> tensor -> scalar
  val op_s't_to_t : s't_to_t -> scalar -> tensor -> tensor
  val op_ta_to_t : ta_to_t -> tensor array -> tensor
  val op_t_to_ta : t_to_ta -> tensor -> tensor array

  val der_s_to_s : s_to_s -> scalar -> (scalar -> scalar)
  val der_s's_to_s : s's_to_s -> scalar -> scalar -> (scalar -> scalar * scalar)

  val der_t_to_t : t_to_t -> tensor -> (tensor -> tensor)
  val der_t't_to_t : t't_to_t -> tensor -> tensor -> (tensor -> tensor * tensor)

  val der_t_to_s : t_to_s -> tensor -> (scalar -> tensor)
  val der_s't_to_t : s't_to_t -> scalar -> tensor -> (tensor -> scalar * tensor)
  val der_ta_to_t : ta_to_t -> tensor array -> (tensor -> tensor array)
  val der_t_to_ta : t_to_ta -> tensor -> (tensor array -> tensor)
end

module type SMOOTH_NON_DIFF = sig
  type scalar
  type tensor

  val shape : tensor -> int array
  val add_ : tensor -> tensor -> unit
end

module Smooth (T : SMOOTH_NON_DIFF) : SMOOTH
  with type scalar = T.scalar
  with type tensor = T.tensor
= struct
  include T

  type scalar = T.scalar
  type tensor = T.tensor
  type _ Effect.t +=
      Ap_u_to_s : u_to_s -> scalar Effect.t
    | Ap_s_to_s : s_to_s * scalar -> scalar Effect.t
```

```
    | Ap_s's_to_s : s's_to_s * scalar * scalar -> scalar Effect.t
    | Ap_u_to_t : u_to_t -> tensor Effect.t
    | Ap_t_to_t : t_to_t * tensor -> tensor Effect.t
    | Ap_t't_to_t : t't_to_t * tensor * tensor -> tensor Effect.t
    | Ap_t_to_s : t_to_s * tensor -> scalar Effect.t
    | Ap_s't_to_t : s't_to_t * scalar * tensor -> tensor Effect.t
    | Ap_ta_to_t : ta_to_t * tensor array -> tensor Effect.t
    | Ap_t_to_ta : t_to_ta * tensor -> tensor array Effect.t

let c s = perform (Ap_u_to_s (Const s))
let log s = perform (Ap_s_to_s (Log, s))
let ( ~. ) s = perform (Ap_s_to_s (Negate, s))
let ( +. ) s1 s2 = perform (Ap_s's_to_s (Add, s1, s2))
let ( -. ) s1 s2 = perform (Ap_s's_to_s (Subtract, s1, s2))
let ( *. ) s1 s2 = perform (Ap_s's_to_s (Multiply, s1, s2))
let ( /. ) s1 s2 = perform (Ap_s's_to_s (Divide, s1, s2))

let zeros ia = perform (Ap_u_to_t (Zeros ia))
let create ia s = perform (Ap_u_to_t (Create (ia, s)))
let concatenate ?axis ta = perform (Ap_ta_to_t (Concatenate axis, ta))
let stack ?axis ta = perform (Ap_ta_to_t (Stack axis, ta))
let split ?axis ia t = perform (Ap_t_to_ta (Split (axis, ia), t))
let transpose ?axis t = perform (Ap_t_to_t (Transpose axis, t))
let reshape t d = perform (Ap_t_to_t (Reshape d, t))
let squeeze ?axis t =  perform (Ap_t_to_t (Squeeze axis, t))
let get_slice ill t = perform (Ap_t_to_t (GetSlice ill, t))
let slice_left t ia = perform (Ap_t_to_t (SliceLeft ia, t))
let get t ia = perform (Ap_t_to_s (Get ia, t))
let set_slice ill t1 t2 = perform (Ap_t't_to_t (SetSlice ill, t1, t2))
let einsum_ijk_mik_to_mij a x =
  perform (Ap_t't_to_t (Einsum_ijk_mik_to_mij, a, x))
let einsum_ijk_mij_to_mik a y =
  perform (Ap_t't_to_t (Einsum_ijk_mij_to_mik, a, y))
let einsum_mij_mik_to_ijk y x =
  perform (Ap_t't_to_t (Einsum_mij_mik_to_ijk, y, x))
let exp t = perform (Ap_t_to_t (Exp, t))
let ( ~- ) t = perform (Ap_t_to_t (Negate, t))
let pow_const t f = perform (Ap_t_to_t (PowerConst f,t))
let ( + ) t1 t2 = perform (Ap_t't_to_t (Add, t1, t2))
let ( - ) t1 t2 = perform (Ap_t't_to_t (Subtract, t1, t2))
let ( * ) t1 t2 = perform (Ap_t't_to_t (Multiply, t1, t2))
let ( / ) t1 t2 = perform (Ap_t't_to_t (Divide, t1, t2))
let sum t = perform (Ap_t_to_s (Sum, t))
let sum_reduce ?axis t = perform (Ap_t_to_t (SumReduce axis, t))
let log_sum_exp ?axis ?keep_dims t =
  perform (Ap_t_to_t (LogSumExp (axis, keep_dims), t))
let softmax ?axis t = perform (Ap_t_to_t (Softmax axis, t))
let scalar_mul s t = perform (Ap_s't_to_t (ScalarMultiply, s, t))
let sub_scalar t s = perform (Ap_s't_to_t (SubtractScalar, s, t))

(* Simple expand operation. ia contains which axes to expand. *)
let _expand t shp ia =
```

```
    let res = ref t in
    for j = 0 to Stdlib.(Array.length ia - 1) do
      res := concatenate ~axis:(ia.(j)) (Array.make shp.(ia.(j)) !res)
    done;
    !res

  (* Inverse of a permutation *)
  let _inv_perm p =
    let l = Array.length p in
    let q = Array.make l 0 in
    for i = 0 to Stdlib.(l - 1) do
      q.(p.(i)) <- i;
    done;
    q

  let op_u_to_s (o : u_to_s) = match o with
    | Const x -> c x
  let op_s_to_s (o : s_to_s) s = match o with
    | Negate -> ~. s
    | Log -> log s
  let op_s's_to_s (o : s's_to_s) s1 s2 = match o with
    | Add -> s1 +. s2
    | Subtract -> s1 -. s2
    | Multiply -> s1 *. s2
    | Divide -> s1 /. s2

  let op_u_to_t (o : u_to_t) = match o with
    | Zeros ia -> zeros ia
    | Create (ia, f) -> create ia f
  let op_t_to_t (o : t_to_t) t = match o with
    | Squeeze iao -> squeeze ?axis:iao t
    | Reshape d -> reshape t d
    | GetSlice ill -> get_slice ill t
    | SliceLeft ia -> slice_left t ia
    | Transpose iao -> transpose ?axis:iao t
    | Exp -> exp t
    | Negate -> ~- t
    | PowerConst f -> pow_const t f
    | SumReduce iao -> sum_reduce ?axis:iao t
    | LogSumExp (io, bo) -> log_sum_exp ?axis:io ?keep_dims:bo t
    | Softmax io -> softmax ?axis:io t
  let op_t't_to_t (o : t't_to_t) t1 t2 = match o with
    | Add -> t1 + t2
    | Subtract -> t1 - t2
    | Multiply -> t1 * t2
    | Divide -> t1 / t2
    | Einsum_ijk_mik_to_mij -> einsum_ijk_mik_to_mij t1 t2
    | Einsum_ijk_mij_to_mik -> einsum_ijk_mij_to_mik t1 t2
    | Einsum_mij_mik_to_ijk -> einsum_mij_mik_to_ijk t1 t2
    | SetSlice ill -> set_slice ill t1 t2

  let op_t_to_s (o : t_to_s) t = match o with
```

```
  | Get ia -> get t ia
  | Sum -> sum t
let op_s't_to_t (o : s't_to_t) s t = match o with
  | ScalarMultiply -> scalar_mul s t
  | SubtractScalar -> sub_scalar t s
let op_ta_to_t (o : ta_to_t) ta = match o with
  | Concatenate io ->  concatenate ?axis:io ta
  | Stack io -> stack ?axis:io ta
let op_t_to_ta (o : t_to_ta) t = match o with
  | Split (io, ia) -> split ?axis:io ia t

let der_s_to_s (o : s_to_s) s = match o with
  | Negate -> fun sd -> ~. sd
  | Log -> fun sd -> sd /. s
let der_s's_to_s (o : s's_to_s) s1 s2 = match o with
  | Add -> fun sd -> (sd, sd)
  | Subtract -> fun sd -> (sd, ~. sd)
  | Multiply -> fun sd -> (s2 *. sd, s1 *. sd)
  | Divide -> fun sd -> (sd /. s2, (sd *. (~. s1)) /. (s2 *. s2))

let der_t_to_t (o : t_to_t) t = match o with
  | Squeeze _ -> fun td -> reshape td (shape t)
  | Reshape _ -> fun td -> reshape td (shape t)
  | GetSlice ill -> fun td -> set_slice ill (zeros (shape t)) td
  | SliceLeft ia -> fun td ->
    let ill = Array.to_list (Array.map (fun i -> [i]) ia) in
    let shp = Array.(append (make (length ia) 1) (shape td)) in
    let tdr = reshape td shp in
    set_slice ill (zeros (shape t)) tdr
  | Transpose iao ->
    let ia = match iao with
      | None ->
        let d = Array.length (shape t) in
        Array.init d Stdlib.(fun i -> d - i - 1)
      | Some ia -> ia
    in
    fun td -> transpose ~axis:(_inv_perm ia) td
  | Exp -> fun td -> exp t * td
  | Negate -> fun td -> ~- td
  | PowerConst f -> fun td ->
    scalar_mul (c f) (td * pow_const t Stdlib.(f -. 1.0))
  | SumReduce iao ->
    let ia = (match iao with
      | None -> Array.init (Array.length (shape t)) (fun i -> i)
      | Some ia -> ia
    ) in
    fun td -> _expand td (shape t) ia
  | LogSumExp (io, bo) -> (
    let (i, b) = match (io, bo) with
      | (None, None) -> (0, true)
      | (Some i, None) -> (i, true)
      | (None, Some b) -> (0, b)
```

```
      | (Some i, Some b) -> (i, b)
    in
    if b
      then fun td -> td * softmax ~axis:i t
      else fun td ->
        let shp = shape t in
        shp.(i) <- 1;
        (reshape td shp) * (softmax ~axis:i t)
    )
  | Softmax _io -> raise (Invalid_argument "Softmax not implemented")
let der_t't_to_t (o : t't_to_t) t1 t2 = match o with
  | Add -> fun td -> (td, td)
  | Subtract -> fun td -> (td, ~- td)
  | Multiply -> fun td -> (t2 * td, t1 * td)
  | Divide -> fun td -> (td / t2, (td * (~- t1)) / (t2 * t2))
  | Einsum_ijk_mik_to_mij -> fun td ->
    (einsum_mij_mik_to_ijk td t2, einsum_ijk_mij_to_mik t1 td)
  | Einsum_ijk_mij_to_mik -> fun td ->
    (einsum_ijk_mik_to_mij t1 td, einsum_mij_mik_to_ijk t2 td)
  | Einsum_mij_mik_to_ijk -> fun td ->
    (einsum_ijk_mik_to_mij td t2, einsum_ijk_mij_to_mik td t1)
  | SetSlice ill -> fun td ->
    (set_slice ill td (zeros (shape t2)), get_slice ill td)

let der_t_to_s (o : t_to_s) t = match o with
  | Get ia ->
    let ill = Array.to_list (Array.map (fun i -> [i]) ia) in
    (fun sd ->
      let ones = Array.(make (length (shape t)) 1) in
      set_slice ill (zeros (shape t)) (scalar_mul sd (create ones 1.0))
    )
  | Sum -> fun sd -> scalar_mul sd (create (shape t) 1.0)
let der_s't_to_t (o : s't_to_t) s t = match o with
  | ScalarMultiply -> fun td -> (sum (t * td), scalar_mul s td)
  | SubtractScalar -> fun td -> (~. (sum td), td)
let der_ta_to_t (o : ta_to_t) ta = match o with
  | Concatenate io ->
    let i = (match io with
      | None -> 0
      | Some i -> i
    ) in
    fun td -> split ~axis:i (Array.map (fun x -> (shape x).(i)) ta) td
  | Stack io ->
    let i = (match io with
      | None -> 0
      | Some i -> i
    ) in
    (fun td ->
      let shp = shape td in
      let ndim = Array.length shp in
      let axis = Owl_utils.adjust_index i ndim in
      let inp_shp = shape ta.(0) in
```

```
        split ~axis:i (Array.make shp.(axis) 1) td
          |> Array.map (fun x -> reshape x inp_shp)
      )
  let der_t_to_ta (o : t_to_ta) _ = match o with
    | Split (io, _) ->
      let i = (match io with
        | None -> 0
        | Some i -> i
      ) in
      fun tda -> concatenate ~axis:i tda
end
```

B.2 Reverse Mode with Tensors

```
open Effect.Deep
open Modules_effect_handlers_smooth_tensor

type 't prop = {v : 't; mutable dv : 't}

module Reverse_Non_Diff (T : SMOOTH_NON_DIFF) : SMOOTH_NON_DIFF
  with type scalar = T.scalar prop
  with type tensor = T.tensor prop
= struct
  type scalar = T.scalar prop
  type tensor = T.tensor prop

  let shape t = T.shape t.v
  let add_ x dx = T.add_ x.v dx.v; T.add_ x.dv dx.dv
end

module Reverse (T : SMOOTH) = struct
  include Smooth (Reverse_Non_Diff (T : SMOOTH_NON_DIFF))

  let reverse = {
    retc = (fun x -> x);
    exnc = raise;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | Ap_u_to_s o -> Some (fun (k : (a, _) continuation) -> let open T in
          continue k {v = op_u_to_s o; dv = c 0.0}
        )
      | Ap_s_to_s (o, s) -> Some (fun k -> let open T in
          let r = {v = op_s_to_s o s.v; dv = c 0.0} in
          continue k r;
          s.dv <- s.dv +. (der_s_to_s o s.v r.dv)
        )
      | Ap_s's_to_s (o, s1, s2) -> Some (fun k -> let open T in
          let r = {v = op_s's_to_s o s1.v s2.v; dv = c 0.0} in
          continue k r;
          let (dv1, dv2) = der_s's_to_s o s1.v s2.v r.dv in
          s1.dv <- s1.dv +. dv1;
          s2.dv <- s2.dv +. dv2
        )
```

```
    | Ap_u_to_t o -> Some (fun k -> let open T in
      let v = op_u_to_t o in
      continue k {v = v; dv = create (shape v) 0.0}
     )
    | Ap_t_to_t (o, t) -> Some (fun k -> let open T in
      let v = op_t_to_t o t.v in
      let r = {v = v; dv = create (shape v) 0.0} in
      continue k r;
      let dv = der_t_to_t o t.v r.dv in
      if shape t.dv = shape dv then add_ t.dv dv else t.dv <- t.dv + dv
     )
    | Ap_t't_to_t (o, t1, t2) -> Some (fun k -> let open T in
      let v = op_t't_to_t o t1.v t2.v in
      let r = {v = v; dv = create (shape v) 0.0} in
      continue k r;
      let (dv1, dv2) = der_t't_to_t o t1.v t2.v r.dv in
      if shape t1.dv = shape dv1
       then add_ t1.dv dv1 else t1.dv <- t1.dv + dv1;
      if shape t2.dv = shape dv2
       then add_ t2.dv dv2 else t2.dv <- t2.dv + dv2
     )
    | Ap_t_to_s (o, t) -> Some (fun k -> let open T in
      let r = {v = op_t_to_s o t.v; dv = c 0.0} in
      continue k r;
      let dv = der_t_to_s o t.v r.dv in
      if shape t.dv = shape dv then add_ t.dv dv else t.dv <- t.dv + dv
     )
    | Ap_s't_to_t (o, s, t) -> Some (fun k -> let open T in
      let v = op_s't_to_t o s.v t.v in
      let r = {v = v; dv = create (shape v) 0.0} in
      continue k r;
      let (ds, dt) = der_s't_to_t o s.v t.v r.dv in
      s.dv <- s.dv +. ds;
      if shape t.dv = shape dt then add_ t.dv dt else t.dv <- t.dv + dt
     )
    | Ap_ta_to_t (o, ta) -> Some (fun k -> let open T in
      let tva = Array.(map (fun t -> t.v) ta) in
      let v = op_ta_to_t o tva in
      let r = {v = v; dv = create (shape v) 0.0} in
      continue k r;
      let rdva = der_ta_to_t o tva r.dv in
      ignore Array.(map2 (fun t rdv -> (
       if shape t.dv = shape rdv then add_ t.dv rdv else t.dv <- t.dv + rdv
      )) ta rdva)
     )
    | Ap_t_to_ta (o, t) -> Some (fun k -> let open T in
      let va = op_t_to_ta o t.v in
      let ra =
       Array.(map (fun v -> {v = v; dv = create (shape v) 0.0}) va)
      in
      continue k ra;
      let rdva = Array.(map (fun r -> r.dv) ra) in
```

```
          let dv = der_t_to_ta o t.v rdva in
          if shape t.dv = shape dv then add_ t.dv dv else t.dv <- t.dv + dv
        )
      | _ -> None
    )
  }

  let grad f ta =
    let ra = Array.map (fun t -> {v = t; dv = T.(create (shape t) 0.0)}) ta in
    match_with (fun ta -> (f ta).dv <- T.c 1.0) ra reverse;
    Array.map (fun r -> r.dv) ra
end
```