# Automatic Differentiation via Effects and Handlers in OCaml

Jesse Sigal (University of Edinburgh)

ML 2024, September 6, 2024

# Overview

# Overview

Automatic differentiation

Effects and handlers

The `Smooth` effect

Evaluate handler

Reverse mode handler

Benchmarks

Conclusion

# Automatic differentiation: what and why

- Derivative based optimization.
- Automatic differentiation (AD) is a family of algorithms which automatically computes derivatives.
- AD is only a small constant multiple slower than the original program.
- Wide variety of implementations and methods.
- Available methods depend on the language's features.
- *Idea:* effects and handlers provide a practical basis for AD.

# Automatic differentiation: chain rule two different ways

$g(f(x))$ at $x = a$

# Automatic differentiation: chain rule two different ways

$g(f(x))$ at $x = a$

```
let x = a in
let y = f x in
g y
```

# Automatic differentiation: chain rule two different ways

$g(f(x))$ at $x = a$

```
let x = a in
let y = f x in
g y
```

**Chain rule, two functions**

$$\frac{d}{dx} g\left(f(x)\right) = g'\left(f(x)\right) \cdot f'(x)$$

# Automatic differentiation: chain rule two different ways

$g(f(x))$ at $x = a$

```
let x = a in
let y = f x in
g y
```

**Chain rule, two functions**

$$\frac{d}{dx} g\left(f(x)\right) = g'\left(f(x)\right) \cdot f'(x)$$

**Chain rule, three functions**

$$\frac{d}{dx} h\left(g\left(f(x)\right)\right) = h'\left(g\left(f(x)\right)\right) \cdot g'\left(f(x)\right) \cdot f'(x)$$

# Automatic differentiation: chain rule two different ways

$g(f(x))$ at $x = a$

```
let x = a in
let y = f x in
g y
```

$$\frac{d}{dx} g(f(x)) = g'\left(f(x)\right) \cdot \left(f'(x) \cdot 1\right)$$

# Automatic differentiation: chain rule two different ways

$g(f(x))$ at $x = a$

```
let x = a in
let y = f x in
g y
```

$$\frac{d}{dx} g(f(x)) = g'\left(f(x)\right) \cdot \left(f'(x) \cdot 1\right)$$

```
let (x, dx) = (a, 1) in
let (y, dy) = (f x, (f' x) * dx) in
(g y, (g' y) * dy)
```

# Automatic differentiation: chain rule two different ways

$g(f(x))$ at $x = a$

```
let x = a in
let y = f x in
g y
```

$$\frac{d}{dx} g(f(x)) = g'\left(f(x)\right) \cdot \left(f'(x) \cdot 1\right)$$

```
let (x, dx) = (a, 1) in
let (y, dy) = (f x, (f' x) * dx) in
(g y, (g' y) * dy)
```

$$\frac{d}{dx} g(f(x)) = \left(1 \cdot g'\left(f(x)\right)\right) \cdot f'(x)$$

# Automatic differentiation: chain rule two different ways

$g(f(x))$ at $x = a$

```
let x = a in
let y = f x in
g y
```

$$\frac{d}{dx} g(f(x)) = g'\left(f(x)\right) \cdot \left(f'(x) \cdot 1\right)$$

```
let (x, dx) = (a, 1) in
let (y, dy) = (f x, (f' x) * dx) in
(g y, (g' y) * dy)
```

$$\frac{d}{dx} g(f(x)) = \left(1 \cdot g'\left(f(x)\right)\right) \cdot f'(x)$$

```
let x = a in
let bz = 1 in
let (z, bx) =
  let y = f x in
  let (z, by) =
    let z = g y in
    (z, bz * (g' y))
  in (z, by * (f' x))
in (z, bx)
```

# Automatic differentiation: stateful reverse mode AD

```
let x = a in
let bz = 1 in
let (z, bx) =
  let y = f x in
  let (z, by) =
    let z = g y in
    (z, bz * (g' y))
  in (z, by * (f' x))
in (z, bx)
```

$\mapsto$

```
let (x, bx) = (a, ref 0) in
let (y, by) = (f x, ref 0) in
let (z, bz) = (g y, ref 1) in
by := !by + (!bz * (g' y));
bx := !bx + (!by * (f' x));
(z, !bx)
```

# Automatic differentiation: stateful reverse mode AD

```
let x = a in
let bz = 1 in
let (z, bx) =
  let y = f x in
  let (z, by) =
    let z = g y in
    (z, bz * (g' y))
  in (z, by * (f' x))
in (z, bx)
```

$\mapsto$

```
let (x, bx) = (a, ref 0) in
let (y, by) = (f x, ref 0) in
let (z, bz) = (g y, ref 1) in
by := !by + (!bz * (g' y));
bx := !bx + (!by * (f' x));
(z, !bx)
```

How do we move beyond straight-line programs?

# Automatic differentiation: stateful reverse mode AD

```
let x = a in
let bz = 1 in
let (z, bx) =
  let y = f x in
  let (z, by) =
    let z = g y in
    (z, bz * (g' y))
  in (z, by * (f' x))
in (z, bx)
```

$\mapsto$

```
let (x, bx) = (a, ref 0) in
let (y, by) = (f x, ref 0) in
let (z, bz) = (g y, ref 1) in
by := !by + (!bz * (g' y));
bx := !bx + (!by * (f' x));
(z, !bx)
```

How do we move beyond straight-line programs?

*Effects and handlers!*

# Overview

# Effects and handlers: what and why

- Structured user-defined side-effects.
- Like catchable exceptions, but allows continuing from thrown location.
- Provide abstraction, composition, and reuse.
- Allows for complicated control flow.
- In OCaml as of 5.0!
- Gives a straight-line view of operations.

# Effects and handlers: bubbling up

- We want to “bubble up” the arithmetic operations to get a straight-line program.

# Effects and handlers: bubbling up

- We want to "bubble up" the arithmetic operations to get a straight-line program.
- Say we have:

  let x = $\mathcal{C}$[sin 0.5] in $\mathcal{D}$[x]

# Effects and handlers: bubbling up

- We want to "bubble up" the arithmetic operations to get a straight-line program.
- Say we have:

  let x = $\mathcal{C}$[sin 0.5] in $\mathcal{D}$[x]

- Bubbling up, we get:

  let y = sin 0.5 in
  let x = $\mathcal{C}$[y] in $\mathcal{D}$[x]

# Effects and handlers: bubbling up

- We want to "bubble up" the arithmetic operations to get a straight-line program.
- Say we have:

  let x = $\mathcal{C}$[sin 0.5] in $\mathcal{D}$[x]

- Bubbling up, we get:

  let y = sin 0.5 in
  let x = $\mathcal{C}$[y] in $\mathcal{D}$[x]

- We can also make the second line a first-class function

  let y = sin 0.5 in
  let k = (fun v -> let x = $\mathcal{C}$[v] in $\mathcal{D}$[x]) in
  k y

# Effects and handlers: bubbling up

- We want to "bubble up" the arithmetic operations to get a straight-line program.
- Say we have:

  `let x =` $\mathcal{C}$`[sin 0.5] in` $\mathcal{D}$`[x]`

- Bubbling up, we get:

  `let y = sin 0.5 in`
  `let x =` $\mathcal{C}$`[y] in` $\mathcal{D}$`[x]`

- We can also make the second line a first-class function

  `let y = sin 0.5 in`
  `let k = (fun v -> let x =` $\mathcal{C}$`[v] in` $\mathcal{D}$`[x]) in`
  `k y`

- Effects and handlers do exactly this!

# Overview

# Smooth effect: data types

```
type nullary = Const of float
type unary = Negate | Sin | Cos | Exp
type binary = Plus | Subtract | Times | Divide
type arg = L | R
```

# Smooth effect: signature

```
open Effect

module type SMOOTH = sig
  type t
  type _ Effect.t += Ap0 : nullary -> t Effect.t
                  | Ap1 : unary * t -> t Effect.t
                  | Ap2 : binary * t * t -> t Effect.t
  val c : float -> t
  val ( ~. ) : t -> t
  ...
  val ap0 : nullary -> t
  val ap1 : unary -> t -> t
  val ap2 : binary -> t -> t -> t
  val der1 : unary -> t -> t
  val der2 : binary -> arg -> t -> t -> t
end
```

# Smooth effect: base module

```
module Smooth (T : sig type t end) : SMOOTH with type t = T.t = struct
  type t = T.t
  type _ Effect.t += Ap0 : nullary -> t Effect.t
                   | Ap1 : unary * t -> t Effect.t
                   | Ap2 : binary * t * t -> t Effect.t
  let c x = perform (Ap0 (Const x))
  let ( ~. ) a = perform (Ap1 (Negate, a))
  ...
  let ap0 n =  perform (Ap0 n)
  let ap1 u x = perform (Ap1 (u, x))
  let ap2 b x y = perform (Ap2 (b, x, y))

  let der1 u x = match u with (* ∂/∂x(u(x)) *)
    | Negate -> ~. (c 1.0)    (* ∂/∂x(−x) = −1 *)
    | Sin    -> cos_ x        (* ∂/∂x(sin(x)) = cos(x) *)
    | Cos    -> ~. (sin_ x)   (* ∂/∂x(cos(x)) = −sin(x) *)
    | Exp    -> exp_ x        (* ∂/∂x(e^x) = e^x *)
  ...
```

# Smooth effect: base module

```
  ...
  let der2 b arg x y = match b with (* ∂/∂x_arg (b(x_L, x_R)), for x_L = x, x_R = y *)
    (* ∂/∂x(x + y) = 1,  ∂/∂y(x + y) = 1 *)
    | Plus -> (match arg with L -> c 1.0 | R -> c 1.0)
    (* ∂/∂x(x − y) = 1,  ∂/∂y(x − y) = −1 *)
    | Subtract -> (match arg with L -> c 1.0 | R -> c (-1.0))
    (* ∂/∂x(x · y) = y,  ∂/∂y(x · y) = x *)
    | Times -> (match arg with L -> y | R -> x)
    (* ∂/∂x(x/y) = 1/y,  ∂/∂y(x/y) = −x/y² *)
    | Divide ->
      (match arg with L -> (c 1.0) /. y | R -> (~. x) /. (y *. y))
end
```

# Overview

# Evaluate: handler

```
open Effect.Deep
open Float
open Smooth

module Evaluate = struct
  include Smooth (struct type t = float end)

```

# Evaluate: handler

```
open Effect.Deep
open Float
open Smooth

module Evaluate = struct
  include Smooth (struct type t = float end)

  let (evaluate : ('a, 'a) handler) = {
    retc = (fun x -> x);
    exnc = raise;
    effc = (fun (type x) (eff : x Effect.t) ->
      match eff with
```

# Evaluate: handler

```
open Effect.Deep
open Float
open Smooth

module Evaluate = struct
  include Smooth (struct type t = float end)

  let (evaluate : ('a, 'a) handler) = {
    retc = (fun x -> x);
    exnc = raise;
    effc = (fun (type x) (eff : x Effect.t) ->
      match eff with
      | Ap0 n -> Some (fun (k : (x, 'a) continuation) ->
          match n with
          | Const x -> continue (k :  (float, 'a) continuation) x
        )
  ...
```

# Evaluate: handler

```
  ...
      | Ap1 (u, x) -> Some (fun k ->
        match u with
        | Negate -> continue k (neg x)
        | Sin    -> continue k (sin x)
        | Cos    -> continue k (cos x)
        | Exp    -> continue k (exp x)
      )
      | Ap2 (b, x, y) -> Some (fun k ->
        match b with
        | Plus     -> continue k (add x y)
        | Subtract -> continue k (sub x y)
        | Times    -> continue k (mul x y)
        | Divide   -> continue k (div x y)
      )
      | _ -> None
    )
  }
end
```

# Evaluate: example

```
open Effect.Deep
open Evaluate

let _ =
  let open Evaluate in
  let sqr x = x *. x in
  let res = (match_with : ('c -> 'a) -> 'c -> ('a, 'b) handler -> 'b)
    (fun (twice, x) -> if twice then sqr (sqr x) else sqr x)
    (true, 5.0)
    evaluate
  in
  Printf.printf "%f\n" res (* Prints "625.000000"=5^4 *)
```

# Overview

# Reverse mode

$$\frac{d}{dx} g(f(x)) = \left(1 \cdot g'\left(f(x)\right)\right) \cdot f'(x)$$

```
let (x, bx) = (a, ref 0) in
let (y, by) = (f x, ref 0) in
let (z, bz) = (g y, ref 1) in
by := !by + ((g' y) * !bz);
bx := !bx + ((f' x) * !by);
(z, !bx)
```

# Reverse mode: numeric type

```
open Effect.Deep
open Smooth

type 't mpaired = {v : 't; mutable bv : 't}

```

# Reverse mode: numeric type

```
open Effect.Deep
open Smooth

type 't mpaired = {v : 't; mutable bv : 't}

module Reverse (T : SMOOTH) = struct
  include Smooth (struct type t = T.t mpaired end)

```

# Reverse mode: numeric type

```
open Effect.Deep
open Smooth

type 't mpaired = {v : 't; mutable bv : 't}

module Reverse (T : SMOOTH) = struct
  include Smooth (struct type t = T.t mpaired end)

  let (reverse : (unit, unit) handler) = {
    retc = (fun x -> x);
    exnc = raise;
    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; bv = c 0.0}
          )
        | Ap1 (u, x) -> Some (fun k -> let open T in
            let r = {v = ap1 u x.v; bv = c 0.0} in
            continue k r;
            x.bv <- x.bv +. (der1 u x.v *. r.bv)
          )
        | Ap2 (b, x, y) -> Some (fun k -> let open T in
            let r = {v = ap2 b x.v y.v; bv = c 0.0} in
            continue k r;
            x.bv <- x.bv +. (der2 b L x.v y.v *. r.bv);
            y.bv <- y.bv +. (der2 b R x.v y.v *. r.bv)
          )
        | _ -> None
      )
    }
```

# Reverse mode: derivative function

```
  ...
  (* grad f x = ∂f(z)/∂z (x) *)
  let grad (f : T.t mpaired -> T.t mpaired) (x : T.t) =
    let r = {v = x; bv = T.c 0.0} in
    match_with (fun x -> (f x).bv <- T.c 1.0) r reverse;
    r.bv
end
```

# Reverse mode: example

```
let _ =
  let module E = Evaluate in
  let module R = Reverse(E) in
  let sqr x = R.(x *. x) in
  let res = match_with
    (fun (twice, y) ->
      R.grad (fun x -> if twice then sqr (sqr x) else sqr x) y
    )
    (true, 5.0)
    E.evaluate
  in
  Printf.printf "%f\n" res (* Prints "500.000000" = 4 · 5^3 = ∂(x^4)/∂x (5)) *)
```

# Reverse mode: nested example

```
let _ =
  let module E = Evaluate in
  let module R = Reverse(E) in
  let module RR = Reverse(R) in
  let sqr x = RR.(x *. x) in
  let res = match_with (fun (twice, z) ->
    R.grad (fun y ->
      RR.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 · 5^2 = ∂^2(x^4)/∂x^2(5) *)
```

# Reverse mode: original implementation

- We are not the first to implement reverse mode AD with handlers, see [Sivaramakrishnan, 2018].
- [Sivaramakrishnan, 2018] was inspired by [Wang et al., 2019] who used delimited continuations.
- We are the first to design a larger system and add tensor valued operations, as well as benchmark.

# Overview

# What to expect

- [Griewank and Walther, 2008, Sec. 4.4] show that for a composite measure of "work", reverse mode is $O(1)$ w.r.t. the original program.
- Work includes
  - memory fetches and stores,
  - additions and subtractions,
  - multiplications, and
  - non-linear operations.
- With reasonable assumptions, they prove reverse mode should be $3\times$ to $4\times$ slower.

# Microbenchmark: code

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

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

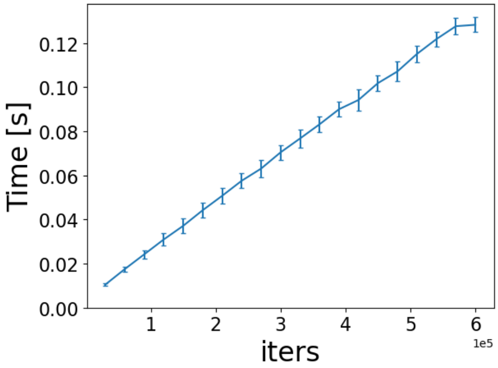
# Microbenchmark: code

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

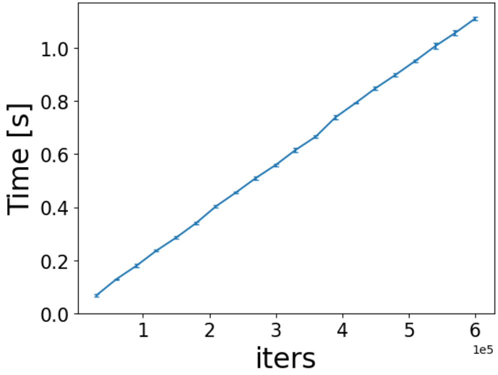
$$a_0 = 1, \qquad a_n = -(x-1) \cdot a_{n-1}$$

```
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 (* $\sum_{n=0}^{iters} a_n$ *)
end
```




(a) Evaluation mode




(b) Reverse mode

# Microbenchmark: results

Reverse mode is about 8.3× slower

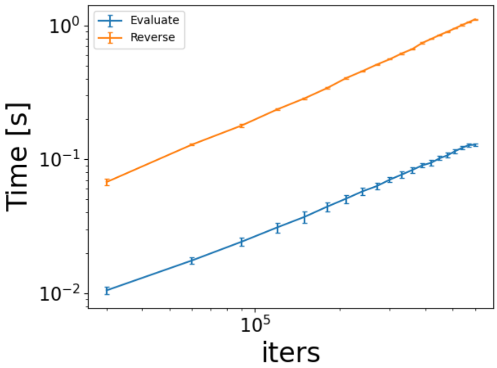



Figure: Reverse and evaluation modes, log-log scale

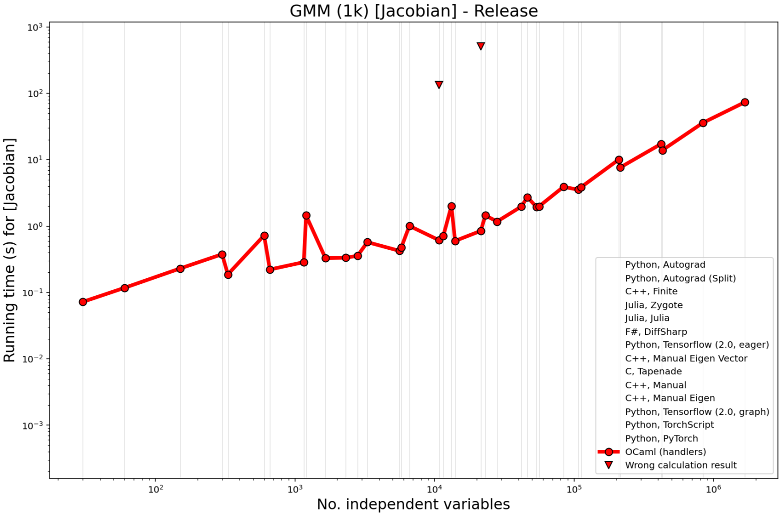
# Macrobenchmark

- Benchmark suite of [Srajer et al., 2018]
  - reproducible: containers
  - extensible: test harnesses, modular
  - realistic: real ML and computer vision functions
- *Problem:* one effect call per real-valued operation will be inefficient
- *Solution:* tensor/matrix/ND-array operations
  - Owl scientific computing library of [Wang et al., 2022]
  - Extend `Smooth` to 35 operations
  - Extend `Reverse` to handle new operation types
- We implement the objective function for Gaussian mixture models
- 3 different parameters $N$, $K$, and $D$
- $K \cdot D$ is the total number of input variables, giving our $x$-axis

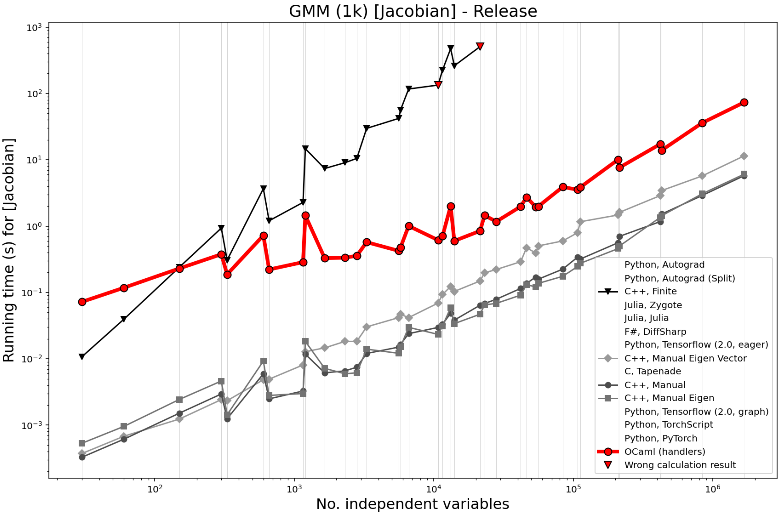
# Macrobenchmark: other systems

| Language | Tool | Approach |
|---|---|---|
| C++ | - | Manual (by hand) |
| C++ | - | Finite differences |
| C | Tapenade | Static |
| Python | Autograd | Dynamic |
| Python | TensorFlow 2.0 (eager) | Dynamic |
| Python | TensorFlow 2.0 (graph) | Static |
| Python | PyTorch | Dynamic |
| Python | TorchScript | Static |
| Julia | ForwardDiff.jl | Dynamic |
| Julia | Zygote | Static |
| F# | DiffSharp | Dynamic |
| **OCaml** | **This work** | **Dynamic** |

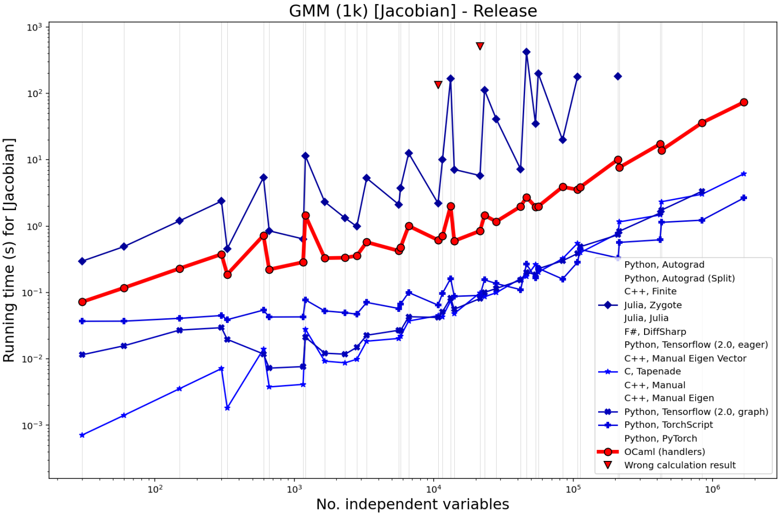
# Macrobenchmark: results (1k)



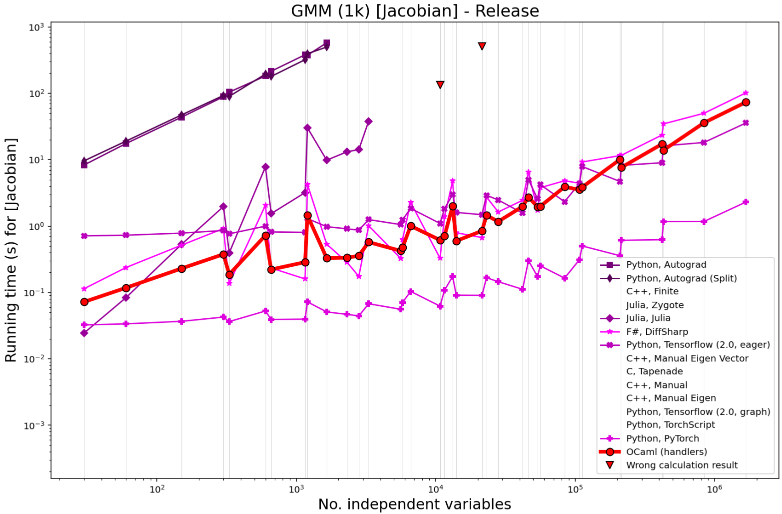
# Macrobenchmark: results (1k, manual)



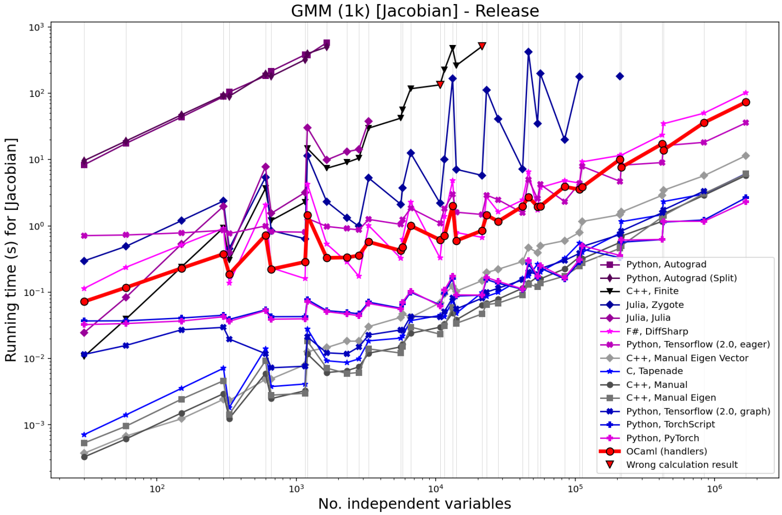
# Macrobenchmark: results (1k, static)



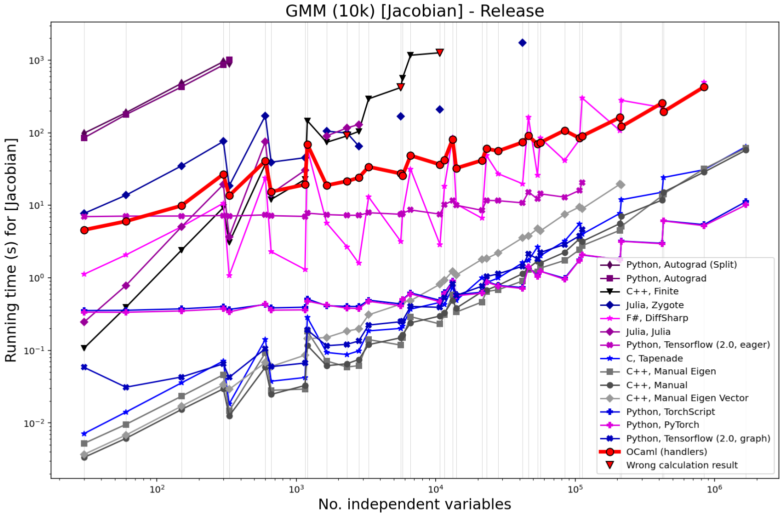
# Macrobenchmark: results (1k, dynamic)



# Macrobenchmark: results (1k, all)



# Macrobenchmark: results (10k)



# Overview

# More to say on AD and effect handlers

- Many different modes
  - Checkpointed reverse mode for time-space tradeoff
  - Higher-order functions
  - Hessians
- Different languages, some with effect type systems
  - Koka
  - Frank
  - Eff
- Mathematical correctness
  - Denotational semantics
  - For forward mode and (simpler) reverse mode

See my thesis:

# Conclusion

- AD (and reverse mode specifically) requires complex control flow.
- Effect handlers enable a simple implementation which follows the math.
- With little effort and moving to tensor valued operations, we are competitive among similar tools.
- **Future work:**
  - Use Torch bindings in OCaml.
  - Correctness of reverse mode.
  - Custom functions (higher-order effects?).

Preprint:

# Bibliography I


Griewank, A. and Walther, A. (2008).
*Evaluating Derivatives*.
Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics.

Sivaramakrishnan, K. C. (2018).
Reverse-mode Algorithmic differentiation using effect handlers.

Srajer, F., Kukelova, Z., and Fitzgibbon, A. (2018).
A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning.
*Optimization Methods and Software*, 33(4-6):889–906.
Publisher: Taylor & Francis _eprint:
https://doi.org/10.1080/10556788.2018.1435651.

# Bibliography II


Wang, F., Zheng, D., Decker, J., Wu, X., Essertel, G. M., and Rompf, T. (2019).
Demystifying differentiable programming: shift/reset the penultimate backpropagator.
*Proceedings of the ACM on Programming Languages*, 3(ICFP):96:1–96:31.

Wang, L., Zhao, J., and Mortier, R. (2022).
*OCaml Scientific Computing: Functional Programming in Data Science and Artificial Intelligence*.
Undergraduate Topics in Computer Science. Springer International Publishing, Cham.