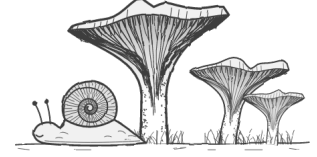


Dual Numbers for Automatic Differentiation

Mikkel Paltorp (mpasc@dtu.dk), 25 September 2026

Department of Applied Mathematics and Computer Science, Technical University of Denmark



1. Scalar derivatives

Dual numbers are a key part of forward-mode differentiation. However, I find the introduction of them unintuitive. Usually, as done in [1], they are introduced such that

$$f(x + \varepsilon) = f(x) + f'(x)\varepsilon, \quad (1)$$

where $\varepsilon^2 = 0$. This is not incorrect. However, I think it is more natural to think of dual numbers, at least in the context of automatic differentiation, as a number type that simply tracks both $f(x)$ and $f'(x)$ at the same time by introducing an ε . This is sort of similar to how a complex number can be seen to track an a and b at the same time by introducing an i . However, for complex numbers the aim is to be able to solve $x^2 = -1$. For dual numbers the intent, as noted before, is to track $f(x)$ and $f'(x)$ at the same time. We therefore define how a function acts on a dual number by the composition rule

$$\begin{aligned} f(g(x) + g'(x)\varepsilon) &:= f(g(x)) + (f(g(x)))'\varepsilon \\ &= f(g(x)) + f'(g(x))g'(x)\varepsilon, \end{aligned} \quad (2)$$

of which Equation 1 is the special case $g(x) + g'(x)\varepsilon = x + \varepsilon$. In the same way we define multiplication and division of dual numbers by the product and quotient rules

$$\begin{aligned} (f(x) + f'(x)\varepsilon)(g(x) + g'(x)\varepsilon) &:= f(x)g(x) + (f(x)g(x))'\varepsilon \\ &= f(x)g(x) + (f'(x)g(x) + f(x)g'(x))\varepsilon, \\ \frac{f(x) + f'(x)\varepsilon}{g(x) + g'(x)\varepsilon} &:= \frac{f(x)}{g(x)} + \left(\frac{f(x)}{g(x)}\right)'\varepsilon \\ &= \frac{f(x)}{g(x)} + \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}\varepsilon, \end{aligned} \quad (3)$$

and addition and subtraction by the sum and difference rules

$$\begin{aligned} (f(x) + f'(x)\varepsilon) + (g(x) + g'(x)\varepsilon) &:= (f(x) + g(x)) + (f'(x) + g'(x))\varepsilon \\ &= (f(x) + g(x)) + (f'(x) + g'(x))\varepsilon, \\ (f(x) + f'(x)\varepsilon) + (c + 0\varepsilon) &:= (f(x) + c) + (f'(x) + 0)\varepsilon \\ &= (f(x) + c) + f'(x)\varepsilon, \\ (f(x) + f'(x)\varepsilon) - (g(x) + g'(x)\varepsilon) &:= (f(x) - g(x)) + (f'(x) - g'(x))\varepsilon \\ &= (f(x) - g(x)) + (f'(x) - g'(x))\varepsilon. \end{aligned} \quad (4)$$

Now an initial complaint would be: What if $f'(x)$ (or $g'(x)$) is not known—which is usually the case when forward-mode differentiation is applied to a general function f ? The answer is that we don't need to know them as long as the derivatives of the basic operations that make up f are known. That is, we can simply evaluate $f(x + \varepsilon)$, which by Equation 2 should give us the derivative of f at x as the coefficient of ε . In the code example below we define the following composition rules for the elementary functions `sin` and `exp` as

Contents

1. Scalar derivatives
 2. Jacobian-vector products
- Bibliography

$$\begin{aligned}\sin(a + b\varepsilon) &= \sin(a) + b \cos(a)\varepsilon, \\ \exp(a + b\varepsilon) &= \exp(a) + b \exp(a)\varepsilon.\end{aligned}\tag{5}$$

Note that in general b does not have to come from the derivative of a with respect to some variable.

Example 1 (Evaluating a function on a dual number) Let $f(x) = \sin(x \cdot x) + x$ and $x = 2$. Evaluating $f(2 + \varepsilon)$ one operation at a time using the rules above gives

$$\begin{aligned}x &= 2 + \varepsilon, \\ x \cdot x &= 4 + (1 \cdot 2 + 2 \cdot 1)\varepsilon = 4 + 4\varepsilon, \\ \sin(x \cdot x) &= \sin(4) + 4 \cos(4)\varepsilon, \\ \sin(x \cdot x) + x &= (\sin(4) + 2) + (4 \cos(4) + 1)\varepsilon.\end{aligned}\tag{6}$$

The coefficient of ε is $f'(2) = 2 \cdot 2 \cos(2^2) + 1 = 4 \cos(4) + 1$, even though f' was never written down.

Example 1: Evaluating a function on a dual number

Example 2: Scalar derivative in Julia

Example 2 (Scalar derivative in Julia) Julia lets us define the arithmetic of a new type directly, so the scalar case takes only a few lines of code.

julia

```
1 # A small, deliberately incomplete forward-mode AD implementation.
2 struct Dual{T<:Real}
3     value::T
4     deriv::T
5 end
6 # Promote mixed inputs such as Dual(2, 1.0) to a common type
7 Dual(value::Real, deriv::Real) = Dual(promote(value, deriv)...)
8
9 # Product rule: (f * g)' = f' * g + f * g'
10 Base.:(f::Dual, g::Dual) = Dual(
11     f.value * g.value,
12     f.deriv * g.value + f.value * g.deriv,
13 )
14 Base.:(f::Dual, g::Real) = f * Dual(g, zero(g))
15 Base.:(f::Real, g::Dual) = Dual(f, zero(f)) * g
16
17 # Quotient rule: (f / g)' = (f' * g - f * g') / g^2
18 Base.:(f::Dual, g::Dual) = Dual(
19     f.value / g.value,
20     (f.deriv * g.value - f.value * g.deriv) / g.value^2,
21 )
22
23 # Composition rule: (f ∘ g)' = (f' ∘ g) * g'
24 Base.sin(f::Dual) = Dual(sin(f.value), cos(f.value) * f.deriv)
25 Base.exp(f::Dual) = Dual(exp(f.value), exp(f.value) * f.deriv)
26
27 # Addition rules: (f + g)' = f' + g', (f + c)' = f', (c + f)' = f'
28 Base.:+(f::Dual, g::Dual) = Dual(f.value + g.value, f.deriv + g.deriv)
29 Base.:+(f::Dual, c::Real) = f + Dual(c, zero(c))
30 Base.:+(c::Real, g::Dual) = Dual(c, zero(c)) + g
31
32 # Subtraction rules: (-f)' = -f', (f - g)' = f' - g'
33 Base.:-(f::Dual) = Dual(-f.value, -f.deriv)
34 Base.:-(f::Dual, g::Dual) = f + (-g)
35
36 # The derivative is the epsilon coefficient of f(x + 1*epsilon)
37 derivative(f, x) = f(Dual(x, one(x))).deriv
38
39 # Dual is not a Number, so Base's generic ^ does not apply to it and
```

```

40 # f_scalar(x) = sin(x^2) + x would throw a MethodError
41 f_scalar(x) = sin(x * x) + x
42 @assert isapprox(derivative(f_scalar, 2.0), 4cos(4.0) + 1)
43
44 println("f'(2) = ", derivative(f_scalar, 2.0))

```

2. Jacobian-vector products

For a multivariate function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ the simplest extension keeps a single ε and seeds each input x_j with a direction v_j . The dual numbers then track the function value and a directional derivative (a Jacobian-vector product) at the same time, i.e.,

$$f(x + \varepsilon v) = f(x) + (J_f(x)v)\varepsilon, \quad (7)$$

where $J_f(x)$ is the Jacobian. This definition follows directly by looking at the i th entry as

$$f(x + \varepsilon v)_i = f_i(x + \varepsilon v) = f_i(x) + \left(\sum_{j=1}^n \frac{\partial f_i(x)}{\partial x_j} v_j \right) \varepsilon. \quad (8)$$

A thing to note here is that the sum is computed automatically by the dual number arithmetic, and hence we never form the Jacobian. From a scaling perspective, this is a big win as the Jacobian is $m \times n$ in size, and hence can be very large.

The price is that one evaluation gives one product $J_f(x)v$. Its cost is a small multiple of the cost of evaluating f , but the full Jacobian requires n evaluations, one for each unit vector $v = e_j$. Forward mode is therefore well suited when n is small compared to m . In the opposite case reverse mode (backpropagation) is preferred. It computes one vector-Jacobian product $w^\top J_f(x)$ per evaluation, also at a small multiple of the cost of evaluating f , so the full Jacobian requires m evaluations, one for each unit vector $w = e_i$. The price is that the intermediate values of f must be stored for the backward pass. The common case of a scalar loss of many parameters ($m = 1$) needs only a single evaluation; see Backpropagation as a Linear Solve^{1,2}.

¹ Later post: *Backpropagation as a Linear Solve*.
mipals.github.io/blog/2026/10-05-backpropagation-linear-solve/

² In the notation of that post, the Jacobian of a network is $PL^{-1}M^\top$, where L is block lower triangular. Forward mode multiplies v from the right, $P(L^{-1}(M^\top v))$, so the solve with L is a forward substitution that follows the layers in order, exactly as the dual numbers do. Reverse mode multiplies $w^\top$ from the left, $((w^\top P)L^{-1})M^\top$, so the solve becomes the backward substitution $L^\top \lambda = P^\top w$.

Example 3: Jacobian-vector product in Julia

Example 3 (Jacobian-vector product in Julia) Let

$$f(x_1, x_2) = (x_1 x_2 + \sin(x_1), \exp(x_2)). \quad (9)$$

At $x = (2, 3)$ in the direction $v = (1, -1)$, Equation 7 gives

$$J_f(2, 3)v = \begin{bmatrix} 3 + \cos(2) & 2 \\ 0 & \exp(3) \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 + \cos(2) \\ -\exp(3) \end{bmatrix}. \quad (10)$$

julia

```

1 @isdefined(Dual) || include("dual_numbers.jl")
2
3 function jvp(f, x, v)
4     @assert length(x) == length(v)
5     seeded = [Dual(xi, vi) for (xi, vi) in zip(x, v)]
6     result = f(seeded)
7     return map(z -> z.value, result), map(z -> z.deriv, result)
8 end
9
10 f_vector(x) = (x[1] * x[2] + sin(x[1]), exp(x[2]))
11 x, v = [2.0, 3.0], [1.0, -1.0]

```

```
12 value, deriv = jvp(f_vector, x, v)
13 @assert all(isapprox.(value, (6 + sin(2.0), exp(3.0))))
14 @assert all(isapprox.(deriv, (1 + cos(2.0), -exp(3.0))))
15
16 println("f($x) = ", value)
17 println("Jf($x)$v = ", deriv)
```

Bibliography

- [1] J. Revels, M. Lubin, and T. Papamarkou, “Forward-Mode Automatic Differentiation in Julia,” *arXiv*, 2016, [Online]. Available: <https://arxiv.org/abs/1607.07892>