Operator Overloading & Vectorized ”OTI”

Smart Tracking of Coefficients and OTIS Implementation

Santiago Garcia Botero

University of Texas at San Antonio

February 9, 2026

The Challenge: Compute Derivatives of $f(\boldsymbol{x})$

We consider a vector function $f(\boldsymbol{x})$ where $\boldsymbol{x} \in \mathbb{R}^n$. We want the **Gradient** $\nabla f \in \mathbb{R}^n$.

Traditional Finite Difference / Complex Step:

1. Perturb component 1: $f(x_1 + h, x_2, \dots) \rightarrow \partial_1 f$
2. Perturb component 2: $f(x_1, x_2 + h, \dots) \rightarrow \partial_2 f$
3. Repeat n times.

Goal of "OTI" (Operator Type Implementation): One single evaluation $f(\boldsymbol{X}_{OTI})$ that automatically computes the value and **all** n gradient components exactly, using **structured truncation**.

The Mathematical Engine: Hyperdual Numbers

We use **Symbolic Nilpotent Directions**. Let ε be a nilpotent unit such that:

$$\varepsilon \neq 0, \quad \varepsilon^2 = 0$$

Consider the Taylor expansion of any smooth function $f(x + h\varepsilon)$:

$$f(x + h\varepsilon) = f(x) + f'(x)(h\varepsilon) + \frac{1}{2}f''(x)\underbrace{(h\varepsilon)^2}_{0} + \dots$$

All higher-order terms vanish **exactly**.

$$f(x + h\varepsilon) = f(x) + hf'(x)\varepsilon$$

Instead of subtracting numbers, we simply **Read The Coefficient** of ε.

Vectorizing OTI: Multiple Directions

For a vector $\boldsymbol{x} \in \mathbb{R}^n$, we need n distinct derivatives. We introduce n **Orthogonal Nilpotent Units** $\{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n\}$.

Rules:

- $\varepsilon_i^2 = 0$ (Nilpotency)
- $\varepsilon_i \varepsilon_j = 0$ (Truncation of mixed terms at 2nd order)

Our OTI variable structure becomes:

$$u_{oti} = \underbrace{v_u}_{\text{Real Part}} + \sum_{k=1}^{n} (\nabla u)_k \varepsilon_k$$

We track a vector of coefficients $\boldsymbol{g}_u = [(\nabla u)_1, \ldots, (\nabla u)_n]^T$.

The Coefficient Extraction Operator

We define the operator $\text{Coeff}_{\varepsilon_k}$ which extracts the component attached to the direction ε_k.

$$\text{Coeff}_{\varepsilon_k}[u] = \frac{\partial u}{\partial x_k}$$

Implementation Strategy (OTIS): Instead of storing a symbolic string, we store the coefficients in a contiguous array (vector).

$$u_{oti} \equiv \langle v_u, \quad \boldsymbol{g}_u \rangle$$

where $\boldsymbol{g}_u = [g_1, g_2, \ldots, g_n]^T$.

Seeding: Assigning Directions to Inputs

To differentiate with respect to input x_i, we must "attach" the direction ε_i to it.
Input Vector $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]$.
OTI Initialization:

$$
\begin{aligned}
X_1 &= x_1 + 1 \cdot \varepsilon_1 + 0 \cdot \varepsilon_2 + \ldots \\
X_2 &= x_2 + 0 \cdot \varepsilon_1 + 1 \cdot \varepsilon_2 + \ldots \\
&\vdots \\
X_n &= x_n + 0 \cdot \varepsilon_1 + 0 \cdot \varepsilon_2 + \cdots + 1 \cdot \varepsilon_n
\end{aligned}
$$

In vector storage format $\langle v, \boldsymbol{g} \rangle$:

$$X_i = \langle x_i, \quad \boldsymbol{e}_i \rangle$$

The Algebra of Coefficients (Product Rule)

How do we multiply two OTI numbers? Let $A = a + \boldsymbol{g}_a \cdot \varepsilon$ and $B = b + \boldsymbol{g}_b \cdot \varepsilon$.

$$A \cdot B = (a + \boldsymbol{g}_a \cdot \varepsilon)(b + \boldsymbol{g}_b \cdot \varepsilon)$$

Expand terms:

$$= ab + (a\boldsymbol{g}_b) \cdot \varepsilon + (b\boldsymbol{g}_a) \cdot \varepsilon + \underbrace{(\boldsymbol{g}_a \cdot \varepsilon)(\boldsymbol{g}_b \cdot \varepsilon)}_{\text{Higher Order} \approx 0}$$

Collecting coefficients of ε:

$$\text{Coeff}_\varepsilon[A \cdot B] = a\boldsymbol{g}_b + b\boldsymbol{g}_a$$

Analytical Example with OTI Variables

Let $f(x, y) = x^2 y$. Evaluate at $x = 2, y = 3$. We track coefficients ε_1 (for x) and ε_2 (for y).

1. OTI Seeding (Initialization):

$$x_{oti} = \langle 2, [1, 0] \rangle \quad (\text{has } \varepsilon_1)$$

$$y_{oti} = \langle 3, [0, 1] \rangle \quad (\text{has } \varepsilon_2)$$

2. Compute x_{oti}^2 (Using Definition of Multiplication in OTIS): Recall: $A \cdot B = \langle a \cdot b, \quad a\boldsymbol{g}_b + b\boldsymbol{g}_a \rangle$. Here $A = x_{oti}, B = x_{oti}$, so $a = 2, \boldsymbol{g}_a = [1, 0]$.

$$x_{oti} \cdot x_{oti} = \langle 2 \cdot 2, \quad 2 \cdot [1, 0] + 2 \cdot [1, 0] \rangle$$

$$= \langle 4, \quad [2, 0] + [2, 0] \rangle = \langle 4, [4, 0] \rangle$$

3. Compute $f_{oti} = x_{oti}^2 \cdot y_{oti}$: Here $A = x_{oti}^2 = \langle 4, [4, 0] \rangle$ and $B = y_{oti} = \langle 3, [0, 1] \rangle$.

$$f_{oti} = A \cdot B = \langle 4 \cdot 3, \quad 4 \cdot [0, 1] + 3 \cdot [4, 0] \rangle$$

$$f_{oti} = \langle 12, \quad [0, 4] + [12, 0] \rangle = \langle 12, [12, 4] \rangle$$

Result: Value 12, Gradient $\nabla f = [12, 4]^T$.

OTIS Implementation: The Class

We implement the ”Hyperdual” concept using a class that stores the ‘val‘ (scalar) and ‘grad‘ (coefficient array).

```
import numpy as np

class HyperDual:
    def __init__(self, val, grad=None):
        self.val = val # The Real Part

        # The Coefficient Vector of the epsilon units
        if grad is None:
            self.grad = np.zeros_like(val)
        else:
            self.grad = grad

    def __repr__(self):
        return f"OTI({self.val}, {self.grad})"
```

OTIS Implementation: Overloading

We map the coefficient algebra directly to Python operators.

```
    def __add__(self, other):
        # Linearity: Coeff(A+B) = Coeff(A) + Coeff(B)
        if isinstance(other, HyperDual):
            return HyperDual(self.val + other.val, self.grad + other.grad)
        return HyperDual(self.val + other, self.grad)

    def __mul__(self, other):
        # Product Rule: Coeff(AB) = a*g_b + b*g_a
        if isinstance(other, HyperDual):
            new_grad = self.val * other.grad + other.val * self.grad
            return HyperDual(self.val * other.val, new_grad)
        return HyperDual(self.val * other, self.grad * other)

    def sin(self):
        # Chain Rule: Coeff(sin(u)) = cos(u) * g_u
        return HyperDual(np.sin(self.val), np.cos(self.val) * self.grad)
```

Conclusion

By defininining our variables as **OTI Type** (Hyperdual):

- We replace numerical differentiation with **Algebraic Coefficient Tracking**.
- The code looks like normal math ($z = x * y + \sin(x)$).
- The result contains exact derivatives for free.

This is the foundation of modern Autodiff tools like PyTorch and JAX, but implemented here from scratch using the fundamental algebra of nilpotent units.