Tangent Linear and Adjoint Models for Variational Data Assimilation

Marcin Chrust and Sébastien Massart

ECMWF

Data Assimilation Training Course

Outline

1. Introduction
2. Fundamentals of 4D-Var
3. Tangent Linear and Adjoint Models
4. Writing Tangent Linear and Adjoint Models
5. Testing Models
6. Automatic Differentiation
7. Key Facts

4D-Var: Overview and Context

- **4D-Var** is based on minimization of a cost function that measures weighted distance between:
  - Model prediction with respect to observations
  - Model state with respect to background state

- To minimize the cost function, we need:
  1. An initial estimate of the solution
  2. Gradient of the cost function

- The gradient is computed using:
  - Direct model integration (forward in time)
  - Backward time integration of the linearized adjoint model
  - (Le Dimet and Talagrand, 1986)

Course Overview

This presentation covers:

1. Brief overview of 4D-Var with focus on tangent linear and adjoint aspects
2. General definitions of Tangent Linear (TL) and Adjoint (AD) models
3. Why TL and AD models are essential for variational assimilation
4. Writing TL and AD models (Python examples)
5. Testing TL and AD models
6. Automatic differentiation software and frameworks

Companion materials:

- `notebookTLAD.ipynb` — Hands-on exercises (TL/AD coding, testing, 4D-Var)

The 4D-Var Cost Function

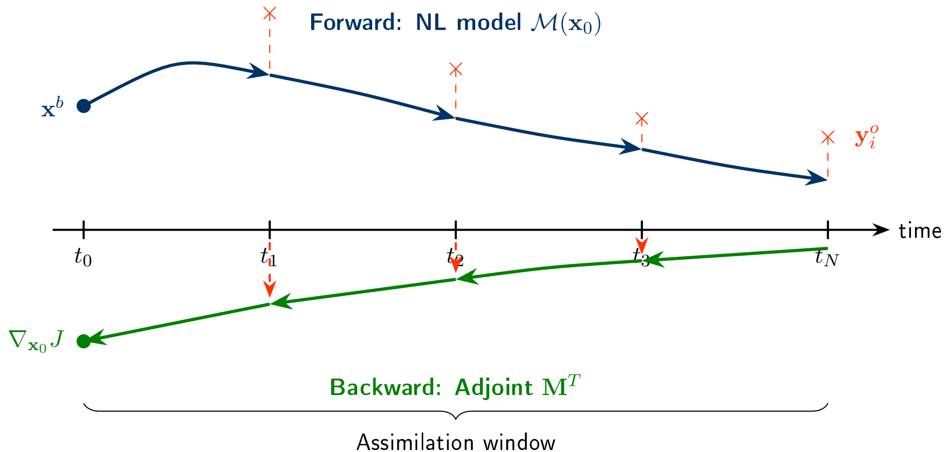
4D-Var minimizes the cost function:

$$J(\mathbf{x}_0) = \frac{1}{2}(\mathbf{x}_0 - \mathbf{x}^b)^T \mathbf{B}^{-1}(\mathbf{x}_0 - \mathbf{x}^b) + \frac{1}{2}\sum_{i=0}^{N}(\mathcal{H}_i\mathcal{M}^i(\mathbf{x}_0) - \mathbf{y}_i^o)^T \mathbf{R}_i^{-1}(\mathcal{H}_i\mathcal{M}^i(\mathbf{x}_0) - \mathbf{y}_i^o) \quad (1)$$

where:

- $\mathbf{x}_0$ = initial state (control variable)
- $\mathbf{x}^b$ = background state
- $\mathbf{B}$ = background error covariance matrix
- $\mathcal{M}^i$ = nonlinear model forecast from t_0 to t_i
- $\mathcal{H}_i$ = observation operator at time t_i
- $\mathbf{y}_i^o$ = observations at time t_i
- $\mathbf{R}_i$ = observation error covariance matrix at time t_i

4D-Var: The Assimilation Window



Forward model propagates state; adjoint propagates sensitivities backward, collecting observation information at each time.

Linearization of the Forward Model

Principle: Linearization is the first-order term of Taylor expansion around a reference state

Consider a nonlinear system:

$$\mathbf{x}(t + \Delta t) = \mathcal{M}(\mathbf{x}(t))$$

Apply Taylor expansion to a perturbed state $\mathbf{x}^* + \delta\mathbf{x}$:

$$\mathcal{M}(\mathbf{x}^* + \delta\mathbf{x}) = \underbrace{\mathcal{M}(\mathbf{x}^*)}_{\mathbf{x}^*(t+\Delta t)} + \left.\frac{\partial\mathcal{M}}{\partial\mathbf{x}}\right|_{\mathbf{x}^*} \delta\mathbf{x} + \mathcal{O}(\|\delta\mathbf{x}\|^2)$$

Subtracting the reference trajectory $\mathbf{x}^*(t + \Delta t) = \mathcal{M}(\mathbf{x}^*)$ and neglecting higher-order terms:

$$\underbrace{\mathcal{M}(\mathbf{x}^* + \delta\mathbf{x}) - \mathbf{x}^*(t + \Delta t)}_{\delta\mathbf{x}(t+\Delta t)} \approx \left.\frac{\partial\mathcal{M}}{\partial\mathbf{x}}\right|_{\mathbf{x}^*} \delta\mathbf{x}(t) = \mathbf{M}(\mathbf{x}^*(t))\delta\mathbf{x}(t)$$

where $\mathbf{M}$ is the Jacobian matrix of $\mathcal{M}$

Definition of Tangent Linear Model

Tangent Linear Model:

Given a nonlinear model $\mathbf{x}_{n+1} = \mathcal{M}(\mathbf{x}_n)$, the tangent linear model computes:

$$\delta\mathbf{x}_{n+1} = \mathbf{M}(\mathbf{x}_n^*)\delta\mathbf{x}_n$$

Properties:

- Computes the evolution of small perturbations
- Requires the nonlinear trajectory $\mathbf{x}_n^*$ (linearization point)
- Cost: approximately 1.5 times the nonlinear model cost
- Essential intermediate step for adjoint development and testing

Notation:

- ECMWF IFS: tangent linear variables have no subscript, trajectory indicated as `x5`
- Common convention: suffix “`d`” for perturbations (TAPENADE convention)

Definition of Adjoint Operator

Adjoint of a Linear Operator:
For a linear operator $\mathbf{M}$, the adjoint operator $\mathbf{M}^*$ satisfies:

$$\langle \mathbf{M}\delta\mathbf{x}, \delta\mathbf{y} \rangle = \langle \delta\mathbf{x}, \mathbf{M}^*\delta\mathbf{y} \rangle$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product

In matrix form:

$$\mathbf{M}^* = \mathbf{M}^T$$

Key properties:

- The adjoint **always exists and is unique** (finite dimensions)
- Adjoint of composite: $(\mathbf{M} \circ \mathbf{N})^T = \mathbf{N}^T \circ \mathbf{M}^T$ (reverse order!)
- Cost: approximately 2–3 times the nonlinear model cost

Notation: We denote adjoint variables with an overline $\overline{x}$ (or suffix `_bar` in code), representing the sensitivity of the cost function with respect to that variable: $\overline{x_i} = \partial J / \partial x_i$.

Matrix Method: Simple 2×2 Example

Consider a simple linear transformation:

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

The Jacobian (tangent linear operator):

$$\mathbf{M} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

The adjoint (transpose of Jacobian):

$$\mathbf{M}^* = \mathbf{M}^T = \begin{pmatrix} a & c \\ b & d \end{pmatrix}$$

Key insight: The adjoint operator reverses the dependencies:

- Forward: $y_1 = ax_1 + bx_2$ depends on x_1 and x_2
- Adjoint: sensitivities from y_1 propagate back via transpose coefficients

Matrix Method: Deriving Line-by-Line Code

From matrix form to line-by-line adjoint:

Step 1: Tangent Linear (forward) – write ALL variables explicitly:

$$\delta x_1 = \delta x_1 \quad \text{(independent variable)}$$
$$\delta x_2 = \delta x_2 \quad \text{(independent variable)}$$
$$\delta y_1 = a \cdot \delta x_1 + b \cdot \delta x_2$$
$$\delta y_2 = c \cdot \delta x_1 + d \cdot \delta x_2$$

Step 2: Adjoint (backward) – reverse order, transpose each statement:

$$\overline{\delta x_1} \mathrel{+}= a \cdot \overline{\delta y_1} + c \cdot \overline{\delta y_2}$$
$$\overline{\delta x_2} \mathrel{+}= b \cdot \overline{\delta y_1} + d \cdot \overline{\delta y_2}$$
$$\overline{\delta y_1} = 0, \quad \overline{\delta y_2} = 0$$

Key rules: += accumulates gradients; dependent bar variables are zeroed after their contributions are distributed

The Adjoint: Transforming Gradients

Key Insight: The adjoint allows us to express gradients with respect to one variable in terms of gradients with respect to another

Given: $\mathbf{y} = \mathcal{M}(\mathbf{x})$ and cost $J = \frac{1}{2}\|\mathbf{y}\|^2$

We want: $\nabla_{\mathbf{x}} J$ (the gradient of J with respect to $\mathbf{x}$)

Derivation using the chain rule:

$$\begin{aligned} \nabla_{\mathbf{x}} J &= \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \frac{\partial J}{\partial \mathbf{y}} \\ &= \mathbf{M}^T \nabla_{\mathbf{y}} J \end{aligned}$$

Why the transpose? Matrix dimensions: $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is $(n_y \times n_x)$, $\frac{\partial J}{\partial \mathbf{y}}$ is $(n_y \times 1)$, so $(\frac{\partial \mathbf{y}}{\partial \mathbf{x}})^T \frac{\partial J}{\partial \mathbf{y}}$ gives $(n_x \times n_y) \times (n_y \times 1) = (n_x \times 1)$

This shows how the adjoint transforms gradients from output space ($\mathbf{y}$) to input space ($\mathbf{x}$)!

Gradient of the 4D-Var Cost Function

The gradient is computed as:

$$\nabla_{\mathbf{x}_0} J = \mathbf{B}^{-1}(\mathbf{x}_0 - \mathbf{x}^b) + \sum_{i=0}^{N} (\mathbf{M}^i)^T \mathbf{H}_i^T \mathbf{R}_i^{-1} (\mathcal{H}_i \mathcal{M}^i(\mathbf{x}_0) - \mathbf{y}_i^o)$$

where:

- $(\mathbf{M}^i)^T$ is the adjoint of the tangent linear model from t_0 to t_i
- $\mathbf{H}_i^T$ is the adjoint of the linearized observation operator at time t_i
- Each observation time contributes via its own adjoint propagation back to t_0

Computational cost:

- 1 forward integration (compute $\mathcal{M}^i(\mathbf{x}_0)$ and observation misfit)
- 1 adjoint integration (backward from $t = N$ to $t = 0$)
- Total: ≈ 3.5 times the nonlinear model cost

Incremental 4D-Var at ECMWF

In practice, ECMWF uses incremental 4D-Var:

Step 1: Compute **departure** using nonlinear model:

$$\mathbf{d}_i = \mathbf{y}_i^o - \mathcal{H}_i(\mathcal{M}^i(\mathbf{x}^b))$$

Step 2: Minimize cost function in increment space using TL/AD:

$$J_{\text{inc}}(\delta\mathbf{x}_0) = \frac{1}{2}(\delta\mathbf{x}_0)^T\mathbf{B}^{-1}\delta\mathbf{x}_0 + \frac{1}{2}\sum_{i=0}^{N}(\mathbf{HM}^i(\delta\mathbf{x}_0) - \mathbf{d}_i)^T\mathbf{R}_i^{-1}(\mathbf{HM}^i(\delta\mathbf{x}_0) - \mathbf{d}_i)$$

Step 3: Update the background:

$$\mathbf{x}^a = \mathbf{x}^b + \delta\mathbf{x}_0^*$$

This approach reduces nonlinearity and improves convergence

Outer Loops and Re-Linearization

Why multiple outer loops?

- The TL/AD models are only valid for small perturbations
- After finding $\delta\mathbf{x}_0^*$, the linearization point may no longer be accurate
- Solution: iterate—update trajectory and re-linearize

ECMWF operational practice (4 outer loops):

1. Run NL model at high resolution $\rightarrow$ compute departures $\mathbf{d}_i$
2. Minimize J_{inc} using TL/AD at reduced resolution (inner loop)
3. Update: $\mathbf{x}^b_{(k+1)} = \mathbf{x}^b_{(k)} + \delta\mathbf{x}_0^*$
4. Re-run NL model from updated state $\rightarrow$ new trajectory $\rightarrow$ repeat

Key benefits:

- TL/AD at lower resolution $\Rightarrow$ much cheaper inner minimization
- Each outer loop reduces nonlinearity error
- Typically converges in 3–4 outer iterations

Think About It: Why Not Just Use Finite Differences?

Question: *How would you compute the gradient $\nabla_{\mathbf{x}_0} J$ without an adjoint?*

Finite difference approach:

$$\frac{\partial J}{\partial x_i} \approx \frac{J(\mathbf{x}_0 + \epsilon \mathbf{e}_i) - J(\mathbf{x}_0)}{\epsilon}, \quad i = 1, \ldots, n$$

This requires $n+1$ forward model integrations to get the full gradient.

For NWP:

- ECMWF IFS state dimension: $n \approx 10^9$
- One forward integration $\approx$ minutes on a supercomputer
- Finite differences: 10^9 integrations $=$ completely infeasible!

The adjoint gives the *exact* gradient in just 1 forward + 1 backward pass, regardless of n.

This is the fundamental reason adjoints exist in data assimilation.

Simple Example: From Math to Python

Forward model equation (Lorenz eq. 1, linear in x_1, x_2):

$$y = -p \cdot x_1 + p \cdot x_2$$

Tangent linear:

$$\delta y = -p \cdot \delta x_1 + p \cdot \delta x_2$$

Python code:

```
# Forward model (Lorenz eq. 1)
y = -p * x[0] + p * x[1]

# Tangent linear (computes perturbation)
dy = -p * dx[0] + p * dx[1]

# Adjoint (backward - accumulate!)
dx_bar[0] += -p * dy_bar
dx_bar[1] +=  p * dy_bar
```

Critical rule: Use += for accumulation, not = assignment!

The Lorenz Model

Lorenz system (3D chaotic system):

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

where $\sigma = 10$ (Prandtl), $\rho = 28$ (Rayleigh), $\beta = 8/3$

Why Lorenz?

- Chaotic dynamics: Small perturbations grow exponentially
- Perfect testbed for TL/adjoint validity and sensitivity analysis
- Simple enough to understand, complex enough to be realistic
- Used extensively in adjoint research and training

To build the TL model, we need the Jacobian of the RHS:

Given $\mathbf{f}(\mathbf{x}) = \begin{pmatrix} \sigma(y-x) \\ x(\rho - z) - y \\ xy - \beta z \end{pmatrix}$, the Jacobian is:

$$\mathbf{J}(\mathbf{x}) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{pmatrix} -\sigma & \sigma & 0 \\ \rho - z & -1 & -x \\ y & x & -\beta \end{pmatrix}$$

Discrete TL model (forward Euler): $\delta \mathbf{x}_{n+1} = \delta \mathbf{x}_n + \Delta t\, \mathbf{J}(\mathbf{x}_n^*)\, \delta \mathbf{x}_n$
Discrete adjoint (backward): $\delta \overline{\mathbf{x}}_n = \delta \overline{\mathbf{x}}_{n+1} + \Delta t\, \mathbf{J}^T(\mathbf{x}_n^*)\, \delta \overline{\mathbf{x}}_{n+1}$

Observations:

- $\mathbf{J}$ depends on the state $\mathbf{x}^*$ (nonlinear terms: xz, xy)
- Linear terms ($-\sigma x$, σy, $-y$, $-\beta z$) give the constant Jacobian entries
- The adjoint uses $\mathbf{J}^T$: note the swapped off-diagonal entries

Lorenz: Nonlinear Model in Python

Note: NL model produces the linearisation trajectory for TL and AD!

```
def lorenz_nonlinear_manual(x0, dt, n_steps=1, sigma=10.0, rho=28.0, beta=8.0/3.0):
    # Initialize trajectory with initial state
    trajectory = [x0.copy()]
    x = x0.copy()

    # Integrate forward in time
    for step in range(n_steps):
        # Compute Lorenz right-hand side (RHS)
        dx_dt = np.zeros(3)
        dx_dt[0] = sigma * (x[1] - x[0])                # dx/dt
        dx_dt[1] = x[0] * (rho - x[2]) - x[1]           # dy/dt
        dx_dt[2] = x[0] * x[1] - beta * x[2]            # dz/dt

        x = x + dt * dx_dt
        trajectory.append(x.copy())

    return x, trajectory
```

Lorenz: Tangent Linear Model in Python (Part 1)

Key point: TL model requires the nonlinear trajectory!

```python
def lorenz_tangent_linear_manual(dx0, dt, trajectory, n_steps, sigma=10.0, rho=28.0,
     beta=8.0/3.0):
    # Initialize perturbation
    dx = dx0.copy()

    # Evolve perturbation forward along the trajectory
    for step in range(n_steps):
        x = trajectory[step]  # Reference state at this time
        # Tangent linear: ddx_dt = J(x) * dx
        ddx_dt = np.zeros(3)
        ddx_dt[0] = sigma * (dx[1] - dx[0])
        ddx_dt[1] = dx[0] * (rho - x[2]) - x[0] * dx[2] - dx[1]
        ddx_dt[2] = dx[0] * x[1] + x[0] * dx[1] - beta * dx[2]
        dx = dx + dt * ddx_dt

    return dx
```

Note: The tangent linear derivatives linearize around each point in the trajectory.

Lorenz: Adjoint Model in Python (Part 1)

Adjoint computes sensitivities backward through the trajectory

```
def lorenz_adjoint_manual(dx_bar_final, dt, trajectory, n_steps, sigma=10.0, rho
    =28.0, beta=8.0/3.0):
    # Initialize sensitivity
    dx_bar = dx_bar_final.copy()

    # Propagate sensitivities backward in time
    for step in range(n_steps - 1, -1, -1):  # Reverse: n_steps-1 to 0
        x = trajectory[step]  # Reference state at this time
        # Adjoint of: dx = dx + dt * ddx_dt  (last TL stmt -> first here)
        ddx_bar_dt = dt * dx_bar
        # Adjoint of: ddx_dt = J(x) * dx  (first TL stmt -> last here)
        dx_bar[0] += -sigma*ddx_bar_dt[0] + (rho-x[2])*ddx_bar_dt[1] + x[1]*
            ddx_bar_dt[2]
        dx_bar[1] += sigma*ddx_bar_dt[0] - ddx_bar_dt[1] + x[0]*ddx_bar_dt[2]
        dx_bar[2] += -x[0]*ddx_bar_dt[1] - beta*ddx_bar_dt[2]

    return dx_bar
```

Complete adjoint implementation: Propagates sensitivities backward through multiple time steps.

Lorenz: Adjoint Model in Python (Part 2)

Example usage of the complete adjoint implementation:

```python
# Get trajectory from nonlinear model
x0 = np.array([1.0, 1.0, 1.0])
dt = 0.01
n_steps = 5
x_final, trajectory = lorenz_nonlinear_manual(x0, dt, n_steps)

# Compute tangent linear
dx0 = np.array([0.1, 0.05, -0.02])
dx_final = lorenz_tangent_linear_manual(dx0, dt, trajectory, n_steps)

# Compute adjoint (reverse operation)
dx_bar_final = np.array([1.0, -0.5, 0.2])
dx_bar_initial = lorenz_adjoint_manual(dx_bar_final, dt, trajectory, n_steps)

print(f"Initial sensitivity: {dx_bar_initial}")
```

Key observations:

- Adjoint returns initial sensitivities given final sensitivities
- Trajectory values are essential for nonlinear coefficients

The TL approximation is only valid for small perturbations over short times.

Validity test: Compare the ratio as perturbation shrinks:

$$r(\epsilon) = \frac{\|\mathcal{M}(\mathbf{x} + \epsilon\,\delta\mathbf{x}) - \mathcal{M}(\mathbf{x})\|}{\|\epsilon\,\mathbf{M}\,\delta\mathbf{x}\|} \xrightarrow{\epsilon \to 0} 1$$

```
for n_steps in [5, 50, 200, 500]:
    for eps in [1e-1, 1e-3, 1e-5, 1e-7]:
        x_pert, _ = lorenz_nonlinear(x0 + eps*dx0, dt, n_steps)
        x_ref, traj = lorenz_nonlinear(x0, dt, n_steps)
        tl_result = lorenz_tangent_linear(eps*dx0, dt, traj, n_steps)
        ratio = np.linalg.norm(x_pert - x_ref) / np.linalg.norm(tl_result)
```

Typical Lorenz results:

- Short window (5 steps): $r \to 1$ cleanly for all ϵ
- Medium window (50 steps): $r \to 1$ only for $\epsilon < 10^{-3}$
- Long window (500 steps): TL diverges—this motivates shorter assimilation windows!

Explore this interactively in the companion Jupyter notebook.

Discretize-then-Differentiate

Two approaches to building TL/AD models:

1. **Differentiate-then-discretize**
   - Derive continuous TL/AD equations
   - Then discretize them
   - Elegant mathematics
   - May not match the NL code exactly

2. **Discretize-then-differentiate** ✓
   - Discretize the NL model first
   - Differentiate the discrete code
   - Exact adjoint of the numerical scheme
   - Required for adjoint test to pass

Our Lorenz example uses approach 2: The TL/AD are derived from the forward Euler discretization, not from the continuous ODEs.

In practice: Approach 2 is standard (ECMWF uses it). The inner product test *requires* exact discrete consistency—it will fail with approach 1.

Line-by-Line Adjoint Coding Rules

Simple transposition rule:

TL equation: $\delta\mathbf{y} = A\delta\mathbf{x}$ becomes Adjoint: $\delta\mathbf{x}^* := \delta\mathbf{x}^* + A^T\delta\mathbf{y}^*$

Critical patterns:

Tangent Linear	Adjoint

`dy = a*dx + b*dz`	`dx_bar += a*dy_bar`
	`dz_bar += b*dy_bar`
	`dy_bar = 0`

`for i in range(N):` `x = f(x, dx)`	`for i in range(N-1, -1, -1):` `dx_bar = adjoint(x, dx_bar)`

Most common errors:

- Using = instead of += (loses prior gradient contributions!)
- Forgetting to initialize _bar variables to 0
- Using trajectory values from wrong time step
- Loop iteration in forward instead of reverse order

Testing the Tangent Linear: Finite Difference Test

Principle: Compare the TL output against a finite-difference approximation:

$$\mathbf{M}\,\delta\mathbf{x} \approx \frac{\mathcal{M}(\mathbf{x}+\epsilon\,\delta\mathbf{x}) - \mathcal{M}(\mathbf{x})}{\epsilon}$$

```
for eps in [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8]:
    fd = (nonlinear(x + eps*dx) - nonlinear(x)) / eps
    tl = tangent_linear(x, dx)
    error = np.linalg.norm(fd - tl) / np.linalg.norm(tl)
    print(f"eps={eps:.0e}  error={error:.6e}")
```

Expected behaviour (sweep over ϵ):

- Error decreases as $\epsilon \to 0$ (truncation error $\sim O(\epsilon)$)
- Error increases for very small ϵ (round-off error $\sim O(\epsilon^{-1})$)
- Sweet spot typically around $\epsilon \approx 10^{-5}$ to 10^{-7}

If the error does NOT decrease: there is a bug in the TL code!

Testing the Adjoint: Inner Product (Dot-Product) Test

The definitive adjoint test. For *any* random vectors $\delta\mathbf{x}$ and $\mathbf{y}$:

$$\langle \mathbf{M}\,\delta\mathbf{x},\ \mathbf{y}\rangle = \langle \delta\mathbf{x},\ \mathbf{M}^T\,\mathbf{y}\rangle$$

```
# Generate random input vectors
dx = np.random.randn(3)
y  = np.random.randn(3)

# TL forward
tl_out = tangent_linear(x, dx)

# Adjoint backward
dx_bar = adjoint(x, y)

# Compare inner products
lhs = np.dot(tl_out, y)
rhs = np.dot(dx, dx_bar)
print(f"LHS = {lhs:.15e}")
print(f"RHS = {rhs:.15e}")
print(f"Ratio LHS/RHS = {lhs/rhs:.15f}")  # Must be 1.0!
```

This test is exact: the ratio must equal 1.0 to machine precision ($\sim 10^{-15}$).

Testing: Interpreting Results and Debugging

What do the test results tell you?

Test	Result	Diagnosis
FD test	Error $\to 0$ as $\epsilon \to 0$	TL is **correct**
FD test	Error does not decrease	**Bug in TL code**
FD test	Error plateaus at $O(1)$	Wrong trajectory or missing term
Dot-product	Ratio $= 1 \pm 10^{-14}$	Adjoint is **correct**
Dot-product	Ratio $= 0.99\dots$ or $1.01\dots$	**Small bug** (sign, coefficient)
Dot-product	Ratio far from 1	**Major bug** (wrong structure)

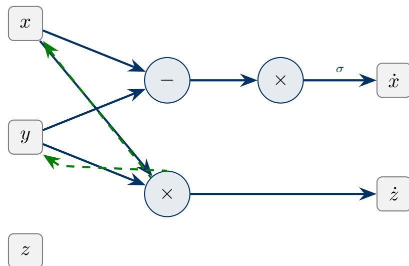
Common adjoint bugs and how they manifest:

- = instead of += $\to$ ratio close to 1 but not exact
- Forgotten zero-initialization of `_bar` variables $\to$ ratio far from 1
- Wrong loop direction $\to$ ratio may vary randomly per trial
- Wrong trajectory time index $\to$ FD test fails for multi-step cases

Practice debugging these cases in the companion notebook.

Every computation forms a directed acyclic graph (DAG):

Forward pass (TL)



Backward pass (Adjoint)

Forward mode (TL):
Propagate perturbations along edges $\rightarrow$

Reverse mode (Adjoint):
Propagate sensitivities against edges $\leftarrow$

AD frameworks (PyTorch, JAX) automate this graph construction and traversal.

Instead of hand-coding adjoints, leverage automatic differentiation!

```
import torch

# Create tensor with gradient tracking
x = torch.tensor([1.0, 1.0, 5.0], requires_grad=True)
rho = 28.0

# Compute Lorenz equation 2
y = x[0] * (rho - x[2]) - x[1]

# Reverse-mode AD (vjp): internally linearises, then transposes
y.backward()  # implicit dy_bar = 1

# x.grad = adjoint variables: [dx0_bar, dx1_bar, dx2_bar]
print(x.grad)
# Output: tensor([23., -1., -1.])
```

`.backward()` implicitly linearises the nonlinear expression, then applies the transpose (vjp).
For scalar y: implicit $\overline{\delta y} = 1$, so `x.grad` $= \mathbf{J}^T \cdot 1 = [\overline{\delta x_0}, \overline{\delta x_1}, \overline{\delta x_2}]$ — the **adjoint (bar) variables**.

Automatic Differentiation: Connection and Advantages

Connection to Adjoint Modeling:

PyTorch's automatic differentiation implements the adjoint method under the hood!

- `y.backward()` computes sensitivities using reverse-mode AD (adjoint)
- `x.grad` contains the adjoint variables (sensitivities)
- This is exactly what we manually implemented in the adjoint functions!

Key advantages:

- Automatic: No hand-coded adjoint needed
- Verified: PyTorch's autograd proven at billions of scales (deep learning)
- Fast: Instant verification against finite differences
- Flexible: Change forward code $\rightarrow$ adjoint updates automatically

Automatic Differentiation: Frameworks and Tools

Modern frameworks supporting automatic differentiation:

Python ML frameworks (reverse mode):

- **PyTorch** - Dynamic graphs, research-friendly, excellent documentation
- **TensorFlow/Keras** - Production-ready, large ecosystems
- **JAX** - Functional, NumPy-like interface, composable transformations

Traditional scientific computing (reverse mode):

- **TAPENADE** (Inria) - Fortran/C, widely used in meteorology
- **TAF/TAMC** - Fortran
- **OpenAD** - Open-source, Fortran/C

When to use:

- **Research/Prototyping:** PyTorch or JAX (rapid development, testing)
- **Production Fortran Code:** TAPENADE, TAF (proven at scale)
- **Validation:** Always compare hand-coded with PyTorch/JAX

Advantages of Automatic Differentiation

Development:

- Mathematically exact gradients
- 10–100× faster development
- Model changes propagate automatically
- Built-in gradient checking

Deployment:

- Proven at billions of parameters
- Easy to experiment with different formulations
- Ideal for prototyping, teaching, and sensitivity analysis

Modern workflow:

1. Prototype and validate with PyTorch/JAX
2. Deploy hand-coded or TAPENADE-generated adjoint for production
3. Use AD output as reference truth for testing

Limitations and Practical Considerations

Realistic limitations to understand:

- **NOT fully automatic:** Requires user guidance on which variables to differentiate
- **Memory overhead:** Must store intermediate activations throughout forward pass (tape)
- **Checkpointing essential:** For long model integrations (like 4D-Var windows), trade compute for memory
- **Trajectory storage:** Can dominate memory for large models with many timesteps
- **Performance:** May be slower than carefully hand-optimized Fortran code

Solution: Memory-efficient checkpointing strategy:

```
from torch.utils.checkpoint import checkpoint

# Save computational graph, recompute activations during backward
output = checkpoint(lorenz_step, x, dt)
# Reduces memory from O(timesteps) to O(sqrt(timesteps))
```

Best practice: Use AD for validation; deploy hand-coded optimized code for production

Linearized Physics: The Hard Part

In NWP, the biggest challenge is linearizing physical parameterizations:

- **Convection:** Highly nonlinear threshold behaviour (on/off switches)
- **Cloud microphysics:** Discontinuous processes (ice/liquid transitions)
- **Radiation:** Expensive and complex, often simplified in TL/AD
- **Surface processes:** Boundary layer switches

ECMWF approach:

- Simplified, smoothed physical parameterizations for TL/AD
- Not a line-by-line derivative of the full-physics NL model
- Carefully tuned to balance accuracy and stability
- See: M. Janisková and P. Lopez (2012), *Linearized physics for data assimilation at ECMWF*

The Lorenz model avoids this complexity, but real NWP systems must address it.

Key Facts and Summary

1. The adjoint **always exists and is unique** (finite dimensions)
   - Questions about existence address TL model existence, not adjoint
2. **Fundamental property:** Adjoints transform gradients with respect to output into gradients with respect to input
   - Enables efficient high-dimensional optimization
3. Cost estimates:
   - TL model: $\approx 1.5\times$ nonlinear code
   - Adjoint model: ≈ 2–$3\times$ nonlinear code
4. **Modern workflow:** Use PyTorch for validation; hand-code with AD tool support (TAPENADE, TAF) for production

References I

Variational Data Assimilation:

- Lorenc, A. C. (1986). Analysis methods for numerical weather prediction. *Q. J. R. Meteorol. Soc.*, **112**, 1177–1194.
- Courtier, P. et al. (1994). Variational assimilation of conventional meteorological observations with a multilevel primitive equation model. *Q. J. R. Meteorol. Soc.*, **120**, 1367–1387.
- Rabier, F. et al. (2000). The ECMWF operational implementation of four-dimensional variational assimilation. *Q. J. R. Meteorol. Soc.*, **126**, 1143–1170.

The Adjoint Technique:

- Le Dimet, F.-X. and Talagrand, O. (1986). Variational algorithms for analysis and assimilation of meteorological observations: theoretical aspects. *Tellus A*, **38**, 97–110.
- Errico, R. M. (1997). What is an adjoint model? *Bull. Am. Meteorol. Soc.*, **78**, 2577–2591.

References II

Tangent Linear Approximation:

- Errico, R. M. et al. (1993). Sensitivity analysis using an adjoint of the PSU–NCAR mesoscale model. *Tellus*, **45A**, 462–477.
- Janisková, M. et al. (1999). Linearized physics for data assimilation at ECMWF. *Monthly Weather Rev.*, **127**, 26–45.

Lorenz Model:

- Huang, X. Y., and X. Yang (1996). Variational data assimilation with the Lorenz model. HIRLAM Technical Report 26.
- Lorenz, E. N. (1963). Deterministic nonperiodic flow. *J. Atmos. Sci.*, **20**, 130–141.

Automatic Differentiation:

- Griewank, A. (2000). Evaluating derivatives: principles and techniques of algorithmic differentiation. SIAM.
- Naumann, U. (2012). The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation. SIAM.
- Giles, M. B. (2008). An extended collection of matrix derivative results for forward and reverse mode algorithmic differentiation. Oxford University Technical Report.

Thank You

Questions?

Contact: Marcin Chrust
ECMWF