Generation of Symbolic Adjoint Derivatives

Efficient row-wise Jacobian evaluation in OpenModelica

Felix Brandt, Philip Hannebohm and Bernhard Bachmann

HSBI

February 2, 2026

Outline

Outline

Why should I care?

- A function $F : \mathbb{R}^n \to \mathbb{R}^m$ with $n \gg m$ has a wide jacobian.
- Finite differences and directional derivatives **do not scale well** for wide Jacobians, they evaluate column-wise so their cost scales with n.
- Adjoint derivatives **scale independently of the number of inputs** and only incur a constant-factor overhead compared to evaluating the original function.
- This is efficient when the number of outputs $n \gg m$, which is common in e.g. ML.

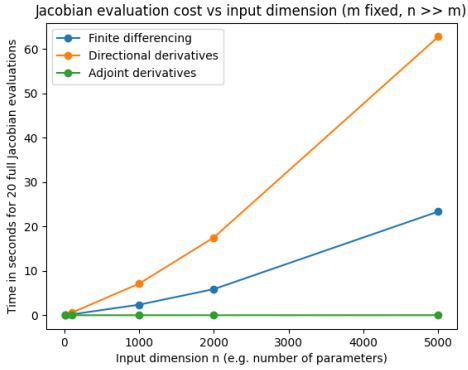


Figure 1: The scaling behaviour of finite differences, directional derivatives and adjoint derivatives when evaluating a wide jacobian.

Outline

What Are **Symbolic** Adjoint Derivatives?

- Adjoint derivatives are commonly implemented as reverse-mode AD in standard ML frameworks such as PyTorch.
- There, the computational graph and the derivatives are computed at runtime, with concrete variable values already inserted.
- In our approach, **symbolic expressions** to compute the adjoint derivatives are generated during compilation.
- So during runtime, these expressions only have to be evaluated not generated.
- It also allows optimization (e.g. simplification) of the generated expressions.

What Are Symbolic **Adjoint** Derivatives?

- A program F can be decomposed into a composition of elementary functions/strong components (here 3)

$$F = F_3 \circ F_2 \circ F_1.$$

- The jacobian J of F is then, by chain rule, a matrix product of the individual jacobians

$$J(x) = J_3(z)J_2(y)J_1(x).$$

- Directional derivatives correspond to column-wise evaluation of the jacobian with seed vector v in input space:

$$J(x)v = J_3(z)J_2(y)J_1(x)v.$$

- Adjoint derivatives correspond to row-wise evaluation with seed vector w in output space:

$$w^T J(x) = w^T J_3(z)J_2(y)J_1(x).$$

What Are Symbolic **Adjoint** Derivatives?

- For $y = F(x_1, x_2)$ with scalars y and x_i with $dy = \nabla F(x_1, x_2)^T \begin{pmatrix} dx_1 \\ dx_2 \end{pmatrix}$.
- As state-transformation:
- $$v_1 = \begin{pmatrix} dx_1 \\ dx_2 \\ dy \end{pmatrix}_1 = J(x_1, x_2) v_0 = \begin{pmatrix} 1 & & \\ & 1 & \\ \partial_1 F(x_1, x_2) & \partial_2 F(x_1, x_2) & 0 \end{pmatrix} \begin{pmatrix} dx_1 \\ dx_2 \\ dy \end{pmatrix}_0$$
- Directional:
$$dy = \partial_1 F(x_1, x_2) dx_1 + \partial_2 F(x_1, x_2) dx_2$$
- $$w_1^T = w_0^T J(x_1, x_2) = \begin{pmatrix} \bar{x_1} & \bar{x_2} & \bar{y} \end{pmatrix} \begin{pmatrix} 1 & & \\ & 1 & \\ \partial_1 F(x_1, x_2) & \partial_2 F(x_1, x_2) & 0 \end{pmatrix}$$
- Adjoint:
$$\bar{x_1} \mathrel{+}= \partial_1 F(x_1, x_2) \bar{y}$$
$$\bar{x_2} \mathrel{+}= \partial_2 F(x_1, x_2) \bar{y}$$

Outline

From Model to **Adjoint Model**

```
model RLC
  parameter Voltage Vb=24;
  parameter Inductance L = 1;
  parameter Resistance R = 100;
  parameter Capacitance C = 1e-3;
  Voltage V;
  Current i_L, i_R, i_C;
equation
  i_R = V / R;
  i_C = -i_R + i_L;
  der(i_L) = (Vb - V) / L;
  der(V) = i_C / C;
end RLC;
```

Figure 2: We want to generate an adjoint model for standard Modelica models.

- The adjoint derivatives are generated during the post-optimization phase (after BLT) in the NB.
- Only continuous strong components are considered.
- For each strong component, one or more corresponding adjoint strong components are generated.
- Adjoint strong components are ordered in reverse execution order (LIFO).

The Adjoint Model

```
model RLC
  parameter Voltage Vb=24;
  parameter Inductance L = 1;
  parameter Resistance R = 100;
  parameter Capacitance C = 1e-3;
  Voltage V;
  Current i_L, i_R, i_C;
equation
  i_R = V / R;
  i_C = -i_R + i_L;
  der(i_L) = (Vb - V) / L;
  der(V) = i_C / C;
end RLC;
```

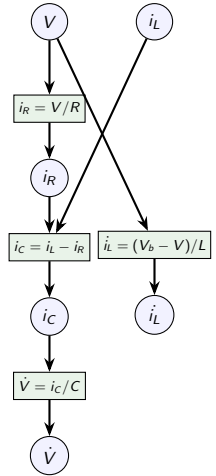
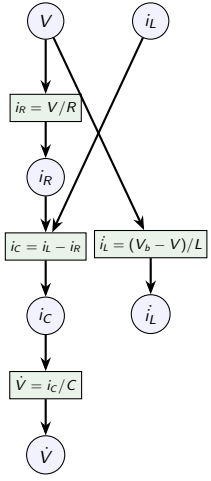


Figure 3: Example Model with rectangular strong components and circular variables.

The Adjoint Model



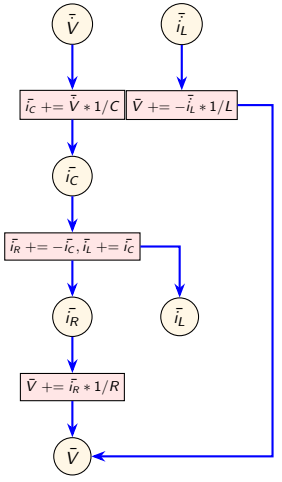


Figure 4: Adjoint Model with rectangular strong components and circular adjoint variables.

From Model Equations to **Adjoint Equations**

For the simple assignment strong component

```
z = 3 * x + 2 * y;
```

the adjoint equations

```
x_adj += 3 * z_adj;
y_adj += 2 * z_adj;
```

are generated.

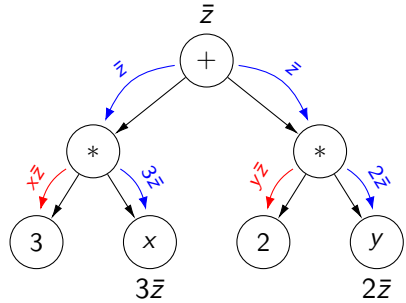


Figure 5: The differentiation happens on the expression tree multiplying local derivatives of the operators down the tree e.g. $\bar{x} = \frac{\partial z}{\partial x} = \frac{\partial z}{\partial z}\frac{\partial((3x)+(2y))}{\partial(3x)}\frac{\partial(3*x)}{\partial x} = \bar{z} * 1 * 3 = 3\bar{z}$. The values in the leaves are written in a map: $\{\bar{x} \mapsto \{3 * \bar{z}\}, \bar{y} \mapsto \{2 * \bar{z}\}\}$.

The **Possible Types** of Strong Components

It can get more complex. Let us look at examples involving:

- `if` statements
- `for` loops
- Algebraic loops

if statements

```
// Forward
if c < 1.0 then
    z = 2 * b;
else
    z = 3 * b;
end if;

// Adjoint
if c < 1.0 then
    b_adj += 2 * z_adj;
else
    b_adj += 3 * z_adj;
end if;

// Same Forward as Assignment
z = if c < 1.0 then 2 * b else 3 * b;

// Same Adjoint as Assignment
b_adj += if c < 1.0 then 2 * z_adj else 3 * z_adj;
```

for loops

```
// Forward
x[1] = 1;
for i in 2:3 then
    x[i] = i * x[i-1];
end for;

// Adjoint
x[3]_adj = 0;
for i in 3:-1:2 then
    x[i-1]_adj += i * x[i]_adj;
end for;
```

The Special Case: Algebraic Loops

- So far, all examples were explicit; algebraic loops, however, are implicit.
- These are typically handled numerically. In our symbolic approach, the equations are constructed symbolically in the backend and the resulting linear system is solved at runtime, without explicitly instantiating the matrices—unlike approaches such as [Ning and McDonnell, 2023].

```
// Forward
der(y)^3 + der(x)^2 - x * y = 0;
der(x)^3 - der(y)^2 - y - x = 0;
// The adjoint of an algebraic loop is a linear system:
2 * der(x)   * l_1 + 3 * der(x)^2 * l_2 = der(x)_adj;
3 * der(y)^2 * l_1 - 2 * der(y)   * l_2 = der(y)_adj;
// With a subsequent matrix vector product:
x_adj += -(-y * l_1 - l_2);
y_adj += -(-x * l_1 - l_2);
```

One Example

1. Build the current master.
2. Execute a .mos file like this:

```
loadString('
  model RLC
    parameter Voltage Vb=24;
    parameter Inductance L = 1;
    parameter Resistance R = 100;
    parameter Capacitance C = 1e-3;
    Voltage V;
    Current i_L;
    Current i_R;
    Current i_C;
  equation
    V = i_R*R;
    C*der(V) = i_C;
    L*der(i_L) = (Vb-V);
    i_L=i_R+i_C;
  end RLC;
');
setCommandLineOptions(''--newBackend --generateDynamicJacobian=symbolicadjoint -d=debugAdjoint'');
buildModel(RLC);
```

3. Inspect the generated ...12jac.c file and find the adjoint code:

```
$pDER_ODE_JAC_ADJ.i_C := $SEED_ODE_JAC_ADJ.$DER.V * 1.0 / C
$pDER_ODE_JAC_ADJ.i_R := -$pDER_ODE_JAC_ADJ.i_C
$pDER_ODE_JAC_ADJ.i_L := $pDER_ODE_JAC_ADJ.i_C
$pDER_ODE_JAC_ADJ.V   := $pDER_ODE_JAC_ADJ.i_R * 1.0 / R -
                         $SEED_ODE_JAC_ADJ.$DER.i_L * 1.0 / L
```

Outline

When should I care?

- Efficient implementation of `fmi3GetAdjointDerivative` to compute

$$\mathbf{v}_{sensitivity}^T = \mathbf{v}_{seed}^T \cdot \mathbf{J}, \quad \mathbf{J} = \begin{bmatrix} \frac{\partial g_1}{\partial v_{known,1}} & \cdots & \frac{\partial g_1}{\partial v_{known,n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_m}{\partial v_{known,1}} & \cdots & \frac{\partial g_m}{\partial v_{known,n}} \end{bmatrix}.$$

- Use of bicoloring to efficiently evaluate sparsely structured Jacobians, e.g., arrowhead matrices:

$$\begin{bmatrix} * & * & * & * \\ * & * & 0 & 0 \\ * & 0 & * & 0 \\ * & 0 & 0 & * \end{bmatrix}.$$

- Generation of symbolic Hessians via Directional-over-Adjoint differentiation.
- Supports efficient dynamic optimization and machine learning workflows.

Outline

Key Takeaways

- Adjoint derivatives enable efficient row-wise and mixed Jacobian evaluation.
- Conceptually, they work by reversing the data flow; for more information see the standard reference [Griewank and Walther, 2008].

Limitations and Current Development

- A prototype is already implemented, supporting basic models (e.g., scalar assignment components), many builtins, and some linear algebra operations.
- Most functionality is available in `NBJacobian.mo` and `NBDifferentiate.mo`.
- Some advanced features (e.g., `for` loops or algorithms) are still under development.
- This topic is currently part of an ongoing bachelor's thesis.

Outline

References I

Griewank, A. and Walther, A. (2008).
Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.
SIAM.

Ning, A. and McDonnell, T. (2023).
Automating steady and unsteady adjoints: Efficiently utilizing implicit and algorithmic differentiation.