Development of discrete adjoint incompressible flow solvers using Automatic Differentiation

J.-D. Müller, D. Jones
Queen Mary, University of London

S. Bayyuk, V. Brown,
ESI-Group, Huntsville AL, USA

FlowHead Adjoint Optimisation Conference, Munich, March 2012

Why bother with Ajoints?

Navier Stokes equations, fixed-point iteration to steady state:

$$R(U(\alpha),\alpha) = 0$$

Linearisation with respect to a design (control) variable α

$$\frac{\partial R}{\partial U}\frac{\partial U}{\partial \alpha} = -\frac{\partial R}{\partial \alpha},$$
$$\mathbf{A}u = f.$$

Sensitivity of an objective function L with respect to α

$$\frac{dL}{d\alpha} = \frac{\partial L}{\partial \alpha} + \frac{\partial L}{\partial U}\frac{\partial U}{\partial \alpha} = \frac{\partial L}{\partial \alpha} + g^T u = \frac{\partial L}{\partial \alpha} + g^T \mathbf{A}^{-1} f$$

$\frac{\partial L}{\partial \alpha}$ is directly computable, $g^T u$ requires an expensive solve for the perturbation flow field u for each α_i.

Why bother with Ajoints?

Navier Stokes equations, fixed-point iteration to steady state:

$$R(U(\alpha), \alpha) = 0$$

Linearisation with respect to a design (control) variable α

$$\begin{gathered}\frac{\partial R}{\partial U}\frac{\partial U}{\partial \alpha} = -\frac{\partial R}{\partial \alpha}, \\ \mathbf{A}u = f.\end{gathered}$$

Sensitivity of an objective function L with respect to α

$$\frac{dL}{d\alpha} = \frac{\partial L}{\partial \alpha} + \frac{\partial L}{\partial U}\frac{\partial U}{\partial \alpha} = \frac{\partial L}{\partial \alpha} + g^T u = \frac{\partial L}{\partial \alpha} + g^T \mathbf{A}^{-1} f$$

$\frac{\partial L}{\partial \alpha}$ is directly computable, $g^T u$ requires an expensive solve for the perturbation flow field u for each α_i.

Why bother with Ajoints?

Navier Stokes equations, fixed-point iteration to steady state:

$$R(U(\alpha), \alpha) = 0$$

Linearisation with respect to a design (control) variable α

$$\frac{\partial R}{\partial U}\frac{\partial U}{\partial \alpha} = -\frac{\partial R}{\partial \alpha},$$
$$\mathbf{A}u = f.$$

Sensitivity of an objective function L with respect to α

$$\frac{dL}{d\alpha} = \frac{\partial L}{\partial \alpha} + \frac{\partial L}{\partial U}\frac{\partial U}{\partial \alpha} = \frac{\partial L}{\partial \alpha} + g^T u = \frac{\partial L}{\partial \alpha} + g^T \mathbf{A}^{-1} f$$

$\frac{\partial L}{\partial \alpha}$ is directly computable, $g^T u$ requires an expensive solve for the perturbation flow field u for each α_i.

Why bother with Ajoints?

Navier Stokes equations, fixed-point iteration to steady state:

$$R(U(\alpha),\alpha)=0$$

Linearisation with respect to a design (control) variable α

$$\begin{gathered}\frac{\partial R}{\partial U}\frac{\partial U}{\partial \alpha}=-\frac{\partial R}{\partial \alpha},\\ \mathbf{A}u=f.\end{gathered}$$

Sensitivity of an objective function L with respect to α

$$\frac{dL}{d\alpha}=\frac{\partial L}{\partial \alpha}+\frac{\partial L}{\partial U}\frac{\partial U}{\partial \alpha}=\frac{\partial L}{\partial \alpha}+g^Tu=\frac{\partial L}{\partial \alpha}+g^T\mathbf{A}^{-1}f$$

$\frac{\partial L}{\partial \alpha}$ is directly computable, g^Tu requires an expensive solve for the perturbation flow field u for each α_i.

Why bother with Ajoints?

Navier Stokes equations, fixed-point iteration to steady state:

$$R(U(\alpha), \alpha) = 0$$

Linearisation with respect to a design (control) variable α

$$\frac{\partial R}{\partial U}\frac{\partial U}{\partial \alpha} = -\frac{\partial R}{\partial \alpha},$$
$$\mathbf{A}u = f.$$

Sensitivity of an objective function L with respect to α

$$\frac{dL}{d\alpha} = \frac{\partial L}{\partial \alpha} + \frac{\partial L}{\partial U}\frac{\partial U}{\partial \alpha} = \frac{\partial L}{\partial \alpha} + g^T u = \frac{\partial L}{\partial \alpha} + g^T \mathbf{A}^{-1} f$$

$\frac{\partial L}{\partial \alpha}$ is directly computable, $g^T u$ requires an expensive solve for the perturbation flow field u for each α_i.

The Adjoint Equations

Regroup the terms in the sensitivity computation:

$$\frac{dL}{d\alpha} = \frac{\partial L}{\partial \alpha} + g^T \mathbf{A}^{-1} f = \frac{\partial L}{\partial \alpha} + \left(\mathbf{A}^{-T} g\right)^T f = \frac{\partial L}{\partial \alpha} + v^T f$$

leads to the definition of the adjoint equation:

$$\mathbf{A}^T v = g$$

$$\left(\frac{\partial R}{\partial U}\right)^T \frac{\partial L}{\partial R}^T = \left(\frac{\partial L}{\partial U}\right)^T.$$

From this follows the *Adjoint Equivalence*

$$g^T u = (\mathbf{A}^T v)^T u = v^T \mathbf{A} u = v^T f$$

Using $v^T f$, needs a single solve of $\mathbf{A}^T v = g$ and the evaluation of f_i for each α_i.

The Adjoint Equations

Regroup the terms in the sensitivity computation:

$$\frac{dL}{d\alpha} = \frac{\partial L}{\partial \alpha} + g^T \mathbf{A}^{-1} f = \frac{\partial L}{\partial \alpha} + \left(\mathbf{A}^{-T} g\right)^T f = \frac{\partial L}{\partial \alpha} + v^T f$$

leads to the definition of the adjoint equation:

$$\mathbf{A}^T v = g$$
$$\left(\frac{\partial R}{\partial U}\right)^T \frac{\partial L}{\partial R}^T = \left(\frac{\partial L}{\partial U}\right)^T .$$

From this follows the *Adjoint Equivalence*

$$g^T u = (\mathbf{A}^T v)^T u = v^T \mathbf{A} u = v^T f$$

Using $v^T f$, needs a single solve of $\mathbf{A}^T v = g$ and the evaluation of f_i for each α_i.

The Adjoint Equations

Regroup the terms in the sensitivity computation:

$$\frac{dL}{d\alpha} = \frac{\partial L}{\partial \alpha} + g^T \mathbf{A}^{-1} f = \frac{\partial L}{\partial \alpha} + \left(\mathbf{A}^{-T} g\right)^T f = \frac{\partial L}{\partial \alpha} + v^T f$$

leads to the definition of the adjoint equation:

$$\mathbf{A}^T v = g$$

$$\left(\frac{\partial R}{\partial U}\right)^T \frac{\partial L}{\partial R}^T = \left(\frac{\partial L}{\partial U}\right)^T .$$

From this follows the *Adjoint Equivalence*

$$g^T u = (\mathbf{A}^T v)^T u = v^T \mathbf{A} u = v^T f$$

Using $v^T f$, needs a single solve of $\mathbf{A}^T v = g$ and the evaluation of f_i for each α_i.

The Adjoint Equations

Regroup the terms in the sensitivity computation:

$$\frac{dL}{d\alpha} = \frac{\partial L}{\partial \alpha} + g^T \mathbf{A}^{-1} f = \frac{\partial L}{\partial \alpha} + \left(\mathbf{A}^{-T} g\right)^T f = \frac{\partial L}{\partial \alpha} + v^T f$$

leads to the definition of the adjoint equation:

$$\mathbf{A}^T v = g$$

$$\left(\frac{\partial R}{\partial U}\right)^T \frac{\partial L}{\partial R}^T = \left(\frac{\partial L}{\partial U}\right)^T .$$

From this follows the *Adjoint Equivalence*

$$g^T u = (\mathbf{A}^T v)^T u = v^T \mathbf{A} u = v^T f$$

Using $v^T f$, needs a single solve of $\mathbf{A}^T v = g$ and the evaluation of f_i for each α_i.

Advantages of adjoint sensitivities

- Each design step requires a solve for $\mathbf{R}(\mathbf{U}) = 0$.
- Gradient-based optimisation requires a gradient for each design variable α_i.
- Using $g^T u$, each α_i needs a solve of $\mathbf{A}u = f$.
- Using $v^T f$, needs a single solve of $\mathbf{A}^T v = g$ and the evaluation of f_i for each α_i.
- Roughly speaking, solving $\mathbf{R}(\mathbf{U}) = 0$, $\mathbf{A}u = f$ and $\mathbf{A}^T v = g$ incur a similar cost.
- Computing f is of the order of a single explicit sweep, simplified boundary formulations exist.
- **Using the adjoint, the cost of gradient calculations for large design problems is essentially constant.**

Advantages of adjoint sensitivities

- Each design step requires a solve for $\mathbf{R}(\mathbf{U}) = 0$.
- Gradient-based optimisation requires a gradient for each design variable α_i.
- Using $g^T u$, each α_i needs a solve of $\mathbf{A}u = f$.
- Using $v^T f$, needs a single solve of $\mathbf{A}^T v = g$ and the evaluation of f_i for each α_i.
- Roughly speaking, solving $\mathbf{R}(\mathbf{U}) = 0$, $\mathbf{A}u = f$ and $\mathbf{A}^T v = g$ incur a similar cost.
- Computing f is of the order of a single explicit sweep, simplified boundary formulations exist.
- **Using the adjoint, the cost of gradient calculations for large design problems is essentially constant.**

Advantages of adjoint sensitivities

- Each design step requires a solve for $\mathbf{R}(\mathbf{U}) = 0$.
- Gradient-based optimisation requires a gradient for each design variable α_i.
- Using $g^T u$, each α_i needs a solve of $\mathbf{A}u = f$.
- Using $v^T f$, needs a single solve of $\mathbf{A}^T v = g$ and the evaluation of f_i for each α_i.
- Roughly speaking, solving $\mathbf{R}(\mathbf{U}) = 0$, $\mathbf{A}u = f$ and $\mathbf{A}^T v = g$ incur a similar cost.
- Computing f is of the order of a single explicit sweep, simplified boundary formulations exist.
- **Using the adjoint, the cost of gradient calculations for large design problems is essentially constant.**

Advantages of adjoint sensitivities

- Each design step requires a solve for $\mathbf{R}(\mathbf{U}) = 0$.
- Gradient-based optimisation requires a gradient for each design variable α_i.
- Using $g^T u$, each α_i needs a solve of $\mathbf{A}u = f$.
- Using $v^T f$, needs a single solve of $\mathbf{A}^T v = g$ and the evaluation of f_i for each α_i.
- Roughly speaking, solving $\mathbf{R}(\mathbf{U}) = 0$, $\mathbf{A}u = f$ and $\mathbf{A}^T v = g$ incur a similar cost.
- Computing f is of the order of a single explicit sweep, simplified boundary formulations exist.
- **Using the adjoint, the cost of gradient calculations for large design problems is essentially constant.**

Advantages of adjoint sensitivities

- Each design step requires a solve for $\mathbf{R}(\mathbf{U}) = 0$.
- Gradient-based optimisation requires a gradient for each design variable α_i.
- Using $g^T u$, each α_i needs a solve of $\mathbf{A}u = f$.
- Using $v^T f$, needs a single solve of $\mathbf{A}^T v = g$ and the evaluation of f_i for each α_i.
- Roughly speaking, solving $\mathbf{R}(\mathbf{U}) = 0$, $\mathbf{A}u = f$ and $\mathbf{A}^T v = g$ incur a similar cost.
- Computing f is of the order of a single explicit sweep, simplified boundary formulations exist.
- **Using the adjoint, the cost of gradient calculations for large design problems is essentially constant.**

Advantages of adjoint sensitivities

- Each design step requires a solve for $\mathbf{R}(\mathbf{U}) = 0$.
- Gradient-based optimisation requires a gradient for each design variable α_i.
- Using $g^T u$, each α_i needs a solve of $\mathbf{A}u = f$.
- Using $v^T f$, needs a single solve of $\mathbf{A}^T v = g$ and the evaluation of f_i for each α_i.
- Roughly speaking, solving $\mathbf{R}(\mathbf{U}) = 0$, $\mathbf{A}u = f$ and $\mathbf{A}^T v = g$ incur a similar cost.
- Computing f is of the order of a single explicit sweep, simplified boundary formulations exist.
- **Using the adjoint, the cost of gradient calculations for large design problems is essentially constant.**

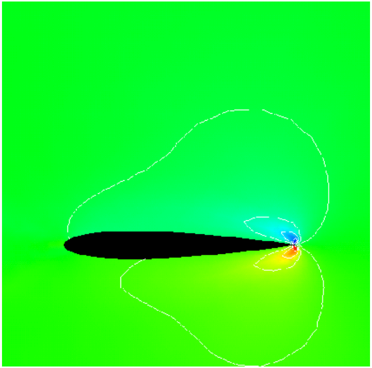
Advantages of adjoint sensitivities

- Each design step requires a solve for $\mathbf{R}(\mathbf{U}) = 0$.
- Gradient-based optimisation requires a gradient for each design variable α_i.
- Using $g^T u$, each α_i needs a solve of $\mathbf{A}u = f$.
- Using $v^T f$, needs a single solve of $\mathbf{A}^T v = g$ and the evaluation of f_i for each α_i.
- Roughly speaking, solving $\mathbf{R}(\mathbf{U}) = 0$, $\mathbf{A}u = f$ and $\mathbf{A}^T v = g$ incur a similar cost.
- Computing f is of the order of a single explicit sweep, simplified boundary formulations exist.
- **Using the adjoint, the cost of gradient calculations for large design problems is essentially constant.**

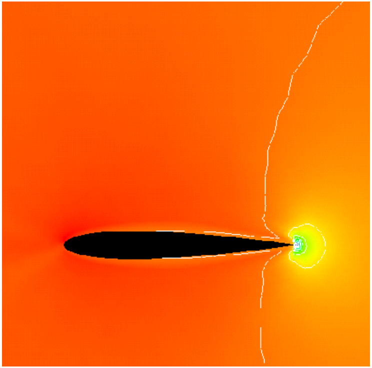
Physical meaning of the adjoint solution: lifting aerofoil

NACA 0012, Ma=0.4, $\alpha = 2^\circ$
Sensitivity w.r.t. lift

mass flux



y-momentum



Contents

Introduction to Algorithmic Differentiation

Development of incompressible adjoint solvers

AD on industrial CFD codes

Summary

Contents

Introduction to Algorithmic Differentiation

Development of incompressible adjoint solvers

AD on industrial CFD codes

Summary

Simple example of AD

Compute $\frac{\partial \mathbf{y}}{\partial x_1}$ for

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi \cos(3x_1 + 2x_2 + x_3) \cdot \pi \sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3) x_1 \end{bmatrix}$$

```
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
sum = v + u
y(1) = v * w
y(2) = w*x(1)
```

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
gv = -gu*pi*sin(u)
gw = gu*pi*cos(u)
gy(1) = gv*w + v*gw
gy(2) = gw*x(1) + gx(1)*w
```

The initial values in the chain rule need to be *seeded*, either set at the beginning of the computation, or computed in a preceding function call.

Simple example of AD

Compute $\frac{\partial \mathbf{y}}{\partial x_1}$ for

$$\mathbf{y} = \left[\begin{array}{c} y_1 \\ y_2 \end{array} \right] = \left[\begin{array}{l} \pi \cos(3x_1 + 2x_2 + x_3) \cdot \pi \sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3) x_1 \end{array} \right]$$

```
u = 3*x(1)+2*x(2)+x(3)          gx(1) = 1
pi = 3.14                       gx(2) = gx(3) = 0
v = pi*cos(u)                   gu = 3*gx(1)+2*gx(2)+gx(3)
w = pi*sin(u)                   gv = -gu*pi*sin(u)
sum = v + u                     gw = gu*pi*cos(u)
y(1) = v * w                    gy(1) = gv*w + v*gw
y(2) = w*x(1)                   gy(2) = gw*x(1) + gx(1)*w
```

The initial values in the chain rule need to be *seeded*, either set at the beginning of the computation, or computed in a preceding function call.

Simple example of AD

Compute $\frac{\partial \mathbf{y}}{\partial x_1}$ for

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi\cos(3x_1 + 2x_2 + x_3)\cdot\pi\sin(3x_1 + 2x_2 + x_3) \\ \pi\cdot\sin(3x_1 + 2x_2 + x_3)x_1 \end{bmatrix}$$

```
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
sum = v + u
y(1) = v * w
y(2) = w*x(1)
```

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
gv = -gu*pi*sin(u)
gw = gu*pi*cos(u)
gy(1) = gv*w + v*gw
gy(2) = gw*x(1) + gx(1)*w
```

The initial values in the chain rule need to be *seeded*, either set at the beginning of the computation, or computed in a preceding function call.

Simple example of AD

Compute $\frac{\partial \mathbf{y}}{\partial x_1}$ for

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi \cos(3x_1 + 2x_2 + x_3) \cdot \pi \sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3) x_1 \end{bmatrix}$$

```
u = 3*x(1)+2*x(2)+x(3)           gx(1) = 1
pi = 3.14                        gx(2) = gx(3) = 0
v = pi*cos(u)                    gu = 3*gx(1)+2*gx(2)+gx(3)
w = pi*sin(u)                    gv = -gu*pi*sin(u)
sum = v + u                      gw = gu*pi*cos(u)
y(1) = v * w                     gy(1) = gv*w + v*gw
y(2) = w*x(1)                    gy(2) = gw*x(1) + gx(1)*w
```

The initial values in the chain rule need to be *seeded*, either set at the beginning of the computation, or computed in a preceding function call.

Simple example of AD

Compute $\frac{\partial \mathbf{y}}{\partial x_1}$ for

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi \cos(3x_1 + 2x_2 + x_3) \cdot \pi \sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3) x_1 \end{bmatrix}$$

```
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
sum = v + u
y(1) = v * w
y(2) = w*x(1)
```

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
gv = -gu*pi*sin(u)
gw = gu*pi*cos(u)
gy(1) = gv*w + v*gw
gy(2) = gw*x(1) + gx(1)*w
```

The initial values in the chain rule need to be *seeded*, either set at the beginning of the computation, or computed in a preceding function call.

Simple example of AD

Compute $\frac{\partial \mathbf{y}}{\partial x_1}$ for

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi\cos(3x_1 + 2x_2 + x_3) \cdot \pi\sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3)x_1 \end{bmatrix}$$

```
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
sum = v + u
y(1) = v * w
y(2) = w*x(1)
```

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
gv = -gu*pi*sin(u)
gw = gu*pi*cos(u)
gy(1) = gv*w + v*gw
gy(2) = gw*x(1) + gx(1)*w
```

The initial values in the chain rule need to be *seeded*, either set at the beginning of the computation, or computed in a preceding function call.

Simple example of AD

Compute $\frac{\partial \mathbf{y}}{\partial x_1}$ for

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi \cos(3x_1 + 2x_2 + x_3) \cdot \pi \sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3) x_1 \end{bmatrix}$$

```
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
sum = v + u
y(1) = v * w
y(2) = w*x(1)
```

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
gv = -gu*pi*sin(u)
gw = gu*pi*cos(u)
gy(1) = gv*w + v*gw
gy(2) = gw*x(1) + gx(1)*w
```

The initial values in the chain rule need to be *seeded*, either set at the beginning of the computation, or computed in a preceding function call.

"Transposing" a statement in reverse-mode

Primal statement: `y(1) = v*w`

forward-mode

gy(1) = gv*w + v*gw

$$\begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_7 = \begin{bmatrix} 1 & & \\ 0 & 1 & \\ w & v & 0 \end{bmatrix} \begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_6$$

$$\dot{z}_{n+1} = E_n \dot{z}_n$$

reverse-mode

vb = vb + w*yb(1)
wb = wb + v*yb(1)

$$\begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_6 = \begin{bmatrix} 1 & 0 & w \\ 0 & 1 & v \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_7$$

$$\begin{aligned} \bar{z}_n E_n &= \bar{z}_{n-1} \\ (\bar{z}_n E_n)^T &= E_n^T \bar{z}_n^T = \bar{z}_{n-1}^T \end{aligned}$$

“Transposing” a statement in reverse-mode

Primal statement: `y(1) = v*w`

forward-mode

gy(1) = gv*w + v*gw

$$\begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_7 = \begin{bmatrix} 1 & & \\ 0 & 1 & \\ w & v & 0 \end{bmatrix} \begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_6$$

$$\dot{z}_{n+1} = E_n \dot{z}_n$$

reverse-mode

vb = vb + w*yb(1)
wb = wb + v*yb(1)

$$\begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_6 = \begin{bmatrix} 1 & 0 & w \\ 0 & 1 & v \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_7$$

$$\begin{aligned} \bar{z}_n E_n &= \bar{z}_{n-1} \\ (\bar{z}_n E_n)^T &= E_n^T \bar{z}_n^T = \bar{z}_{n-1}^T \end{aligned}$$

"Transposing" a statement in reverse-mode

Primal statement: `y(1) = v*w`

forward-mode

gy(1) = gv*w + v*gw

$$\begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_7 = \begin{bmatrix} 1 & & \\ 0 & 1 & \\ w & v & 0 \end{bmatrix} \begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_6$$

$$\dot{z}_{n+1} = E_n \dot{z}_n$$

reverse-mode

vb = vb + w*yb(1)

wb = wb + v*yb(1)

$$\begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_6 = \begin{bmatrix} 1 & 0 & w \\ 0 & 1 & v \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_7$$

$$\begin{aligned} \bar{z}_n E_n &= \bar{z}_{n-1} \\ (\bar{z}_n E_n)^T &= E_n^T \bar{z}_n^T = \bar{z}_{n-1}^T \end{aligned}$$

“Transposing” a statement in reverse-mode

Primal statement: `y(1) = v*w`

forward-mode

gy(1) = gv*w + v*gw

$$\begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_7 = \begin{bmatrix} 1 & & \\ 0 & 1 & \\ w & v & 0 \end{bmatrix} \begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_6$$

$$\dot{z}_{n+1} = E_n \dot{z}_n$$

reverse-mode

vb = vb + w*yb(1)
wb = wb + v*yb(1)

$$\begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_6 = \begin{bmatrix} 1 & 0 & w \\ 0 & 1 & v \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_7$$

$$\begin{aligned} \bar{z}_n E_n &= \bar{z}_{n-1} \\ (\bar{z}_n E_n)^T &= E_n^T \bar{z}_n^T = \bar{z}_{n-1}^T \end{aligned}$$

“Transposing” a statement in reverse-mode

Primal statement: `y(1) = v*w`

forward-mode

reverse-mode

gy(1) = gv*w + v*gw

vb = vb + w*yb(1)
wb = wb + v*yb(1)

$$\begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_7 = \begin{bmatrix} 1 & & \\ 0 & 1 & \\ w & v & 0 \end{bmatrix} \begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_6$$

$$\begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_6 = \begin{bmatrix} 1 & 0 & w \\ 0 & 1 & v \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_7$$

$$\dot{z}_{n+1} = E_n \dot{z}_n$$

$$\begin{aligned} \bar{z}_n E_n &= \bar{z}_{n-1} \\ (\bar{z}_n E_n)^T &= E_n^T \bar{z}_n^T = \bar{z}_{n-1}^T \end{aligned}$$

"Transposing" a statement in reverse-mode

Primal statement: `y(1) = v*w`

forward-mode

reverse-mode

gy(1) = gv*w + v*gw

vb = vb + w*yb(1)
wb = wb + v*yb(1)

$$\begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_7 = \begin{bmatrix} 1 & & \\ 0 & 1 & \\ w & v & 0 \end{bmatrix} \begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_6$$

$$\begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_6 = \begin{bmatrix} 1 & 0 & w \\ 0 & 1 & v \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_7$$

$$\dot{z}_{n+1} = E_n \dot{z}_n$$

$$\begin{aligned} \bar{z}_n E_n &= \bar{z}_{n-1} \\ \left(\bar{z}_n E_n\right)^T &= E_n^T \bar{z}_n^T = \bar{z}_{n-1}^T \end{aligned}$$

"Transposing" a statement in reverse-mode

Primal statement: `y(1) = v*w`

forward-mode

$$gy(1) = gv{*}w + v{*}gw$$

$$\begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_7 = \begin{bmatrix} 1 & & \\ 0 & 1 & \\ w & v & 0 \end{bmatrix} \begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_6$$

$$\dot{z}_{n+1} = E_n \dot{z}_n$$

reverse-mode

$$vb = vb + w{*}yb(1)$$
$$wb = wb + v{*}yb(1)$$

$$\begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_6 = \begin{bmatrix} 1 & 0 & w \\ 0 & 1 & v \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_7$$

$$\begin{aligned} \bar{z}_n E_n &= \bar{z}_{n-1} \\ (\bar{z}_n E_n)^T &= E_n^T \bar{z}_n^T = \bar{z}_{n-1}^T \end{aligned}$$

Example of reverse-mode AD

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi\cos(3x_1+2x_2+x_3)\cdot\pi\sin(3x_1+2x_2+x_3) \\ \pi\cdot\sin(3x_1+2x_2+x_3)x_1 \end{bmatrix}$$

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
gv = -gu*pi*sin(u)
v = pi*cos(u)
gw = gu*pi*cos(u)
w = pi*sin(u)
gy(1) = gv*w + v*gw
y(1) = v * w
gy(2) = gw*x(1) + gx(1)*w
y(2) = w*x(1)
```

```
yb(1) = 1., yb(0) = 0.
u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
xb(:)  = 0.
wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb -
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Example of reverse-mode AD

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi\cos(3x_1 + 2x_2 + x_3)\cdot\pi\sin(3x_1 + 2x_2 + x_3) \\ \pi\cdot\sin(3x_1 + 2x_2 + x_3)x_1 \end{bmatrix}$$

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
gv = -gu*pi*sin(u)
v = pi*cos(u)
gw = gu*pi*cos(u)
w = pi*sin(u)
gy(1) = gv*w + v*gw
y(1) = v * w
gy(2) = gw*x(1) + gx(1)*w
y(2) = w*x(1)
```

```
yb(1) = 1., yb(0) = 0.

u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)

xb(:)  = 0.

wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb -
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Example of reverse-mode AD

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi \cos(3x_1 + 2x_2 + x_3) \cdot \pi \sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3) x_1 \end{bmatrix}$$

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
gv = -gu*pi*sin(u)
v = pi*cos(u)
gw = gu*pi*cos(u)
w = pi*sin(u)
gy(1) = gv*w + v*gw
y(1) = v * w
gy(2) = gw*x(1) + gx(1)*w
y(2) = w*x(1)
```

```
yb(1) = 1., yb(0) = 0.

u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)

xb(:)  = 0.

wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb -
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Example of reverse-mode AD

$$\mathbf{y} = \left[\begin{array}{c} y_1 \\ y_2 \end{array} \right] = \left[\begin{array}{l} \pi\cos(3x_1 + 2x_2 + x_3) \cdot \pi\sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3)x_1 \end{array} \right]$$

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
gv = -gu*pi*sin(u)
v = pi*cos(u)
gw = gu*pi*cos(u)
w = pi*sin(u)
gy(1) = gv*w + v*gw
y(1) = v * w
gy(2) = gw*x(1) + gx(1)*w
y(2) = w*x(1)
```

```
yb(1) = 1., yb(0) = 0.

u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)

xb(:)  = 0.

wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb -
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Example of reverse-mode AD

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi\cos(3x_1 + 2x_2 + x_3) \cdot \pi\sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3)x_1 \end{bmatrix}$$

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
gv = -gu*pi*sin(u)
v = pi*cos(u)
gw = gu*pi*cos(u)
w = pi*sin(u)
gy(1) = gv*w + v*gw
y(1) = v * w
gy(2) = gw*x(1) + gx(1)*w
y(2) = w*x(1)
```

```
yb(1) = 1., yb(0) = 0.

u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)

xb(:)  = 0.

wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb -
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Example of reverse-mode AD

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi \cos(3x_1 + 2x_2 + x_3) \cdot \pi \sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3) x_1 \end{bmatrix}$$

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
gv = -gu*pi*sin(u)
v = pi*cos(u)
gw = gu*pi*cos(u)
w = pi*sin(u)
gy(1) = gv*w + v*gw
y(1) = v * w
gy(2) = gw*x(1) + gx(1)*w
y(2) = w*x(1)
```

```
yb(1) = 1., yb(0) = 0.
u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
xb(:)  = 0.

wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb -
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Example of reverse-mode AD

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi\cos(3x_1 + 2x_2 + x_3)\cdot\pi\sin(3x_1 + 2x_2 + x_3) \\ \pi\cdot\sin(3x_1 + 2x_2 + x_3)x_1 \end{bmatrix}$$

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
gv = -gu*pi*sin(u)
v = pi*cos(u)
gw = gu*pi*cos(u)
w = pi*sin(u)
gy(1) = gv*w + v*gw
y(1) = v * w
gy(2) = gw*x(1) + gx(1)*w
y(2) = w*x(1)
```

```
yb(1) = 1., yb(0) = 0.
u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
xb(:)  = 0.
wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb -
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Example of reverse-mode AD

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi\cos(3x_1 + 2x_2 + x_3) \cdot \pi\sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3)x_1 \end{bmatrix}$$

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
gv = -gu*pi*sin(u)
v = pi*cos(u)
gw = gu*pi*cos(u)
w = pi*sin(u)
gy(1) = gv*w + v*gw
y(1) = v * w
gy(2) = gw*x(1) + gx(1)*w
y(2) = w*x(1)
```

```
yb(1) = 1., yb(0) = 0.

u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)

xb(:)  = 0.

wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb -
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Example of reverse-mode AD

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi \cos(3x_1 + 2x_2 + x_3) \cdot \pi \sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3) x_1 \end{bmatrix}$$

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
gv = -gu*pi*sin(u)
v = pi*cos(u)
gw = gu*pi*cos(u)
w = pi*sin(u)
gy(1) = gv*w + v*gw
y(1) = v * w
gy(2) = gw*x(1) + gx(1)*w
y(2) = w*x(1)
```

```
yb(1) = 1., yb(0) = 0.
u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
xb(:)  = 0.
wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb -
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Example of reverse-mode AD

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi\cos(3x_1+2x_2+x_3)\cdot\pi\sin(3x_1+2x_2+x_3) \\ \pi\cdot\sin(3x_1+2x_2+x_3)x_1 \end{bmatrix}$$

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
gv = -gu*pi*sin(u)
v = pi*cos(u)
gw = gu*pi*cos(u)
w = pi*sin(u)
gy(1) = gv*w + v*gw
y(1) = v * w
gy(2) = gw*x(1) + gx(1)*w
y(2) = w*x(1)
```

```
yb(1) = 1., yb(0) = 0.
u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
xb(:)  = 0.
wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb -
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Example of reverse-mode AD

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi\cos(3x_1 + 2x_2 + x_3) \cdot \pi\sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3)x_1 \end{bmatrix}$$

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
gv = -gu*pi*sin(u)
v = pi*cos(u)
gw = gu*pi*cos(u)
w = pi*sin(u)
gy(1) = gv*w + v*gw
y(1) = v * w
gy(2) = gw*x(1) + gx(1)*w
y(2) = w*x(1)
```

```
yb(1) = 1., yb(0) = 0.
u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
xb(:)  = 0.
wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb -
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Example of reverse-mode AD

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi\cos(3x_1 + 2x_2 + x_3) \cdot \pi\sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3)x_1 \end{bmatrix}$$

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
gv = -gu*pi*sin(u)
v = pi*cos(u)
gw = gu*pi*cos(u)
w = pi*sin(u)
gy(1) = gv*w + v*gw
y(1) = v * w
gy(2) = gw*x(1) + gx(1)*w
y(2) = w*x(1)
```

```
yb(1) = 1., yb(0) = 0.
u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
xb(:)  = 0.
wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb -
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Example of reverse-mode AD

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi\cos(3x_1 + 2x_2 + x_3) \cdot \pi\sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3)x_1 \end{bmatrix}$$

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
gv = -gu*pi*sin(u)
v = pi*cos(u)
gw = gu*pi*cos(u)
w = pi*sin(u)
gy(1) = gv*w + v*gw
y(1) = v * w
gy(2) = gw*x(1) + gx(1)*w
y(2) = w*x(1)
```

```
yb(1) = 1., yb(0) = 0.
u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
xb(:)  = 0.
wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb -
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Example of reverse-mode AD

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi\cos(3x_1 + 2x_2 + x_3) \cdot \pi\sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3)x_1 \end{bmatrix}$$

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
gv = -gu*pi*sin(u)
v = pi*cos(u)
gw = gu*pi*cos(u)
w = pi*sin(u)
gy(1) = gv*w + v*gw
y(1) = v * w
gy(2) = gw*x(1) + gx(1)*w
y(2) = w*x(1)
```

```
yb(1) = 1., yb(0) = 0.
u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
xb(:)  = 0.
wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb -
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Implementation the reverse-mode AD

- For each cost-function we need to seed with $\bar{y}_i = 1$.
- We obtain all the derivatives of y_i w.r.t. all x in one invocation.
- The logic is followed in reverse, hence we need to store or recompute all the intermediate values needed to compute the derivatives.

Contents

Introduction to Algorithmic Differentiation

Development of incompressible adjoint solvers

AD on industrial CFD codes

Summary

Development of incompressible adjoint solvers

- In-house code gpde is a compact incompressible CFD code (5,000 lines) written as a test-bed for developing adjoint N-S fields.
- Fortran 90/95, using its more modern programming features.
- The algorithm computes laminar/turbulent flow through complex 2/3D geometries.
- Code design mimicks typical CFD code setup and algorithms, exploiting support by Automatic differentiation tools to the maximum.
- Via the makefile, either the primal, primal with tangent or with adjoint can be built automatically.
- Run-time ratio of adjoint over primal is approximately 2.

Development of incompressible adjoint solvers

- In-house code `gpde` is a compact incompressible CFD code (5,000 lines) written as a test-bed for developing adjoint N-S fields.
- Fortran 90/95, using its more modern programming features.
- The algorithm computes laminar/turbulent flow through complex 2/3D geometries.
- Code design mimicks typical CFD code setup and algorithms, exploiting support by Automatic differentiation tools to the maximum.
- Via the makefile, either the primal, primal with tangent or with adjoint can be built automatically.
- Run-time ratio of adjoint over primal is approximately 2.

Development of incompressible adjoint solvers

- In-house code `gpde` is a compact incompressible CFD code (5,000 lines) written as a test-bed for developing adjoint N-S fields.
- Fortran 90/95, using its more modern programming features.
- The algorithm computes laminar/turbulent flow through complex 2/3D geometries.
- Code design mimicks typical CFD code setup and algorithms, exploiting support by Automatic differentiation tools to the maximum.
- Via the makefile, either the primal, primal with tangent or with adjoint can be built automatically.
- Run-time ratio of adjoint over primal is approximately 2.

Development of incompressible adjoint solvers

- In-house code gpde is a compact incompressible CFD code (5,000 lines) written as a test-bed for developing adjoint N-S fields.
- Fortran 90/95, using its more modern programming features.
- The algorithm computes laminar/turbulent flow through complex 2/3D geometries.
- Code design mimicks typical CFD code setup and algorithms, exploiting support by Automatic differentiation tools to the maximum.
- Via the makefile, either the primal, primal with tangent or with adjoint can be built automatically.
- Run-time ratio of adjoint over primal is approximately 2.

Development of incompressible adjoint solvers

- In-house code `gpde` is a compact incompressible CFD code (5,000 lines) written as a test-bed for developing adjoint N-S fields.
- Fortran 90/95, using its more modern programming features.
- The algorithm computes laminar/turbulent flow through complex 2/3D geometries.
- Code design mimicks typical CFD code setup and algorithms, exploiting support by Automatic differentiation tools to the maximum.
- Via the makefile, either the primal, primal with tangent or with adjoint can be built automatically.
- Run-time ratio of adjoint over primal is approximately 2.

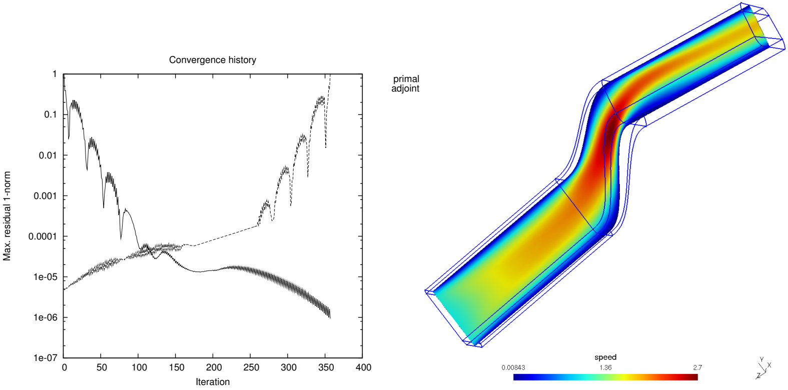
Development of incompressible adjoint solvers

- In-house code `gpde` is a compact incompressible CFD code (5,000 lines) written as a test-bed for developing adjoint N-S fields.
- Fortran 90/95, using its more modern programming features.
- The algorithm computes laminar/turbulent flow through complex 2/3D geometries.
- Code design mimicks typical CFD code setup and algorithms, exploiting support by Automatic differentiation tools to the maximum.
- Via the makefile, either the primal, primal with tangent or with adjoint can be built automatically.
- Run-time ratio of adjoint over primal is approximately 2.

Validation case: VW S-Bend

- Simplified vehicle climatisation duct.
- Uniform flow at inlet, shape modifications only in the bend.
- Objective: total pressure loss. Sensitivities are computed w.r.t. vertex coordinates.
- Turbulent viscosity is approximated using the Spalart-Allmaras model. For $y+ > 11.225$ the standard wall function approximates the near-wall turbulent viscosity.

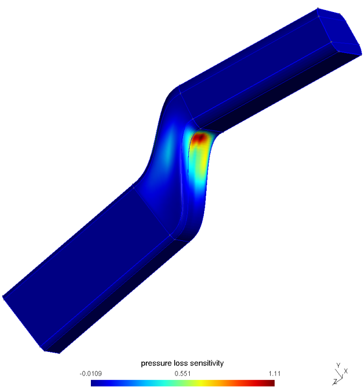
Duct flow case with turb. model, $Re_{\mathrm{H}} = 60$



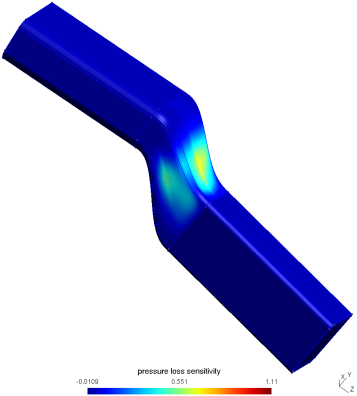
(a) Convergence of primal and adjoint

(b) Fluid speed

Duct flow case with turb. model, $Re_{\mathrm{H}} = 60$

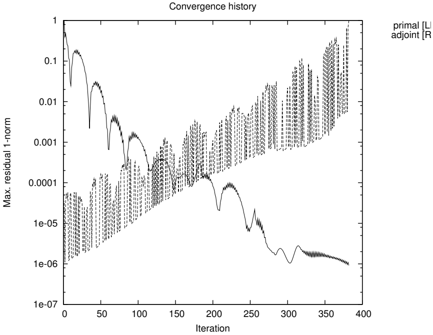


(c) Bottom view

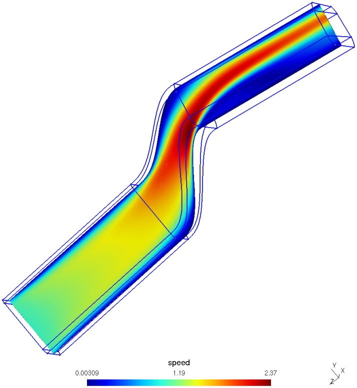


(d) Top view

Duct flow case with turb. model, $Re_{\mathrm{H}} = 600$

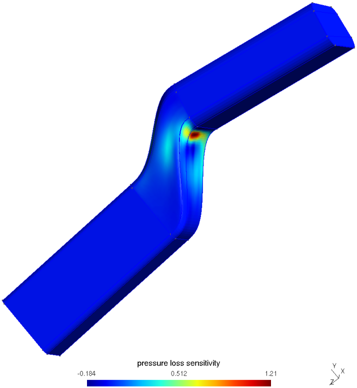


(e) Convergence of primal and adjoint

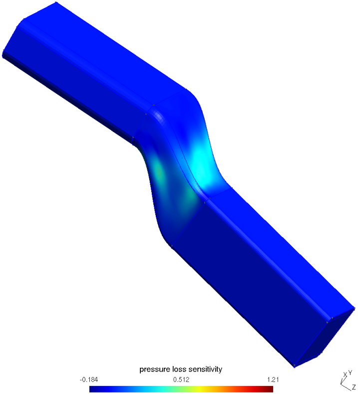


(f) Fluid speed

Duct flow case with turb. model, $Re_{\mathrm{H}} = 600$



(g) Bottom view



(h) Top view

Contents

Introduction to Algorithmic Differentiation

Development of incompressible adjoint solvers

AD on industrial CFD codes

Summary

AD on industrial CFD codes

- A number of industrial CFD packages are intending to have (Star-CCM+) or already have (Fluent 13, OpenFOAM 2.0) adjoint versions.
- So far, none of the current sensitivity implementations use automatic differentiation (AD) to generate the sensitivity algorithm.
- Here we apply AD to incompressible commercial flow solver, ESI's ACE+.
- Original / Full Source Code Size: 3000 files, 1.1M lines / 40MB of source code.
- Reduced Kernel Code Size: 680 files, 230K lines / 8MB of source code, with some Fortran 90 features suppressed or eliminated.

AD on industrial CFD codes

- A number of industrial CFD packages are intending to have (Star-CCM+) or already have (Fluent 13, OpenFOAM 2.0) adjoint versions.
- So far, none of the current sensitivity implementations use automatic differentiation (AD) to generate the sensitivity algorithm.
- Here we apply AD to incompressible commercial flow solver, ESI's ACE+.
- Original / Full Source Code Size: 3000 files, 1.1M lines / 40MB of source code.
- Reduced Kernel Code Size: 680 files, 230K lines / 8MB of source code, with some Fortran 90 features suppressed or eliminated.

AD on industrial CFD codes

- A number of industrial CFD packages are intending to have (Star-CCM+) or already have (Fluent 13, OpenFOAM 2.0) adjoint versions.
- So far, none of the current sensitivity implementations use automatic differentiation (AD) to generate the sensitivity algorithm.
- Here we apply AD to incompressible commercial flow solver, ESI's ACE+.
- Original / Full Source Code Size: 3000 files, 1.1M lines / 40MB of source code.
- Reduced Kernel Code Size: 680 files, 230K lines / 8MB of source code, with some Fortran 90 features suppressed or eliminated.

AD on industrial CFD codes

- To aid the construction of the sensitivity algorithm, source code pre- and post-processing is performed either side of differentiation.
- Pre- and post-processing involve
    1. remove dead code in the original source code via C preprocessor pragmas.
    2. reorganise modules, types and procedures in the generated source code in addition to optimising fixed-point iterators and introducing library functions where appropriate (such as the sparse linear solver).
- Source code processing is where the bulk of the work lies in order to successfully generate adjoint algorithms using AD, but the tools are very difficult to implement.

AD on industrial CFD codes

- To aid the construction of the sensitivity algorithm, source code pre- and post-processing is performed either side of differentiation.
- Pre- and post-processing involve
    1. remove dead code in the original source code via C preprocessor pragmas.
    2. reorganise modules, types and procedures in the generated source code in addition to optimising fixed-point iterators and introducing library functions where appropriate (such as the sparse linear solver).
- Source code processing is where the bulk of the work lies in order to successfully generate adjoint algorithms using AD, but the tools are very difficult to implement.

AD on industrial CFD codes

- To aid the construction of the sensitivity algorithm, source code pre- and post-processing is performed either side of differentiation.
- Pre- and post-processing involve
    1. remove dead code in the original source code via C preprocessor pragmas.
    2. reorganise modules, types and procedures in the generated source code in addition to optimising fixed-point iterators and introducing library functions where appropriate (such as the sparse linear solver).
- Source code processing is where the bulk of the work lies in order to successfully generate adjoint algorithms using AD, but the tools are very difficult to implement.

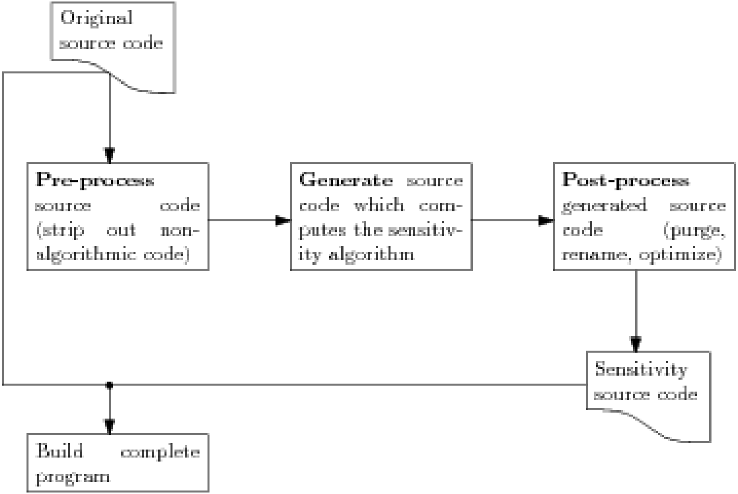
AD on industrial CFD codes

- To aid the construction of the sensitivity algorithm, source code pre- and post-processing is performed either side of differentiation.
- Pre- and post-processing involve
  1. remove dead code in the original source code via C preprocessor pragmas.
  2. reorganise modules, types and procedures in the generated source code in addition to optimising fixed-point iterators and introducing library functions where appropriate (such as the sparse linear solver).
- Source code processing is where the bulk of the work lies in order to successfully generate adjoint algorithms using AD, but the tools are very difficult to implement.

AD on industrial CFD codes

- To aid the construction of the sensitivity algorithm, source code pre- and post-processing is performed either side of differentiation.
- Pre- and post-processing involve
    1. remove dead code in the original source code via C preprocessor pragmas.
    2. reorganise modules, types and procedures in the generated source code in addition to optimising fixed-point iterators and introducing library functions where appropriate (such as the sparse linear solver).
- Source code processing is where the bulk of the work lies in order to successfully generate adjoint algorithms using AD, but the tools are very difficult to implement.

AD on industrial CFD codes: flow-graph



Original source code

Pre-process source code (strip out non-algorithmic code)

Generate source code which computes the sensitivity algorithm

Post-process generated source code (purge, rename, optimize)

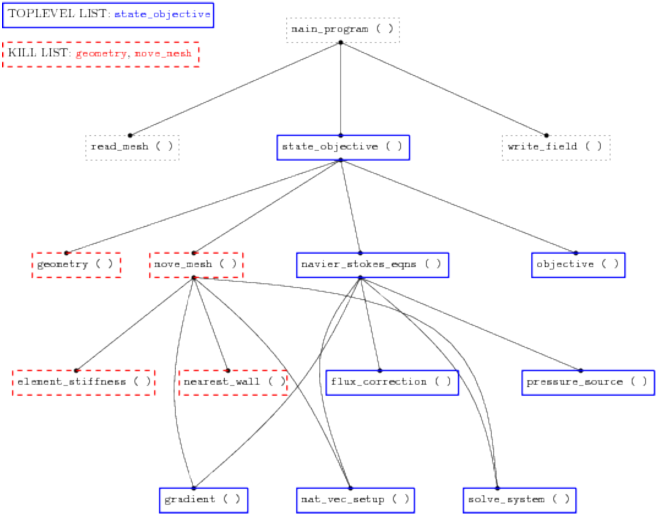
Sensitivity source code

Build complete program

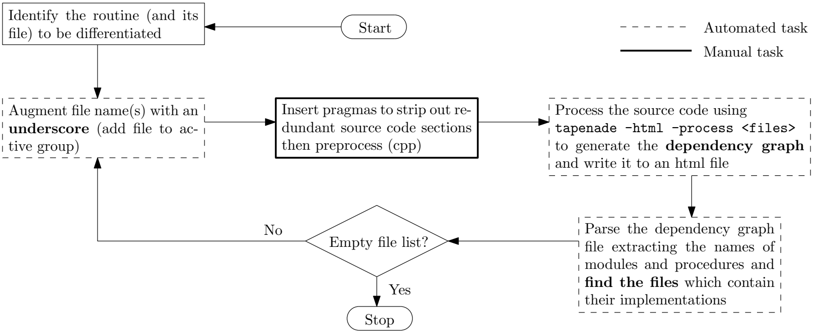
Pre-processing

- This task identifies and extracts the relevant source code which needs to be differentiated from the entire program source code.
- For large programs (100,000+ lines) this becomes tedious and error-prone to perform manually.
- By making use of the call-graph generated by AD tool Tapenade, the tedium of tracing dependencies is removed.
- Knowledge is still required to know which dependent routines are non-essential so can be ignored.

Pre-processing: example of call tree to be pruned



Pre-processing: pruning algorithm



Start

Identify the routine (and its file) to be differentiated

Augment file name(s) with an **underscore** (add file to active group)

Insert pragmas to strip out redundant source code sections then preprocess (cpp)

Process the source code using `tapenade -html -process <files>` to generate the **dependency graph** and write it to an html file

Parse the dependency graph file extracting the names of modules and procedures and **find the files** which contain their implementations

Empty file list?

No

Yes

Stop

- - - - - - Automated task

——— Manual task

Pre-processing: primal source equipped with pragmas

```
module matrix_utils_m
contains

#ifndef STRIP_AD
!! passive
subroutine print_matrix(mat)
  ...
end subroutine
#endif

!! active non-differentiable
function coo_to_csr(i,j,ja,ia) result(kij)
#ifndef STRIP_AD
  ...
#else
  kij = 0 ! retain a trivial dependency
#endif
end function

!! active differentiable
subroutine matrix_setup(a_ij,phi,ja,ia,res)
  ...
#ifndef STRIP
  if(use_advanced_feature) call adv_feature()
#endif
end subroutine
end module
```

Differentiation problems

- We use the source-transformation tool Tapenade that can handle F90.
- There has been significant progress with Tapenade, but the user cannot expect the AD tool to run as robustly as the compiler.
- In the current version 3.6 we need to work around the following issues:
    1. using `save` as an attribute of a module scope causes incomplete differentiation,
    2. the `case default` must be the last case,
    3. the differentiation of statements involving pointers is under development.
    4. differentiation of modules creates a complete copy, which causes problems with `use:only` statements.

Post-processing of AD'ed code

1. inherit original modules into their generated derivative modules,
2. purge generated code of all definitions of primal equivalent routines and data (except private data),
3. ensure that all references to primal routines and data refer to original code and not to equivalent generated code,
4. identify and remove generated derivative type definitions and replace associated declarations using that type with the original type,
5. for the adjoint of linear system solvers, use the hand coded alternative (this must be properly converged at each invocation),
6. in adjoint code, identify fixed-point iterations, reconfigure it to record once and restore once active primal variables,
7. in adjoint code, identify active quantities which can be assumed to behave like constants and remove their associated adjoint computation.

Original code

```
module pdes_m
use base_m

character(3), &
  private::fmt="csr"

type::pde_t
  real,dimension(:), &
    allocatable::phi,rhs
  real::reduc
  integer::max_iter
end type

type(pde_t)::pres, vel

contains

subroutine navier_stokes()
  ...
end subroutine
end module
```

Generated code (Tapenade)

```
module pdes_m__b
use base_m__b

character(3),private::fmt="csr"

type::pde_t
  real,dimension(:), &
    allocatable::phi,rhs
  real::reduc
  integer::max_iter
end type

type::pde_t__b
  real,dimension(:), &
    allocatable::phi,rhs
end type

type(pde_t)::pres, vel
type(pde_t__b)::pres__b, vel__b

contains

subroutine navier_stokes_c__b()
  ...
end subroutine

subroutine navier_stokes__b()
  call gradient_c__b()
  call setup_mat_rhs_c__b()
  call solve_c__b()

  call solve__b()
  call setup__b()
  call gradient__b()
end subroutine
end module
```

Modified (in-place) generated code

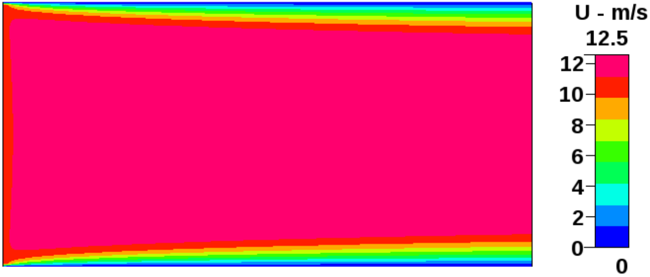
```
module pdes_m__b
use pdes_m ! import original
use base_m__b

character(3),private::fmt="csr"

type(pde_t)::pres__b, vel__b

contains

subroutine navier_stokes__b()
  call gradient()
  call setup_mat_rhs()

! A.x = b:
! not necessary since a
! fixed point is assumed
! to have been reached
!   call solve_c__b()


! adjoint of A.x = b:
! manual implementation
  call solve__rev()

  call setup__b()
  call gradient__b()
end subroutine
end module
```

ACE+ differentiation

- Reduced Kernel Code Size: 680 files, 230K lines / 8MB of source code.
- Using tapenade Version 3.6 (September 2011).
- Memory: 500-900MB RAM, 2.0-2.5GB VM.
- CPU Time: 10 minutes on Intel i5 Equivalent.

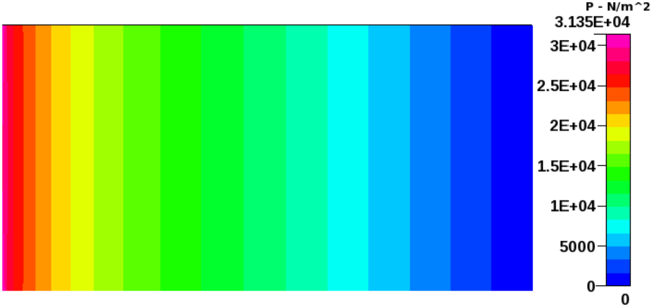
tangent-linear ACE+, laminar channel

Rectangular channel, with aspect ratio of 20, and with 10m/s inlet on the left, fixed-pressure outlet on the right, and no-slip walls on the top and bottom.
Sensitivity of velocity integral wrt perturbation in uniform inlet velocity.



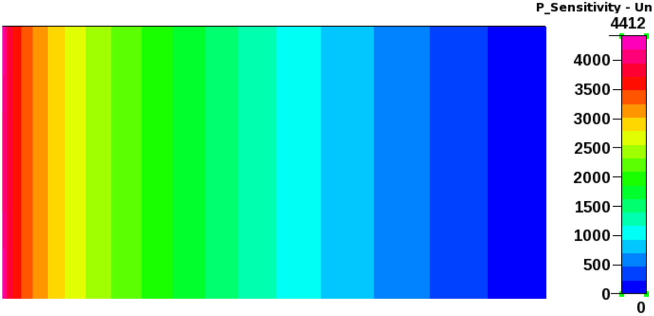
u-velocities, domain scaled by 1/10 in the x-direction

tangent-linear ACE+, laminar channel



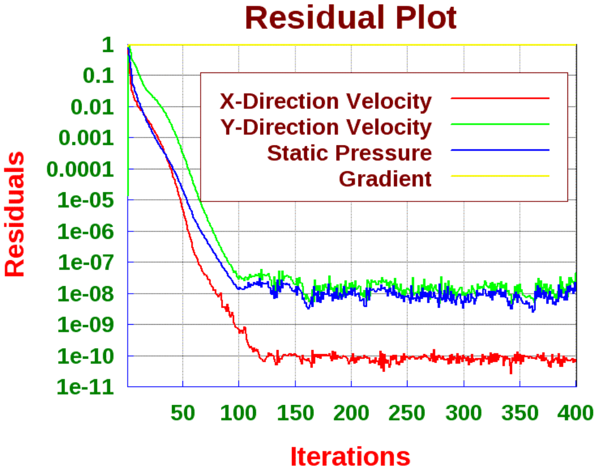
pressure field

tangent-linear ACE+, laminar channel results



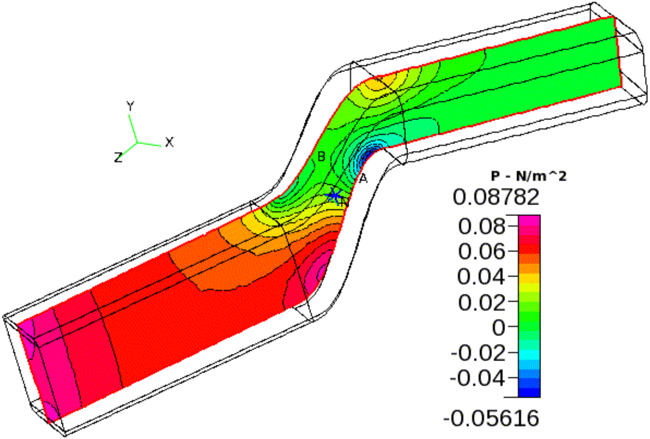
pressure sensitivity to inlet velocity perturbation

tangent-linear ACE+, laminar channel results



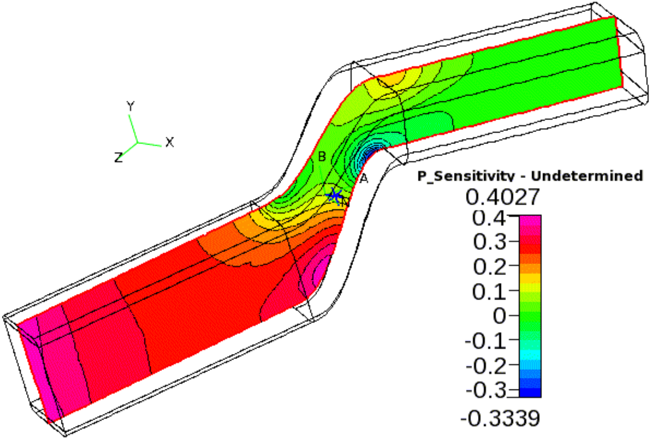
pressure field

tangent-linear ACE+, S-Bend testcase



S-Bend testcase, tangent-linear discrete solution, pressure

tangent-linear ACE+, S-Bend testcase



Pressure sensitivity to inlet variation

Contents

Introduction to Algorithmic Differentiation

Development of incompressible adjoint solvers

AD on industrial CFD codes

Summary

Summary

- A complete methodology has been devised to prepare, differentiate and post-process source code which is largely automated.
- Fortran 90/95 is well supported by source transformation AD tools, large industrial codes can be tackled.
- Application to steady simulations is available for industrial beta evaluation within 6 months.

Summary

- A complete methodology has been devised to prepare, differentiate and post-process source code which is largely automated.
- Fortran 90/95 is well supported by source transformation AD tools, large industrial codes can be tackled.
- Application to steady simulations is available for industrial beta evaluation within 6 months.

Summary

- A complete methodology has been devised to prepare, differentiate and post-process source code which is largely automated.
- Fortran 90/95 is well supported by source transformation AD tools, large industrial codes can be tackled.
- Application to steady simulations is available for industrial beta evaluation within 6 months.

Acknowledgements

FLOWHEAD
Fluid Optimisation Workflows for Highly Effective Automotive Development Processes

This research is part of the European project FlowHead funded by the European Commission under THEME SST.2007-RTD-1.
`http://flowhead.sems.qmul.ac.uk`

Queen Mary
University of London

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche SOPHIA ANTIPOLIS - MÉDITERRANÉE

SEVENTH FRAMEWORK PROGRAMME