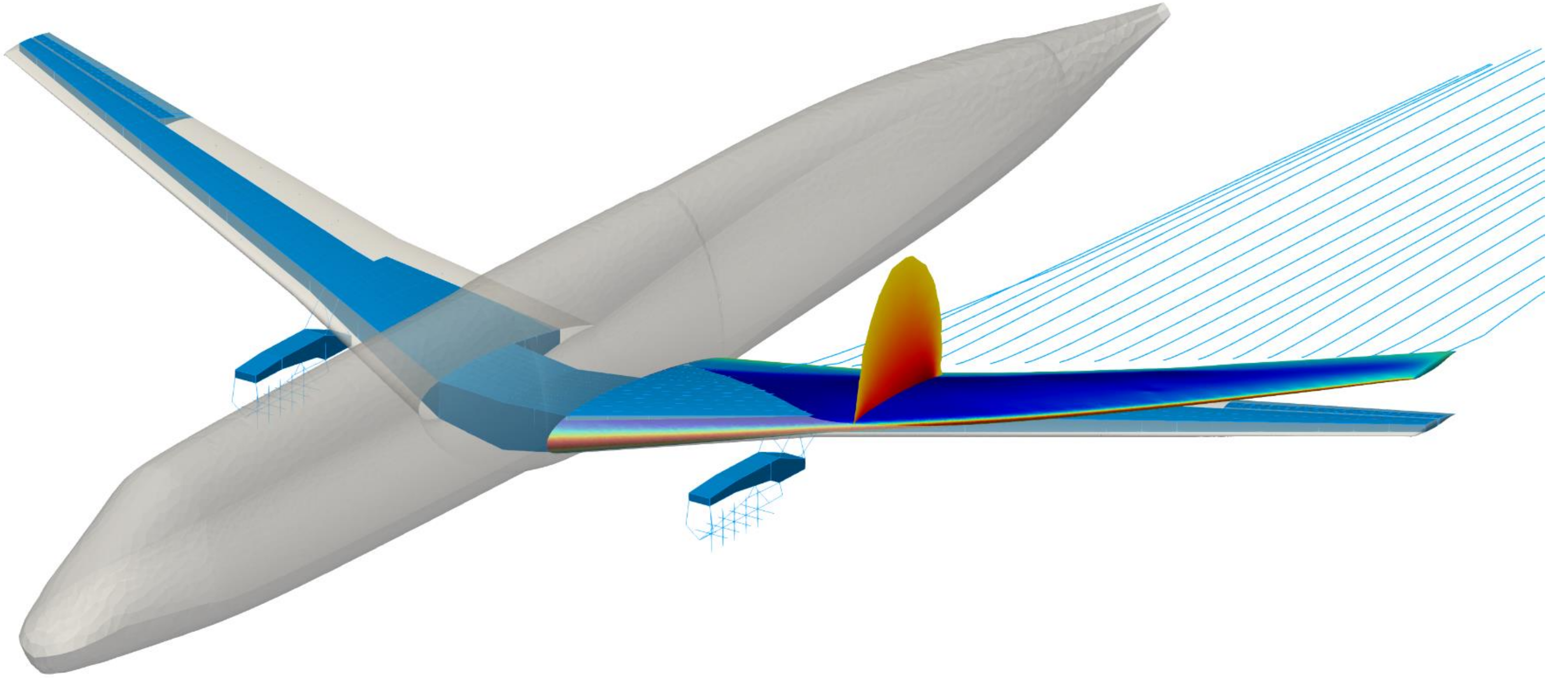
Solution of optimization problems using adjoint automatic differentiation

Adrien Crovato



Liège, August 2023

Optimization problems

General formulation

Minimize **objective function**

with respect to **design variables**

subject to **constraints**

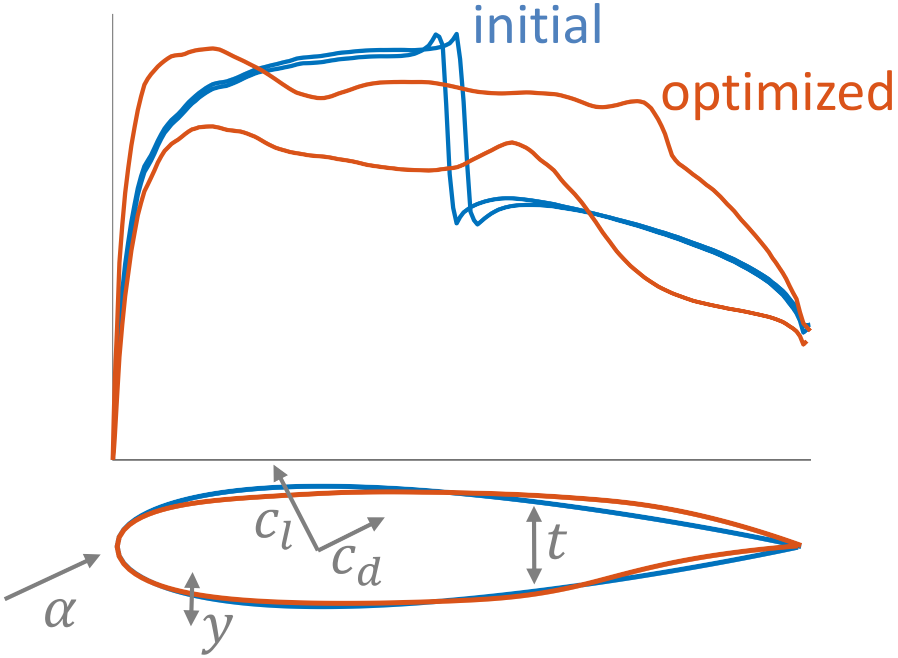
Aerodynamic shape optimization

$$\min_{y,\alpha} c_d$$

$$\text{s.t. } c_l = c_l^\star, \quad t = t^\star$$

Can be solved using:

- **Gradient-based** approach
- Gradient-free approach



Gradient-based optimization

General formulation

$$\min_{\boldsymbol{x}} \quad F(\boldsymbol{u};\ \boldsymbol{x})$$

$$\text{s.t.} \quad \begin{aligned} \boldsymbol{R}(\boldsymbol{u};\boldsymbol{x}) &= 0 \\ \boldsymbol{C}_{\mathrm{E}}(\boldsymbol{u};\boldsymbol{x}) &= 0 \\ \boldsymbol{C}_{\mathrm{I}}(\boldsymbol{u};\boldsymbol{x}) &\geq 0 \end{aligned}$$

F: objective function
$\boldsymbol{u}$: physical variables
$\boldsymbol{x}$: design variables
$\boldsymbol{R}$: residual equations
$\boldsymbol{C}_{\mathrm{E}}$: equality constraints
$\boldsymbol{C}_{\mathrm{I}}$: inequality constraints

Gradient-based approach

$$d_{\boldsymbol{x}} F(\boldsymbol{u};\ \boldsymbol{x}) \rightarrow 0$$

$$\text{s.t.} \quad \begin{aligned} \boldsymbol{R}(\boldsymbol{u};\boldsymbol{x}) &= 0 \\ \boldsymbol{C}_{\mathrm{E}}(\boldsymbol{u};\boldsymbol{x}) &= 0 \\ \boldsymbol{C}_{\mathrm{I}}(\boldsymbol{u};\boldsymbol{x}) &\geq 0 \end{aligned}$$

Need to:
- **Formulate** total gradient
- **Compute** any gradients

The **adjoint** method and the **automatic differentiation** technique are one way of **formulating** and **computing** the gradients

Outline

Theory

- Formulation of the gradients
- Computation of the gradients

Optimization of coupled physics problem

- Description and formulation
- Methodology and cases
- The sellar problem

Implementation details

- DART
- SDPM

Formulation of the gradients

“perturbation”

$$\begin{cases} R\big(u(x+\delta x)\big) = 0 \\ d_x F = \Delta\left\{\dfrac{F\big(u(x+\delta x)\big)}{\delta x}\right\} \end{cases}$$

$$d_x F(u;x) \to 0$$
$$R(u;x) = 0$$

“chain rule”

$$\begin{cases} R\big(u(x)\big) = 0 \\ d_x F = \partial_x F - \partial_u F \partial_u R^{-1} \partial_x R \end{cases}$$

Methods based on perturbation

Finite differences

$$\begin{cases} R\big(u(x)\big) = 0 \\ R\big(u^+(x+\delta x)\big) = 0 \\ d_x F = \dfrac{F(u^+) - F(u)}{\delta x} + O(\delta x) \end{cases}$$

Complex step

$$\begin{cases} R\big(u(x)\big) = 0 \\ R\big(u^+(x+i\delta x)\big) = 0 \\ d_x F = \operatorname{Im}\left\{\dfrac{F(u^+)}{\delta x}\right\} + O(\delta x^2) \end{cases}$$

Cost

Solve equations: $n_x \times n_s \times t_s$

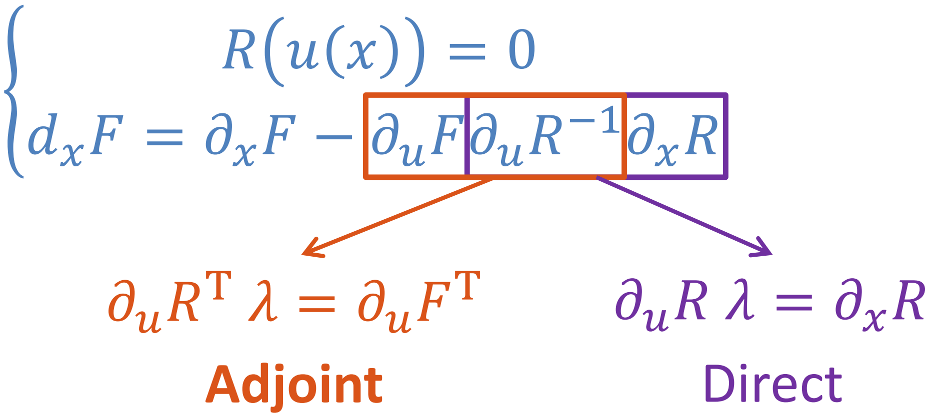
Evaluate gradients: $n_x \times n_f \times t_f$

Total: $n_x \times \left(n_s \times t_s + n_f \times t_f\right)$

n_x: n.o. design variables
n_s: n.o. nonlinear iterations
n_f: n.o. functionals
t_s: time to solve linear equations
t_f: time to compute functional

Methods based on chain rule

Direct and adjoint



$$\begin{cases} R\big(u(x)\big) = 0 \\ d_x F = \partial_x F - \partial_u F \, \partial_u R^{-1} \, \partial_x R \end{cases}$$

$$\partial_u R^{\mathrm{T}} \lambda = \partial_u F^{\mathrm{T}}$$

Adjoint

$$\partial_u R \, \lambda = \partial_x R$$

Direct

Cost (adjoint)

Solve adjoint: $n_f \times t_s$

Evaluate gradients: $(n_u + n_x) \times \left(n_f \times t_f + t_r\right)$

Total: $\left((n_u + n_x) \times \left(n_f \times t_f + t_r\right) + n_f \times t_s\right)$

n_x: n.o. design variables
n_u: n.o. variables
n_s: n.o. nonlinear iterations
n_f: n.o. functionals
t_s: time to solve linear equations
t_f: time to compute functional
t_f: time to compute residuals

Nearly independent on number of design variables

Computation of the gradients

Hand differentiation

✓ Most effective

× Difficult, sometimes not feasible

Finite differences

✓ Very easy

× Inaccurate

Complex step

✓ Accurate

× Complex arithmetic

Automatic differentiation

✓ Straightforward

× Increased memory usage

Automatic differentiation – implementation

Source code transformation

```
double x = 1;
double y = sin(x) * cos(x);
```

```
double x = 1;
double s = sin(x);
double c = cos(x);
double ds = cos(x);
double dc = -sin(x);
double dy = ds * c + s * dc;
```

Inria TAPENADE

Operator overloading

```
ADdouble x = 1;
x.setGradient(1);
ADdouble y = sin(x) * cos(x);
double dy = y.getGradient();
```

Scientific Computing
University of Kaiserslautern-Landau

CoDiPack

Automatic differentiation – accumulation

Consider

$$y = f(x) = g\big(h(x)\big)$$

$$\begin{aligned} w_0 &= x \\ w_1 &= h(w_0) \\ w_2 &= g(w_1) = y \end{aligned}$$

Chain rule yields

$$\frac{dy}{dx} = \frac{dg}{dh}\frac{dh}{dx} = \frac{dy}{dw_2}\frac{dw_2}{dw_1}\frac{dw_1}{dw_0}\frac{dw_0}{dx}$$

Forward (tangent) mode

$$\dot{y} = \frac{df}{dx}\dot{x}$$

$$\frac{dy}{dx} = \frac{dy}{dw_2}\left(\frac{dw_2}{dw_1}\left(\frac{dw_1}{dw_0}\frac{dw_0}{dx}\right)\right)$$

Reverse (adjoint) mode

$$\bar{x} = \frac{df}{dx}^T\bar{y}$$

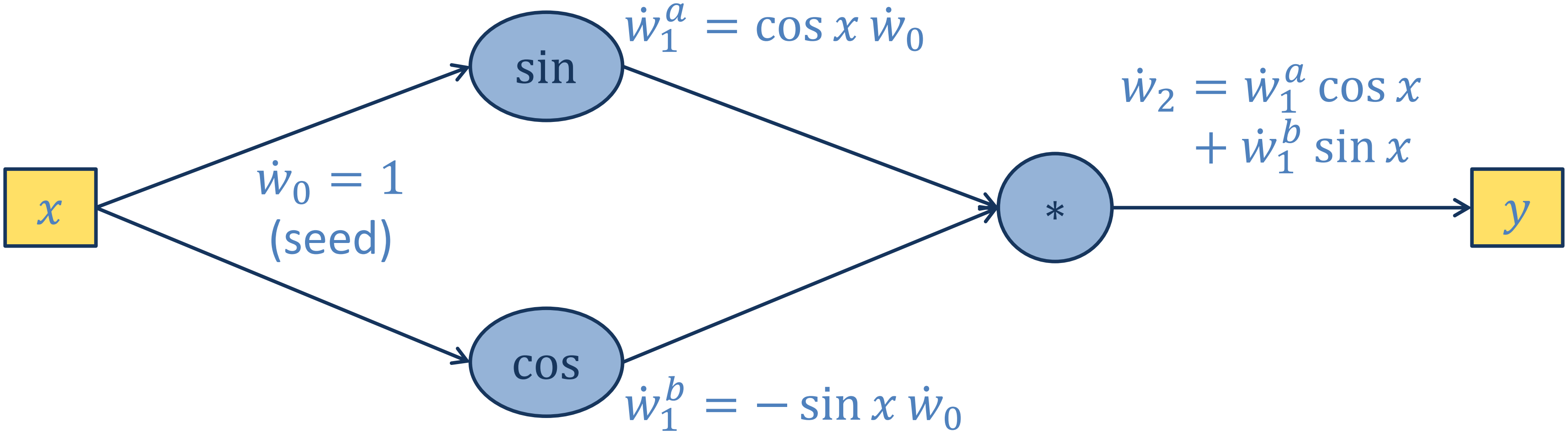
$$\frac{dy}{dx} = \left(\left(\frac{dy}{dw_2}\frac{dw_2}{dw_1}\right)\frac{dw_1}{dw_0}\right)\frac{dw_0}{dx}$$

Automatic differentiation – forward mode

Forward (tangent) mode

$$y = \sin x \cos x$$

$$\dot{y} = \frac{df}{dx}\dot{x}$$



$$\dot{w}_1^a = \cos x \, \dot{w}_0$$

$$\dot{w}_0 = 1$$

(seed)

$$\dot{w}_2 = \dot{w}_1^a \cos x + \dot{w}_1^b \sin x$$

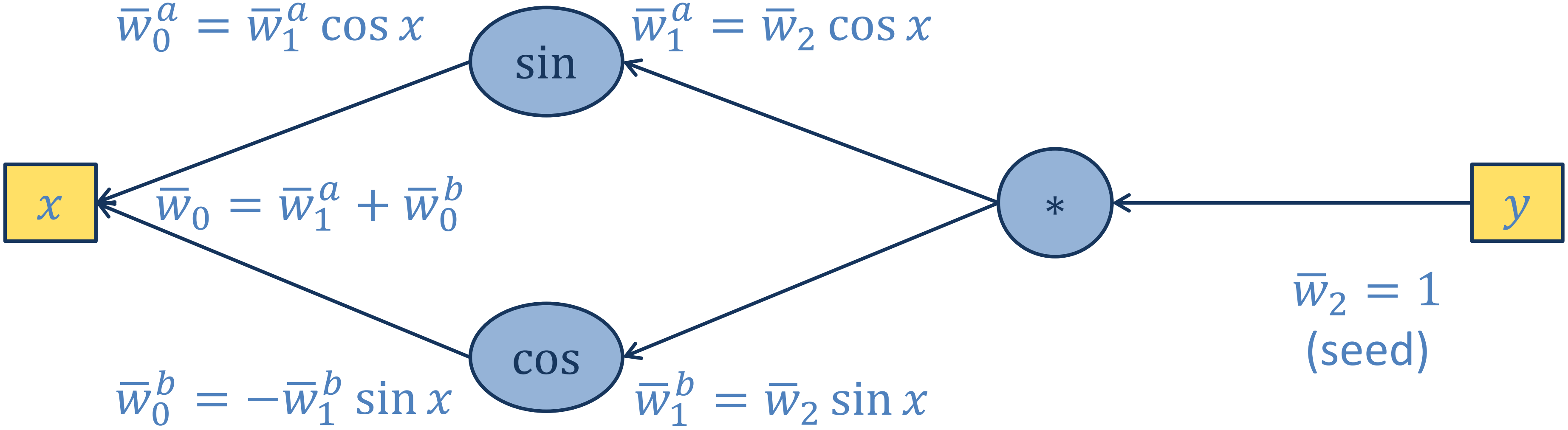
$$\dot{w}_1^b = -\sin x \, \dot{w}_0$$

Automatic differentiation – reverse mode

Forward (tangent) mode

$y = \sin x \cos x$

$\bar{x} = \frac{df}{dx}^T \bar{y}$



Automatic differentiation – modes

Forward mode

```
ADdouble x = 1;
x.setGradient(1);
ADdouble y = sin(x) * cos(x);
double dy = y.getGradient();
```

One pass to **compute value** and **derivative** with respect to **one input**

Reverse mode

```
ADdouble x = 1;
Tape tape;
tape.setActive();
tape.registerInput(x);
ADdouble y = sin(x) * cos(x);
tape.registerOutput(y);
tape.setPassive();
y.setGradient(1);
tape.evaluate();
double dx = x.getGradient();
```

One pass to **compute value** and **cache intermediate** results (**tape**), and a **second pass** to **compute derivatives** of **one output**

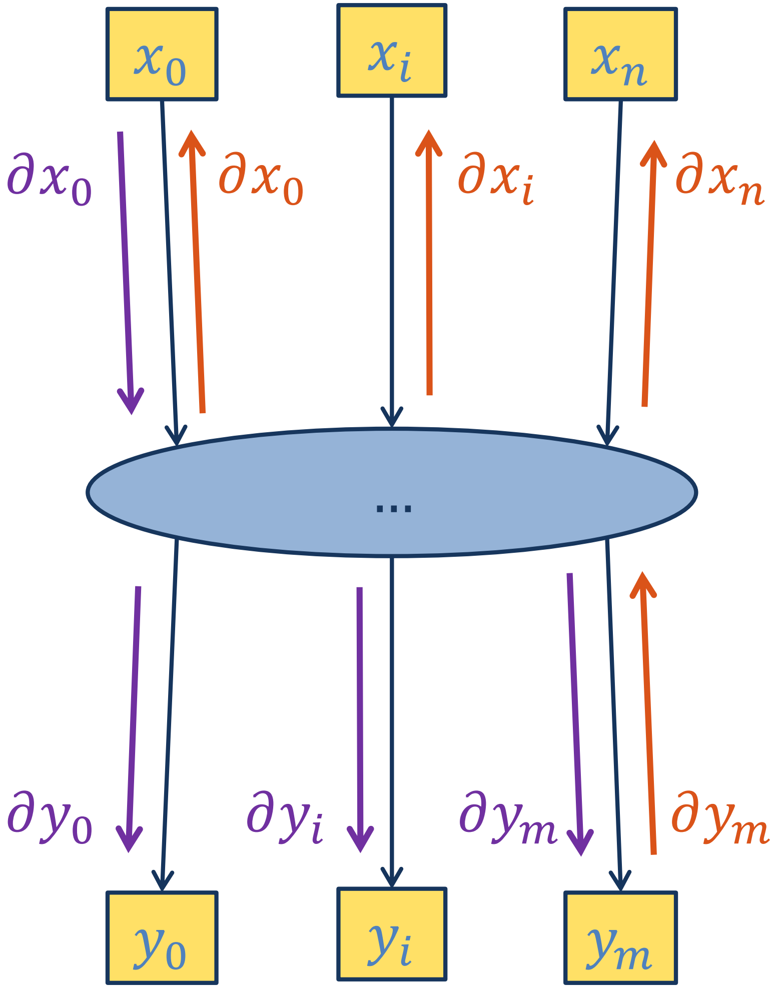
Automatic differentiation – “best” mode

Forward mode

n inputs

m outputs

$\boldsymbol{n \ll m}$



Reverse mode

n inputs

m outputs

$\boldsymbol{n \gg m}$

Outline

Theory

- Formulation of the gradients
- Computation of the gradients

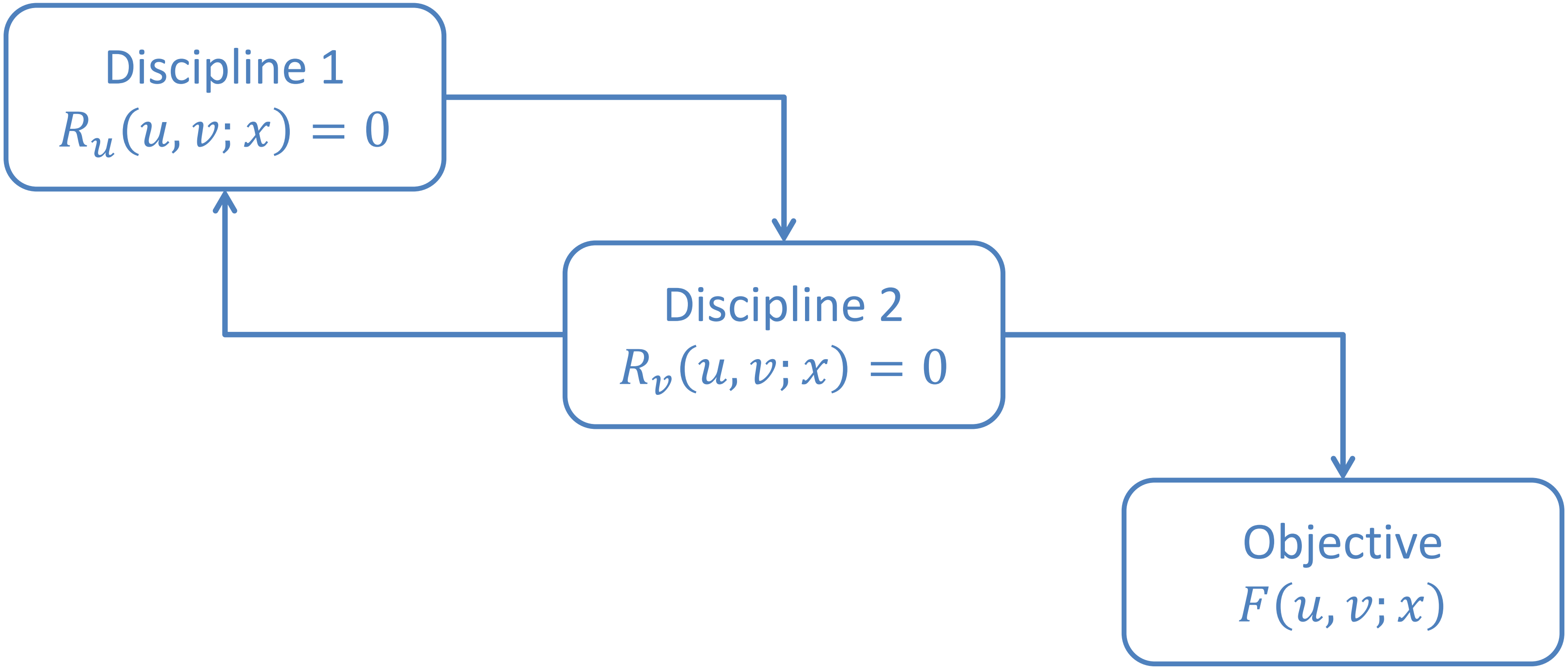
Optimization of coupled physics problem

- Description and formulation
- Methodology and cases
- The sellar problem

Implementation details

- DART
- SDPM

Coupled optimization – description



Discipline 1
$R_u(u, v; x) = 0$

Discipline 2
$R_v(u, v; x) = 0$

Objective
$F(u, v; x)$

Mathematical formulation

$$\min_x F\,(u, v; x)$$

$$\text{s.t.}\quad \begin{aligned} R_u(u, v; x) &= 0 \\ R_v(u, v; x) &= 0 \end{aligned}$$

Coupled optimization – adjoint formulation

Augmented Lagrangian

$$\mathcal{L} = F + \lambda_u R_u + \lambda_v R_v$$

$$\delta\mathcal{L} = 0 \Leftrightarrow \begin{cases} \partial_u F + \lambda_u \partial_u R_u + \lambda_v \partial_u R_v = 0 \\ \partial_v F + \lambda_u \partial_v R_u + \lambda_v \partial_v R_v = 0 \\ \partial_x F + \lambda_u \partial_x R_u + \lambda_v \partial_x R_v = 0 \\ R_u = 0 \\ R_v = 0 \end{cases}$$

Linear algebra

$$\begin{aligned} d_x F &= \partial_x F \\ &\quad + \partial_u F \partial_x u + \partial_v F \partial_x v \\ &= \partial_x F \\ &\quad + \partial_u F \left(\partial_{R_u} u \partial_x R_u + \partial_{R_v} u \partial_x R_v \right) + \partial_v F (\partial_{R_u} v \partial_x R_u + \partial_{R_v} v \partial_x R_v) \\ &= \partial_x F \\ &\quad + (\partial_u F \partial_u R_u^{-1} + \partial_v F \partial_v R_u^{-1}) \partial_x R_u + (\partial_u F \partial_u R_v^{-1} + \partial_v F \partial_v R_v^{-1}) \partial_x R_v \end{aligned}$$

Coupled optimization – methodology

Solve adjoint

$$\begin{bmatrix} \partial_v R_v^T & \partial_v R_u^T \\ \partial_u R_v^T & \partial_u R_u^T \end{bmatrix} \begin{bmatrix} \lambda_v \\ \lambda_u \end{bmatrix} = -\begin{bmatrix} \partial_v F^T \\ \partial_u F^T \end{bmatrix}$$

Compute total gradient

$$d_x F^T = \partial_x F^T + \partial_x R_u^T \lambda_u + \partial_x R_v^T \lambda_v$$

Main cases

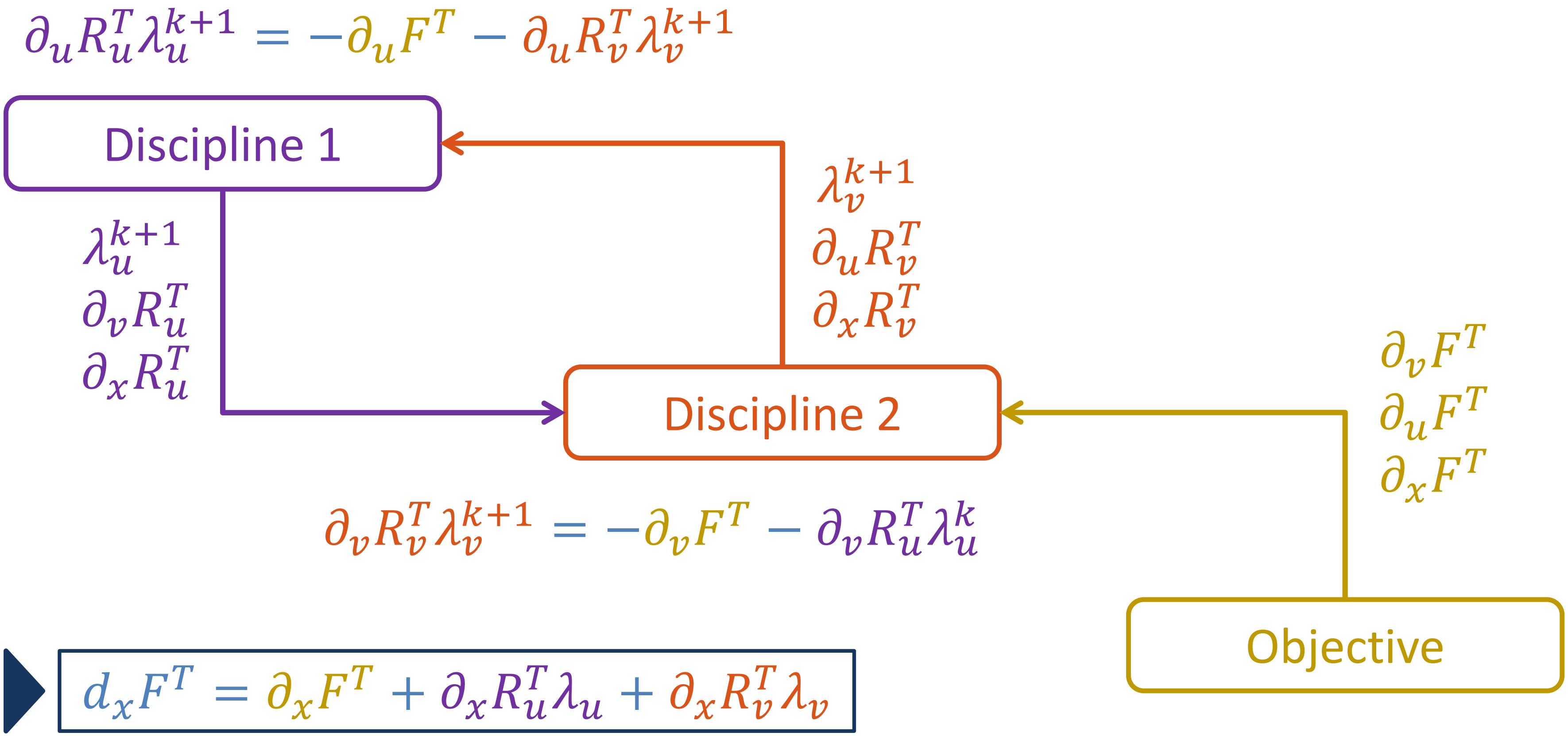
A) **Partial gradients** are **available** and **matrices** are **small** enough

B) **Partial gradients** are **available** but **matrices** are too **large**

C) **Partial gradients** are **not available**

Coupled optimization – case B

Gradients are available but matrices are too large to fit in memory. Solution is computed **iteratively**, e.g. using a **BGS** approach.



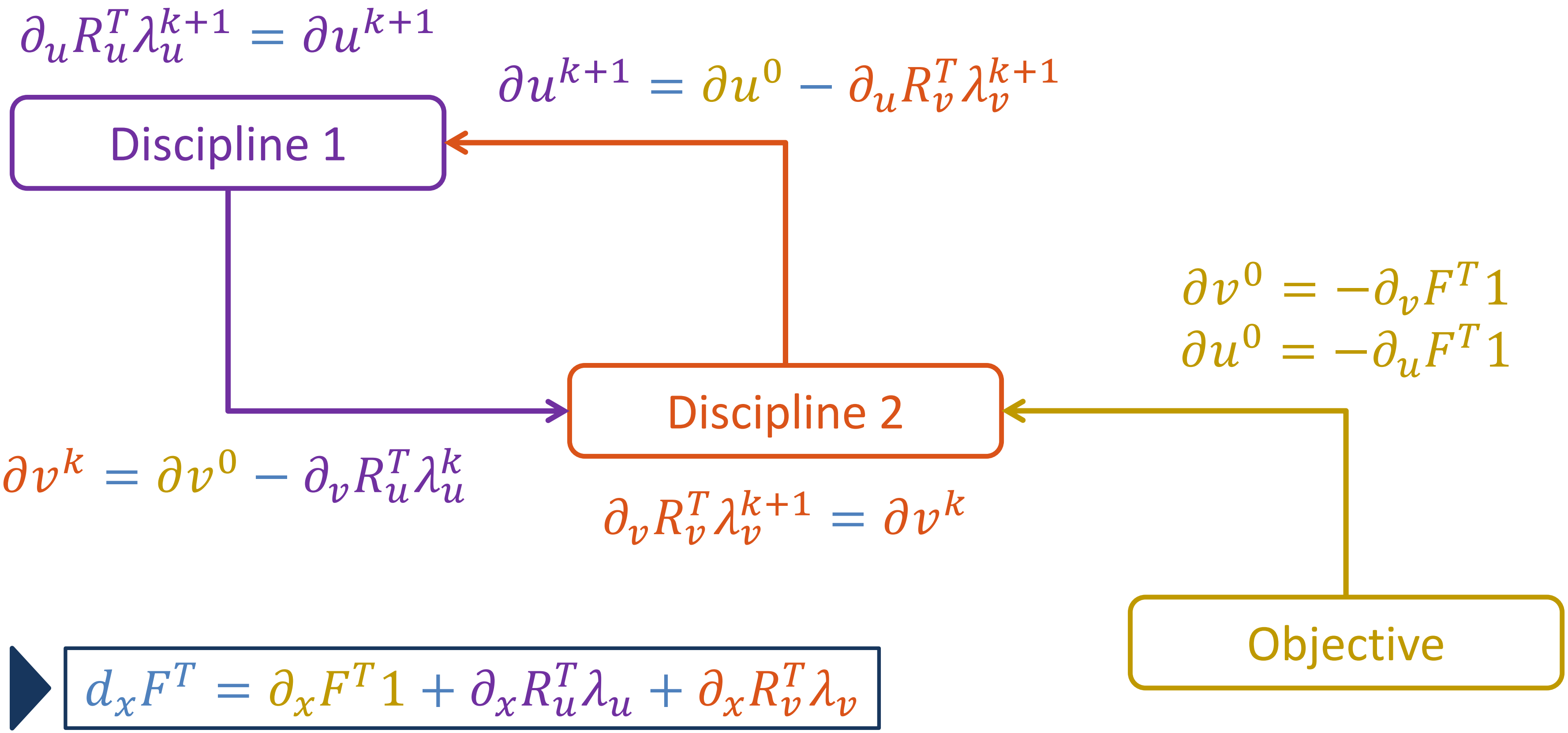
$$\partial_u R_u^T \lambda_u^{k+1} = -\partial_u F^T - \partial_u R_v^T \lambda_v^{k+1}$$

$$\partial_v R_v^T \lambda_v^{k+1} = -\partial_v F^T - \partial_v R_u^T \lambda_u^k$$

$$d_x F^T = \partial_x F^T + \partial_x R_u^T \lambda_u + \partial_x R_v^T \lambda_v$$

Coupled optimization – case C

Gradients are not available.
Solution is computed **iteratively**, e.g. using a **BGS** approach.
Each **contribution** is **added individually** using matrix-vector product.



$\partial_u R_u^T \lambda_u^{k+1} = \partial u^{k+1}$

$\partial u^{k+1} = \partial u^0 - \partial_u R_v^T \lambda_v^{k+1}$

Discipline 1

$\partial v^0 = -\partial_v F^T 1$

$\partial u^0 = -\partial_u F^T 1$

Discipline 2

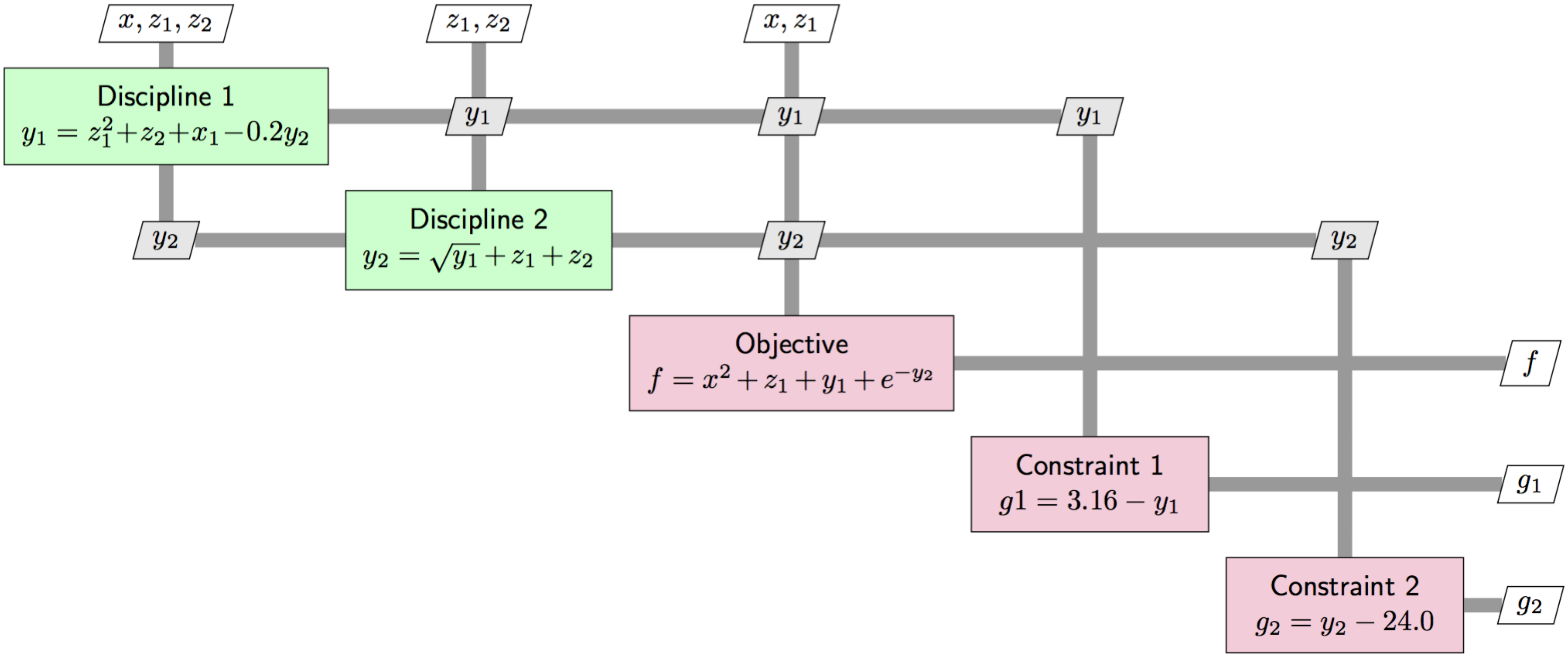
$\partial v^k = \partial v^0 - \partial_v R_u^T \lambda_u^k$

$\partial_v R_v^T \lambda_v^{k+1} = \partial v^k$

Objective

$$\boxed{d_x F^T = \partial_x F^T 1 + \partial_x R_u^T \lambda_u + \partial_x R_v^T \lambda_v}$$

The sellar problem



x, z_1, z_2

z_1, z_2

x, z_1

Discipline 1
$y_1 = z_1^2 + z_2 + x_1 - 0.2 y_2$

y_1

y_1

y_1

y_2

Discipline 2
$y_2 = \sqrt{y_1} + z_1 + z_2$

y_2

y_2

Objective
$f = x^2 + z_1 + y_1 + e^{-y_2}$

f

Constraint 1
$g1 = 3.16 - y_1$

g_1

Constraint 2
$g_2 = y_2 - 24.0$

g_2

https://openmdao.org

Outline

Theory

- Formulation of the gradients
- Computation of the gradients

Optimization of coupled physics problem

- Description and formulation
- Methodology and cases
- The sellar problem

Implementation details

- DART
- SDPM

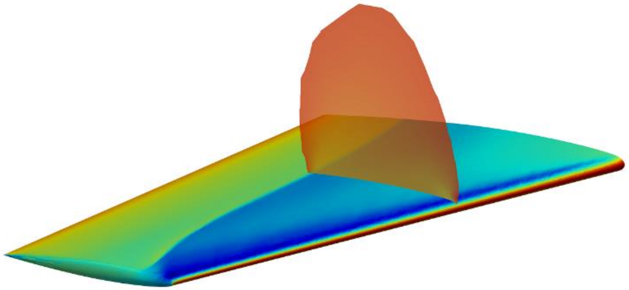
Implementation details

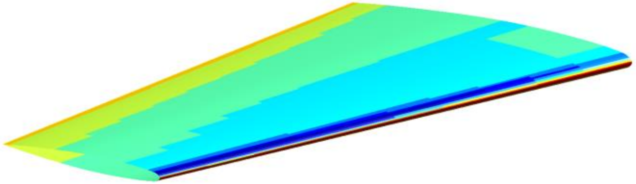
DART

- Steady full potential formulation
- Finite element discretization
- Unstructured tetrahedral grid
- Analytical discrete adjoint
- Mesh morphing
- C++ with python API

SDPM

- Unsteady potential formulation
- Panel discretization
- Unstructured quadrangular grid
- Reverse automatic differentiation
- C++ with python API





DART implementation

Mesh residuals

$R_x(x_s) = 0$

Potential residuals →

$R_\phi(x, \phi, \alpha) = 0$

Loads functional

$[F_x, F_y, F_z](x, \phi, \alpha)$

Coefficients functional

$[C_L, C_D](x, \phi, \alpha)$

```
// $R_\phi = \int_V \rho \nabla\phi \cdot \nabla\psi \, dV - \int_S \rho \nabla\phi \cdot n \, \psi \, dS$
Vector PotentialResidual::build()

// $\partial_x R_\phi = \partial_x \int_V \rho \nabla\phi \cdot \nabla\psi \, dV - \partial_x \int_S \rho \nabla\phi \cdot n \, \psi \, dS$
Matrix PotentialResidual::buildGradientMesh()

// $\partial_\phi R_\phi = \partial_\phi \int_V \rho \nabla\phi \cdot \nabla\psi \, dV - \int_S \rho \nabla\phi \cdot n \, \psi \, dS$
Matrix PotentialResidual::buildGradientFlow()

// $\partial_\alpha R_\phi = \int_V \rho \nabla\phi \cdot \nabla\psi \, dV - \partial_\alpha \int_S \rho \nabla\phi \cdot n \, \psi \, dS$
Vector PotentialResidual::buildGradientAoA()
```

```
# $\partial x = \partial_x R_\phi^T \, \partial R_\phi$
d_in['xv'] += computeFlowMesh(d_res['phi'])
# $\partial \phi = \partial_\phi R_\phi^T \, \partial R_\phi$
d_out['phi'] += computeFlowFlow(d_res['phi'])
# $\partial \alpha = \partial_\alpha R_\phi^T \, \partial R_\phi$
d_in['aoa'] += computeFlowAoa(d_res['phi'])
```

SDPM implementation

Loads functional

$[F_x, F_y, F_z](x, \alpha, \omega)$

Coefficients functional

$[C_L, C_D](x, \alpha, \omega)$

```
// $F_{[x,y,z]}(x,\alpha,\omega), C_{[L,D]}(x,\alpha,\omega)$
void Adjoint::solve() {
  tape.registerInput(aoa);
  solver.run();
  tape.registerOutput(cl); }

// $\partial_{[x,\alpha,\omega]} C_L$
Map Adjoint::compute(dOut) {
  cl.setGradient(dOut);
  tape.evaluate();
  dIn[“aoa”] = aoa.getGradient(); }
```

```
d_x_a_o = sdpm.adjoint.compute(d_out[‘cl'])

d_in['x'] += d_x_a_o[‘x’] # $\partial x = \partial_x C_L^T \, \partial C_L$
d_in[‘aoa'] += d_x_a_o[‘aoa’] # $\partial \alpha = \partial_\alpha C_L^T \, \partial C_L$
d_in[‘omega'] += d_x_a_o[‘om’] # $\partial \omega = \partial_\omega C_L^T \, \partial C_L$
```

Conclusion

Main points

- The **adjoint method** is a **mathematical** method that **formulates** the **total gradient of a functional with respect to any variables** as a function of **partial gradients of intermediate quantities.**

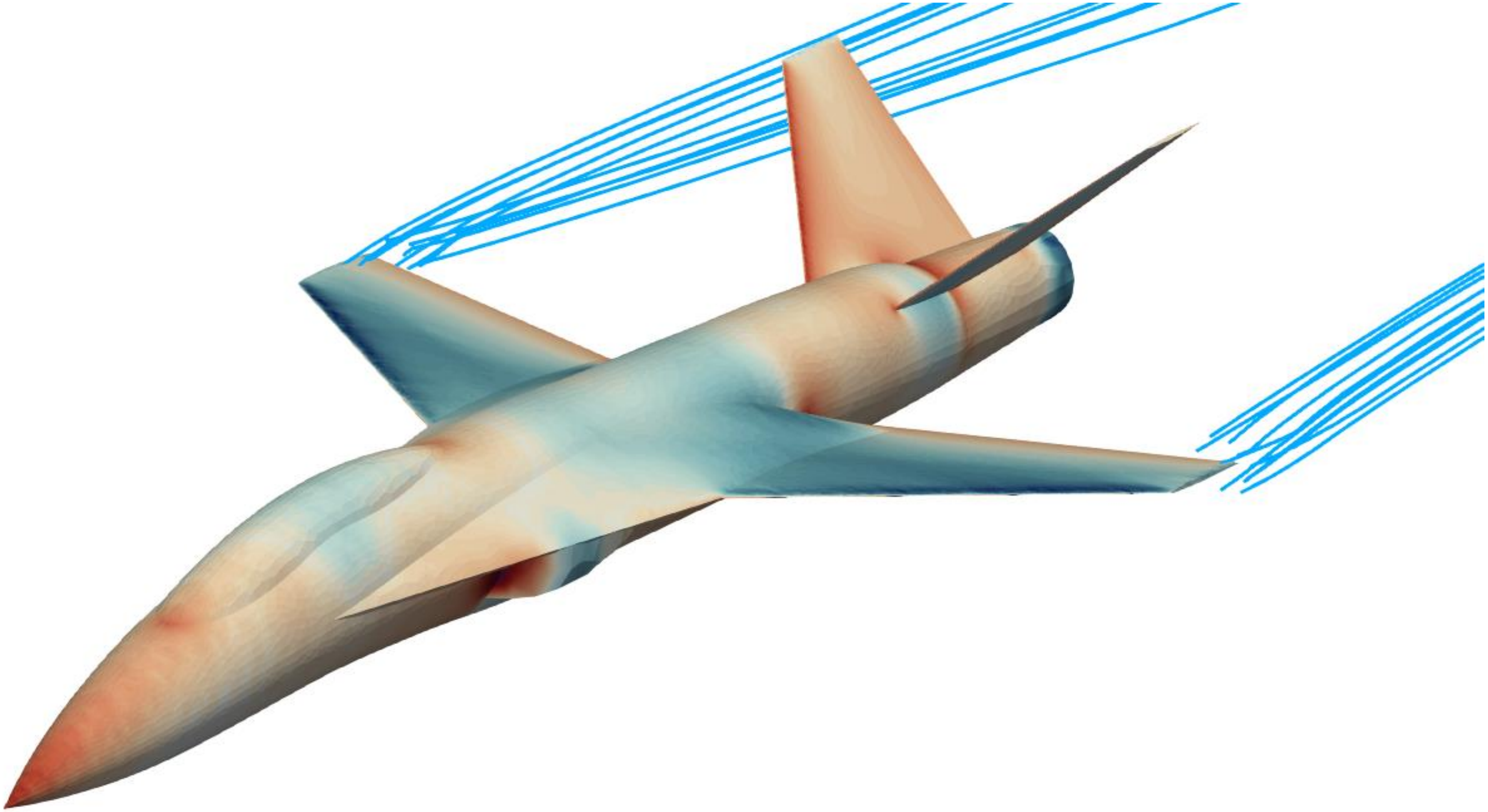
- **Automatic differentiation** is a **numerical** technique that **computes** the **gradient of a variable with respect to another variable** solely based on the **source code** of a computer program. **AD** can operate in **reverse accumulation** mode, which corresponds to the **adjoint** formulation.

- **Optimization of coupled physics problems** often involve **large systems** that need to be **solved iteratively**, for which the **automatic differentiation** method is **well suited**. If the number of **design variables** is **larger** than the number of **functional**, the **adjoint** method and **reverse accumulation** should be preferred.

Group meeting

Adjoint automatic differentiation

Adrien Crovato – Liège, August 2023



https://acrovato.github.io