Jens-Dominik Müller
Queen Mary Univ. London

collaborators
Kumar Mohanamuraly (CERFA
Orest Mykhaskiv (QMUL)
Shenren Xu (NWPU)
Andrea Walther (UPB)
Mladen Banovic (UPB)
Salvatore Auriemma (OCCT)

Shape optimisation using AD of complete CFD workflows including CAD geometry

IPGP, Paris, 24 January 2019 7 December 2018

How to optimise with CFD?

- Stochastic or meta-modelling methods can be used with standard simulation tools such as CFD, CAD to find optima.
- The computational cost is of the order of 100s or 1000s of evaluations of the flow, depending on the size of number of design variables.
- Gradient-based methods are much more effective in finding the optimum, as only a 1-D path through the design space needs to be traced.
- ”One-shot” methods can find the optimum in 5 times the number of the cost of a function evaluation (CFD and adjoint run).
- **But we need gradients for all components of the computational chain.**

How to optimise with CFD?

- Stochastic or meta-modelling methods can be used with standard simulation tools such as CFD, CAD to find optima.
- The computational cost is of the order of 100s or 1000s of evaluations of the flow, depending on the size of number of design variables.
- Gradient-based methods are much more effective in finding the optimum, as only a 1-D path through the design space needs to be traced.
- ”One-shot” methods can find the optimum in 5 times the number of the cost of a function evaluation (CFD and adjoint run).
- **But we need gradients for all components of the computational chain.**

How to optimise with CFD?

- Stochastic or meta-modelling methods can be used with standard simulation tools such as CFD, CAD to find optima.
- The computational cost is of the order of 100s or 1000s of evaluations of the flow, depending on the size of number of design variables.
- Gradient-based methods are much more effective in finding the optimum, as only a 1-D path through the design space needs to be traced.
- ”One-shot” methods can find the optimum in 5 times the number of the cost of a function evaluation (CFD and adjoint run).
- **But we need gradients for all components of the computational chain.**

How to optimise with CFD?

- Stochastic or meta-modelling methods can be used with standard simulation tools such as CFD, CAD to find optima.
- The computational cost is of the order of 100s or 1000s of evaluations of the flow, depending on the size of number of design variables.
- Gradient-based methods are much more effective in finding the optimum, as only a 1-D path through the design space needs to be traced.
- ”One-shot” methods can find the optimum in 5 times the number of the cost of a function evaluation (CFD and adjoint run).
- **But we need gradients for all components of the computational chain.**

How to optimise with CFD?

- Stochastic or meta-modelling methods can be used with standard simulation tools such as CFD, CAD to find optima.
- The computational cost is of the order of 100s or 1000s of evaluations of the flow, depending on the size of number of design variables.
- Gradient-based methods are much more effective in finding the optimum, as only a 1-D path through the design space needs to be traced.
- ”One-shot” methods can find the optimum in 5 times the number of the cost of a function evaluation (CFD and adjoint run).
- **But we need gradients for all components of the computational chain.**

Simulation with Optimisation

We have

- Gradient-based methods, that are very efficient, but limited in maturity and only for CFD.
- A wide range of parametrisations, but not a mature and efficient optimisation workflow with CAD in the loop.
- Multi-disciplinary optimisation using black-box methods without gradients.

We want

- gradient methods that offer sensitivities for multi-disciplinary problems,
- automatic and multi-level parametrisations with sensitivities and CAD,
- to be able to quantify uncertainties and include the in robust optimisation.

Simulation with Optimisation

We have

- Gradient-based methods, that are very efficient, but limited in maturity and only for CFD.
- A wide range of parametrisations, but not a mature and efficient optimisation workflow with CAD in the loop.
- Multi-disciplinary optimisation using black-box methods without gradients.

We want

- gradient methods that offer sensitivities for multi-disciplinary problems,
- automatic and multi-level parametrisations with sensitivities and CAD,
- to be able to quantify uncertainties and include the in robust optimisation.

Other uses of sensitivity information

Mesh adaptation Goal-oriented mesh refinement: weigh a local error estimate with the sensitivity of the objective.

Multi-level approaches that use adaptively select the appropriate level of fidelity as the design space is explored.

Model calibration Determine constants of a constitutive model to best fit observations.

Automatic Differentiation

- If we have the source code of a program, we can differentiate each statement exactly, and assemble the complete derivative using the chain rule of calculus.
- This can be done systematically and automatically with a software tool.
- If there are fewer outputs than inputs, can assemble the transpose of the matrix of derivatives, the adjoint or reverse mode, which is much more efficient.

Automatic Differentiation

- If we have the source code of a program, we can differentiate each statement exactly, and assemble the complete derivative using the chain rule of calculus.
- This can be done systematically and automatically with a software tool.
- If there are fewer outputs than inputs, can assemble the transpose of the matrix of derivatives, the adjoint or reverse mode, which is much more efficient.

Automatic Differentiation

- If we have the source code of a program, we can differentiate each statement exactly, and assemble the complete derivative using the chain rule of calculus.
- This can be done systematically and automatically with a software tool.
- If there are fewer outputs than inputs, can assemble the transpose of the matrix of derivatives, the adjoint or reverse mode, which is much more efficient.

Outline

The adjoint method

Introduction to Algorithmic Differentiation

STAMPS - Source-Transformation Adjoint Multi-Purpose Solver

CAD-based shape optimisation

Uncertainty Quantification

Summary

Outline

The adjoint method

Introduction to Algorithmic Differentiation

STAMPS - Source-Transformation Adjoint Multi-Purpose Solver

CAD-based shape optimisation

Uncertainty Quantification

Summary

Outline

The adjoint method

Introduction to Algorithmic Differentiation

STAMPS - Source-Transformation Adjoint Multi-Purpose Solver

CAD-based shape optimisation

Uncertainty Quantification

Summary

Outline

The adjoint method

Introduction to Algorithmic Differentiation

STAMPS - Source-Transformation Adjoint Multi-Purpose Solver

CAD-based shape optimisation

Uncertainty Quantification

Summary

Outline

The adjoint method

Introduction to Algorithmic Differentiation

STAMPS - Source-Transformation Adjoint Multi-Purpose Solver

CAD-based shape optimisation

Uncertainty Quantification

Summary

Outline

The adjoint method

Introduction to Algorithmic Differentiation

STAMPS - Source-Transformation Adjoint Multi-Purpose Solver

CAD-based shape optimisation

Uncertainty Quantification

Summary

Outline

The adjoint method

Introduction to Algorithmic Differentiation

STAMPS - Source-Transformation Adjoint Multi-Purpose Solver

CAD-based shape optimisation

Uncertainty Quantification

Summary

Physical meaning of the adjoint equations

The flow equations ask the question: “Where does a perturbation travel to?”

What if we could ask the question: “Where does a perturbation come from?”

If we have N sticks, we need to ask N times

If we have M observation spots, we need to ask M times

Physical meaning of the adjoint equations

The flow equations ask the question: “Where does a perturbation travel to?”

If we have N sticks, we need to ask N times

What if we could ask the question: “Where does a perturbation come from?”

If we have M observation spots, we need to ask M times

Physical meaning of the adjoint equations

The flow equations ask the question: “Where does a perturbation travel to?”

If we have N sticks, we need to ask N times

What if we could ask the question: “Where does a perturbation come from?”

If we have M observation spots, we need to ask M times

Physical meaning of the adjoint equations

The flow equations ask the question: “Where does a perturbation travel to?”

If we have N sticks, we need to ask N times

What if we could ask the question: “Where does a perturbation come from?”

If we have M observation spots, we need to ask M times

Physical meaning of the adjoint equations

The flow equations ask the question: “Where does a perturbation travel to?”

If we have N sticks, we need to ask N times

What if we could ask the question: “Where does a perturbation come from?”

If we have M observation spots, we need to ask M times

Physical meaning of the adjoint equations

The flow equations ask the question: “Where does a perturbation travel to?”

If we have N sticks, we need to ask N times

What if we could ask the question: “Where does a perturbation come from?”

If we have M observation spots, we need to ask M times

Physical meaning of the adjoint equations

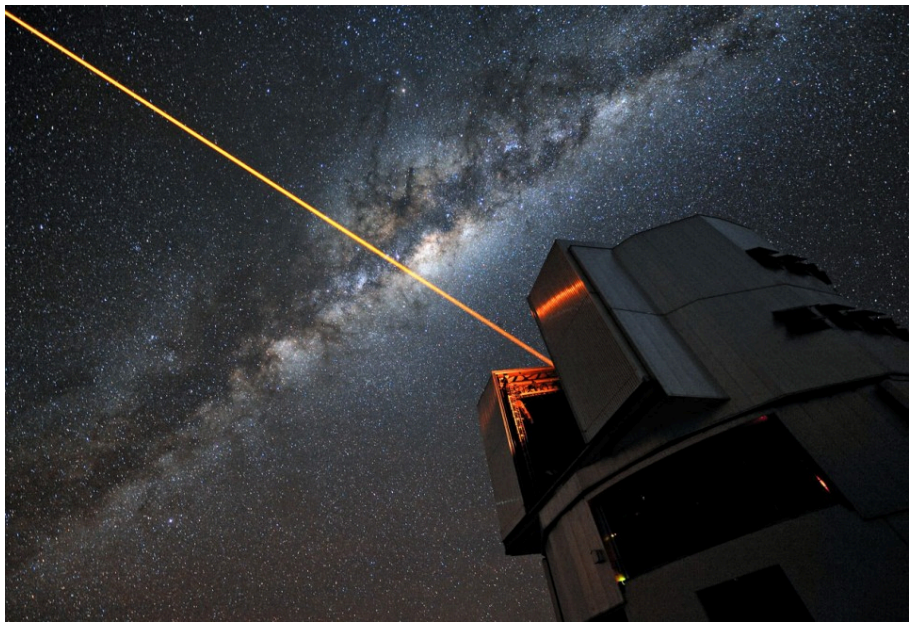
The flow equations ask the question: “Where does a perturbation travel to?”

If we have N sticks, we need to ask N times

What if we could ask the question: “Where does a perturbation come from?”

If we have M observation spots, we need to ask M times

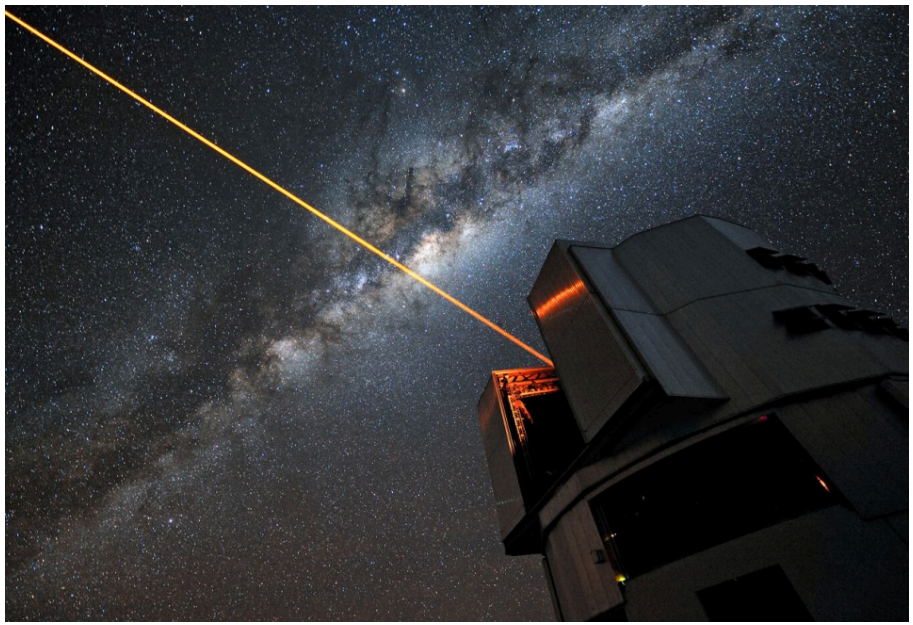
Adjoint equations: the galactic view



Reverse, adjoint approach:
trace back an incoming perturbation

Forward approach:
send a perturbation out

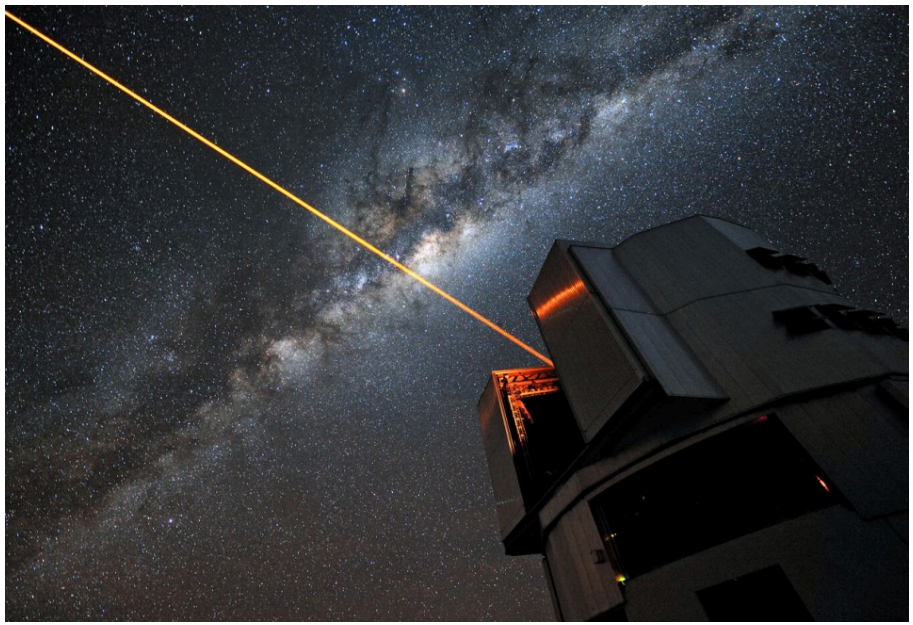
Adjoint equations: the galactic view

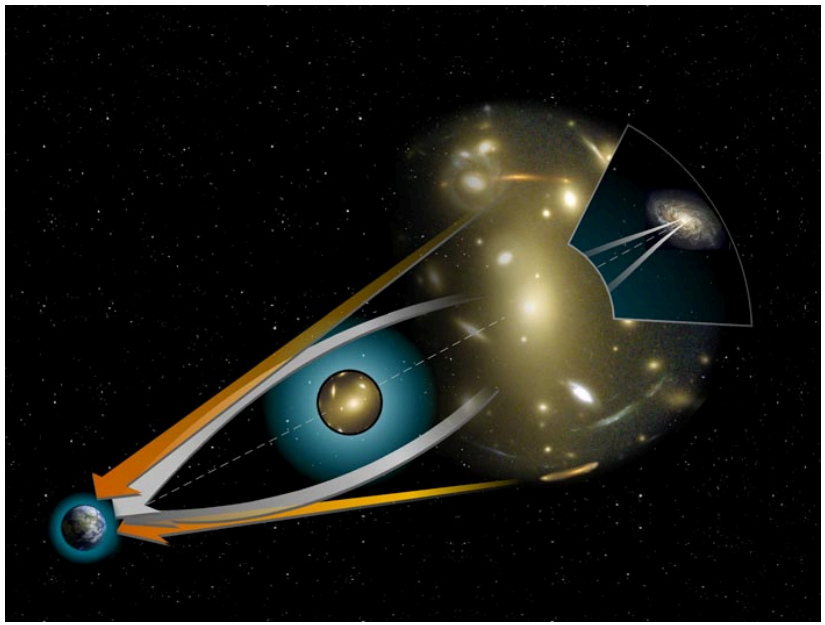


Reverse, adjoint approach:
trace back an incoming perturbation

Forward approach:
send a perturbation out

Adjoint equations: the galactic view

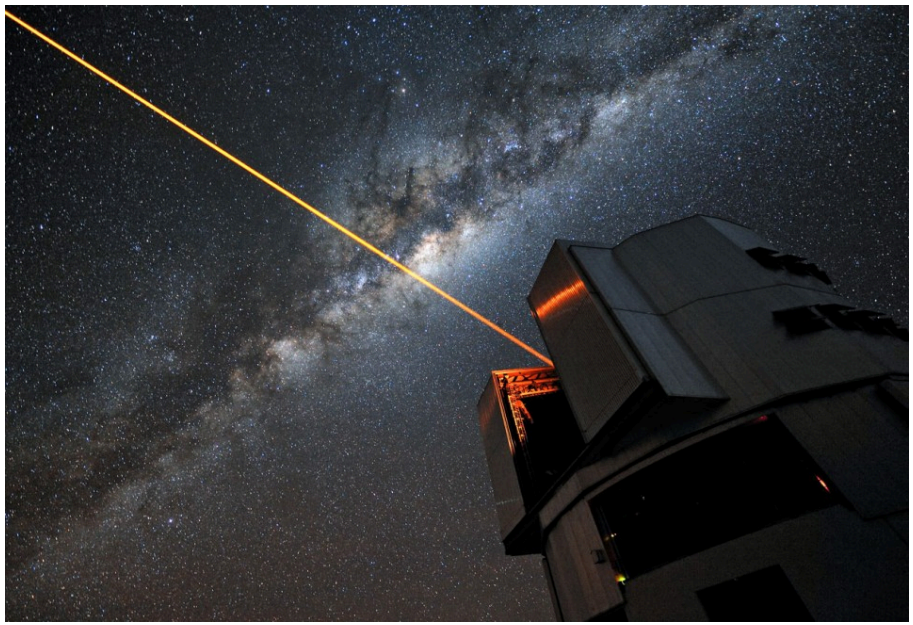


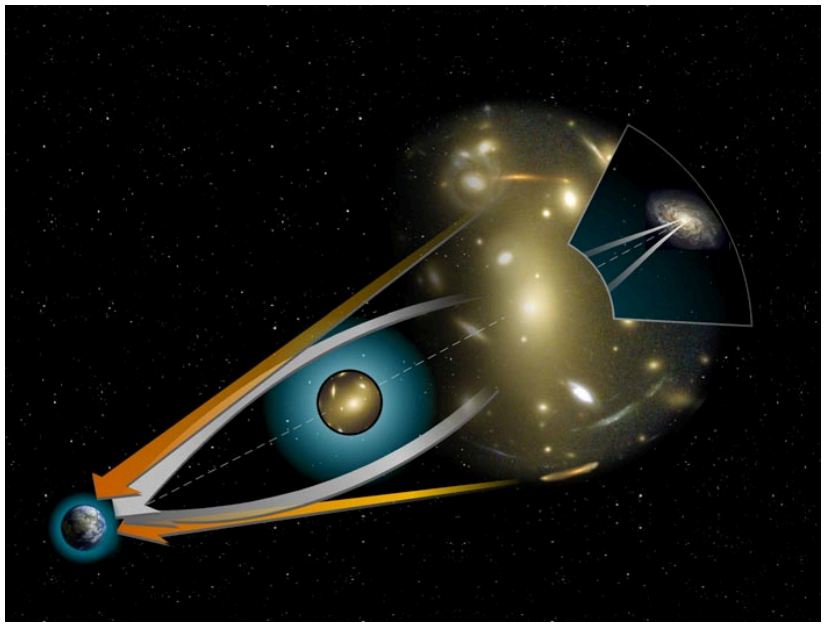


Forward approach:
send a perturbation out

Reverse, adjoint approach:
trace back an incoming perturbation

Adjoint equations: the galactic view

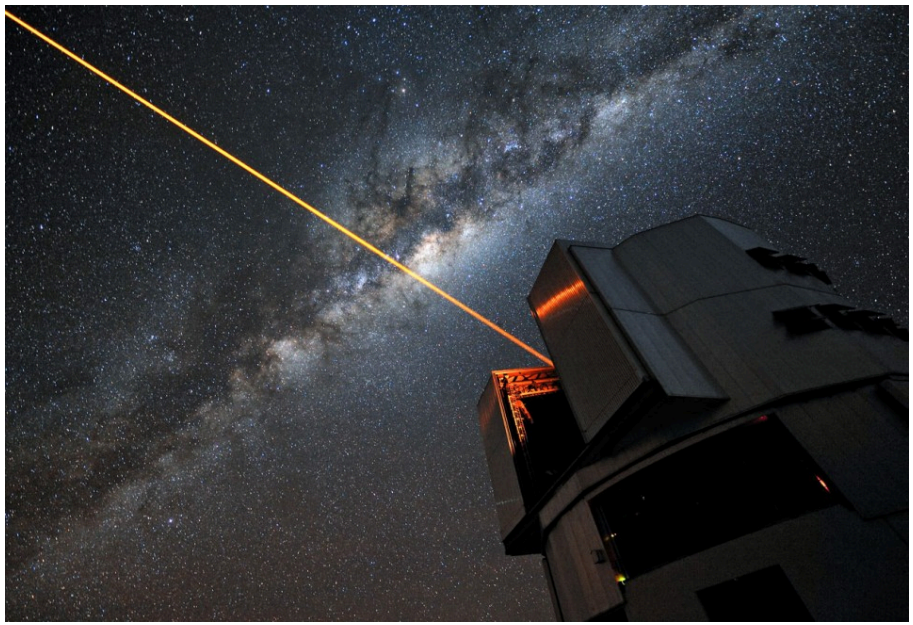


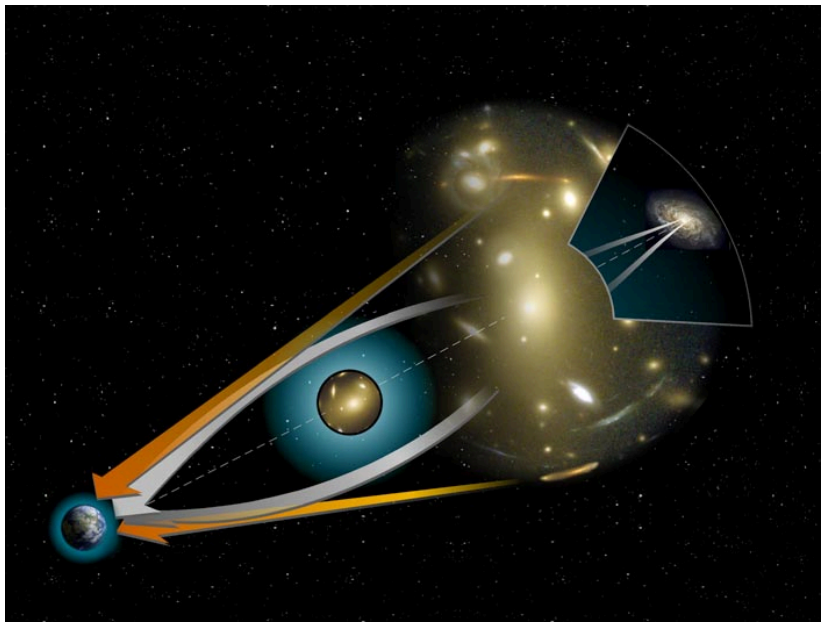


Forward approach:
send a perturbation out

Reverse, adjoint approach:
trace back an incoming perturbation

Adjoint equations: the galactic view



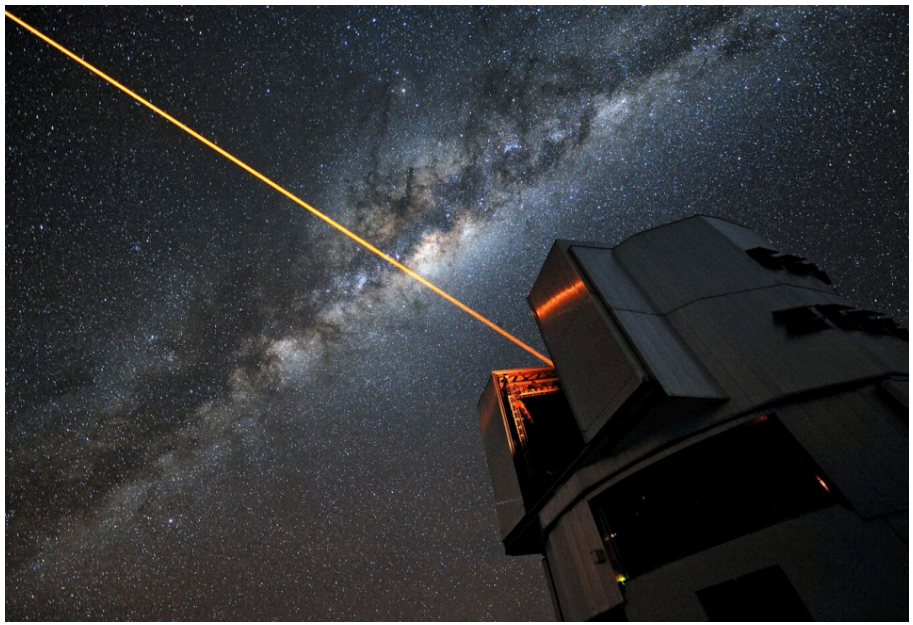


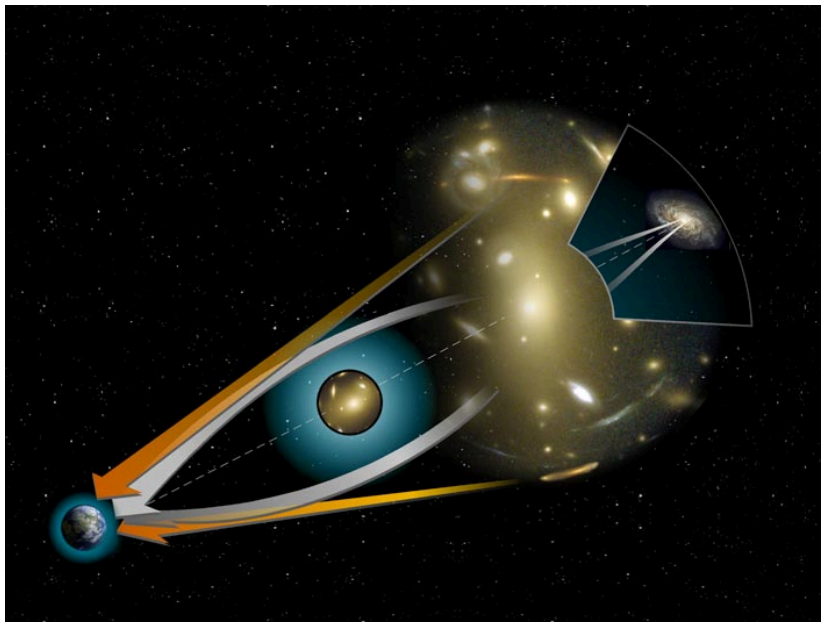
Forward approach:
send a perturbation out

Reverse, adjoint approach:
trace back an incoming perturbation

Use the Force!

Adjoint equations: the galactic view





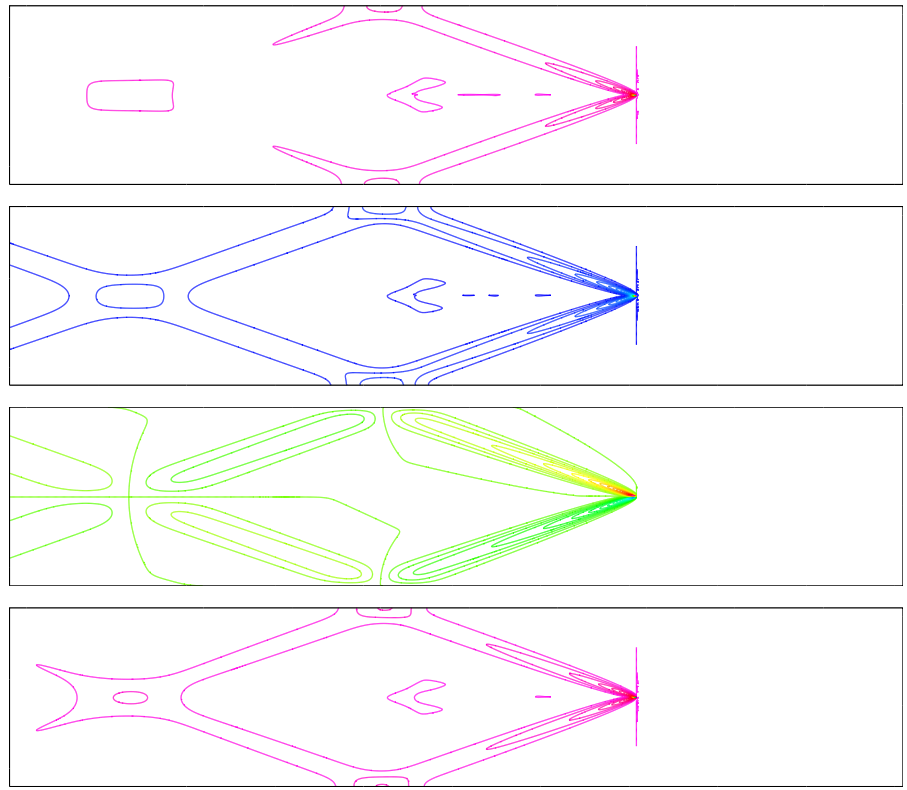
Forward approach:
send a perturbation out

Reverse, adjoint approach:
trace back an incoming perturbation

Use the Force of the adjoint approach.

Physical meaning of the adjoint

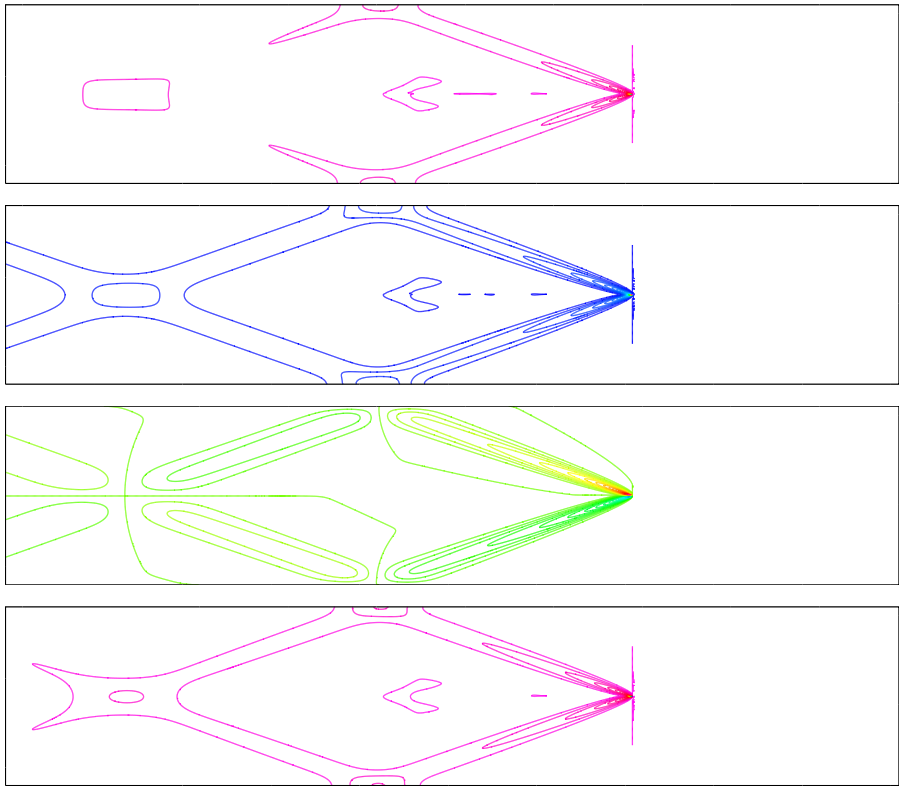
- The adjoint asks: where does a perturbation come from.
- This reverses all time-like directions, or 'transposes' the system matrix.
- The adjoint solution quantifies the effect on the objective function brought by a unit source term in the conservation equations.



Adjoint solution for objective function of pressure in a point in supersonic flow in a channel from left to right.

Physical meaning of the adjoint

- The adjoint asks: where does a perturbation come from.
- This reverses all time-like directions, or 'transposes' the system matrix.
- The adjoint solution quantifies the effect on the objective function brought by a unit source term in the conservation equations.



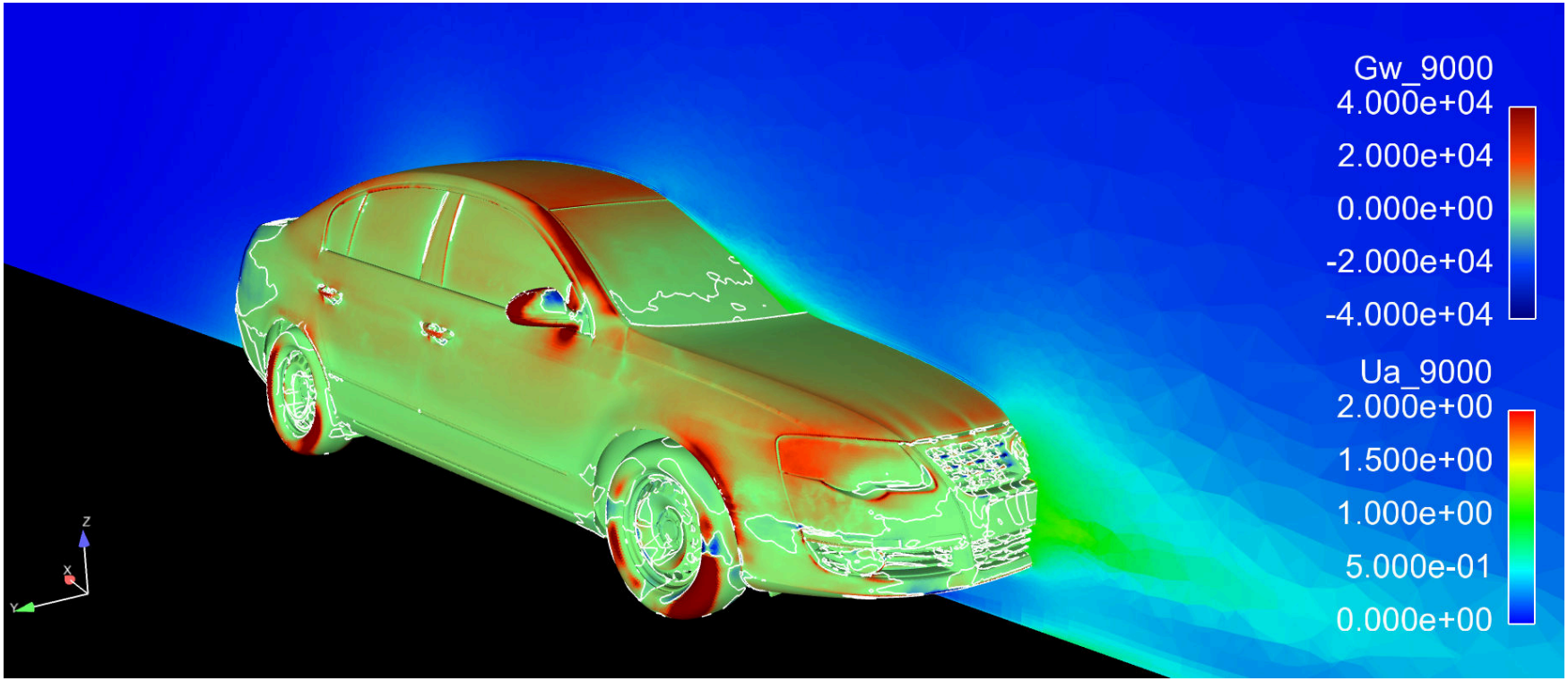
Adjoint solution for objective function of pressure in a point in supersonic flow in a channel from left to right.

Applications of adjoint solutions

Work out the product of adjoint and a source term arising from e.g. normal surface displacement of each mesh point:
to reduce drag: red: push in, blue: pull out

Applications of adjoint solutions

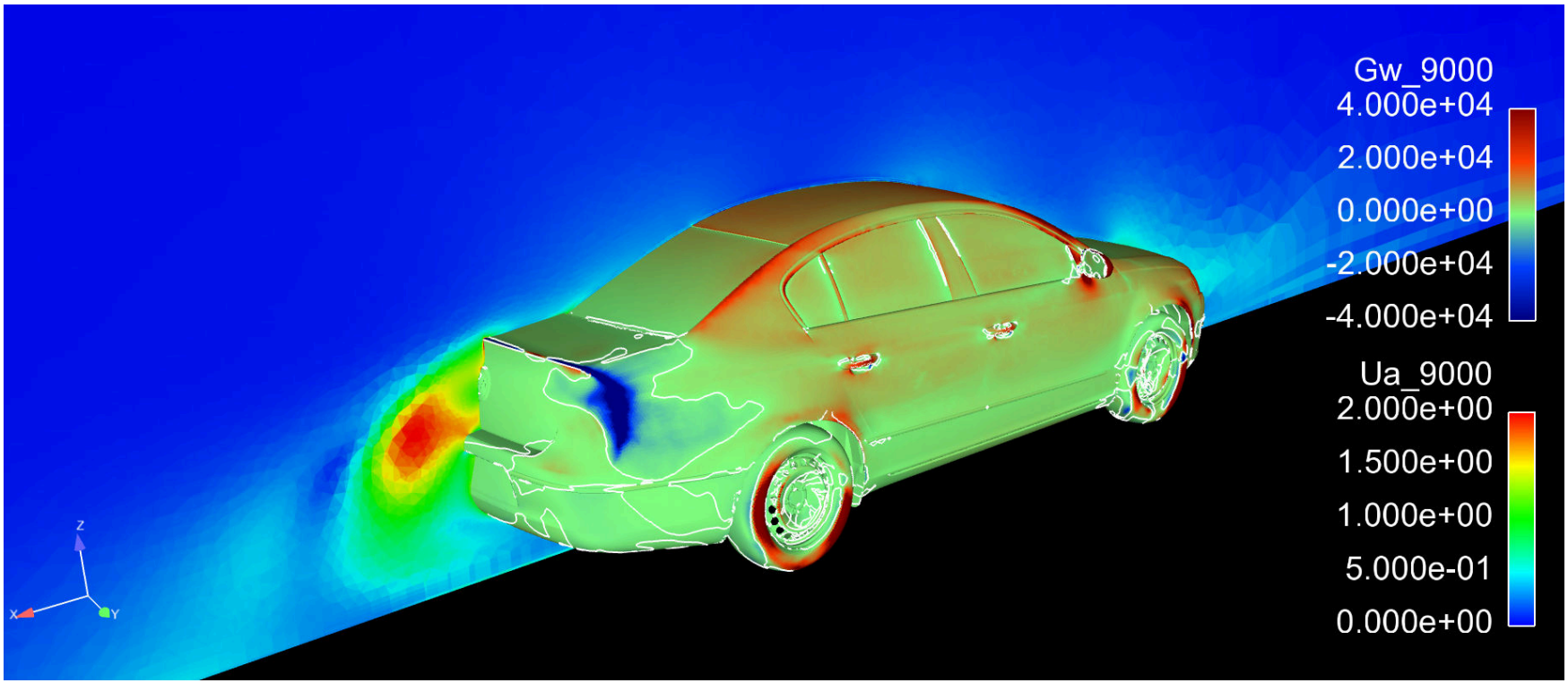
Work out the product of adjoint and a source term arising from e.g. normal surface displacement of each mesh point:
to reduce drag: red: push in, blue: pull out



(Adjoint OpenFOAM, Courtesy of Volkswagen)

Applications of adjoint solutions

Work out the product of adjoint and a source term arising from e.g. normal surface displacement of each mesh point:
to reduce drag: red: push in, blue: pull out



(Adjoint OpenFOAM, Courtesy of Volkswagen)

Finite difference derivative

Approximate the derivative as a forward difference

$$\frac{\partial f(x)}{\partial x_k} = \frac{f(x + \varepsilon\delta_k) - f(x)}{\varepsilon} + O(\varepsilon)$$

with ε a small perturbation size and δ_k a vector of the same length as x with zeros every where, but one in position k.

Similarly with a central difference

$$\frac{\partial f(x)}{\partial x_k} = \frac{f(x + \varepsilon\delta_k) - f(x - \varepsilon\delta_k)}{2\varepsilon} + O(\varepsilon^2)$$

Can we let $\varepsilon \to 0$ to make the truncation error vanish?

Finite difference derivative

Approximate the derivative as a forward difference

$$\frac{\partial f(x)}{\partial x_k} = \frac{f(x + \varepsilon\delta_k) - f(x)}{\varepsilon} + O(\varepsilon)$$

with ε a small perturbation size and δ_k a vector of the same length as x with zeros every where, but one in position k. Similarly with a central difference

$$\frac{\partial f(x)}{\partial x_k} = \frac{f(x + \varepsilon\delta_k) - f(x - \varepsilon\delta_k)}{2\varepsilon} + O(\varepsilon^2)$$

Can we let $\varepsilon \to 0$ to make the truncation error vanish?

Finite difference derivative

Approximate the derivative as a forward difference

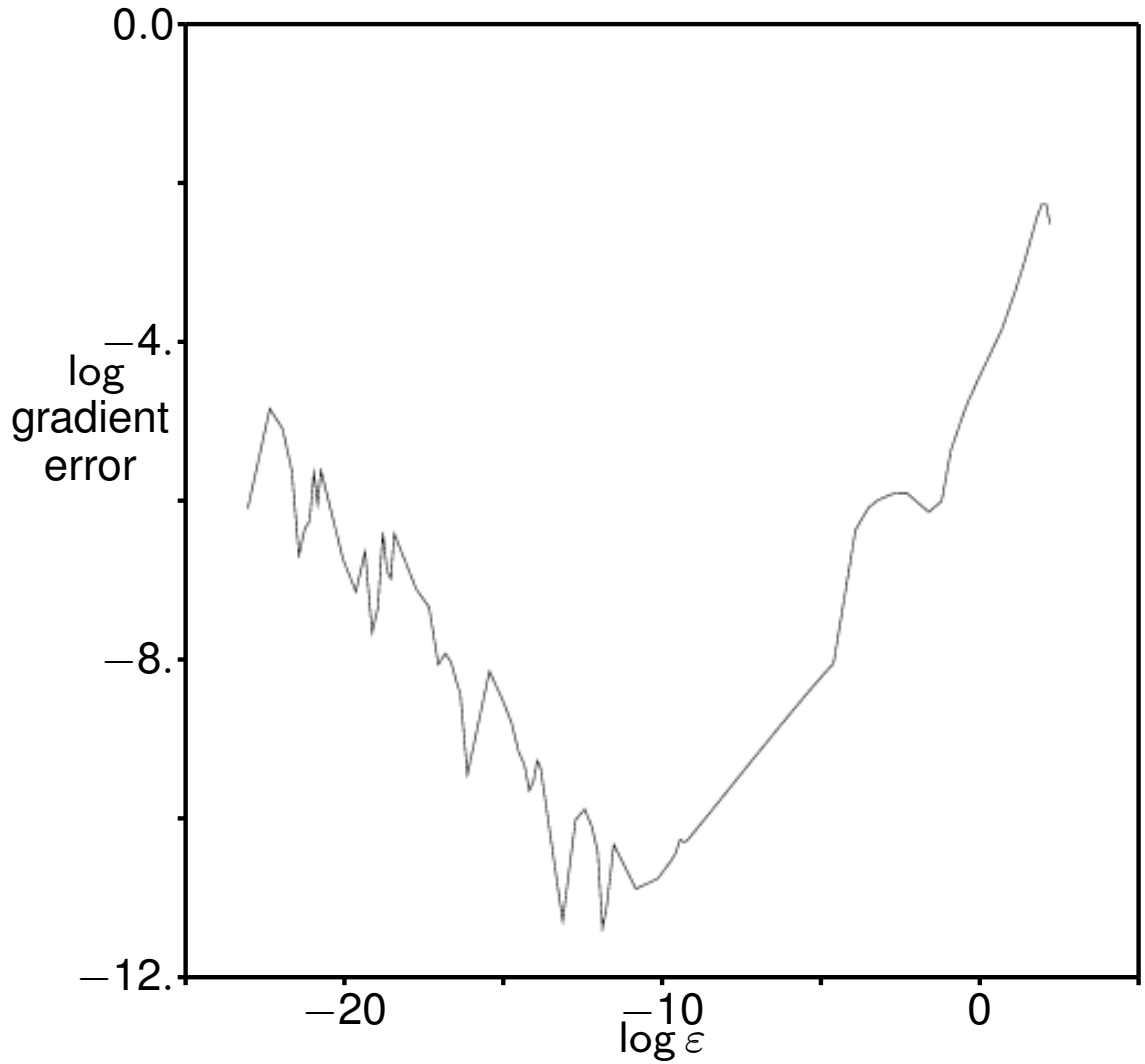
$$\frac{\partial f(x)}{\partial x_k} = \frac{f(x + \varepsilon\delta_k) - f(x)}{\varepsilon} + O(\varepsilon)$$

with ε a small perturbation size and δ_k a vector of the same length as x with zeros every where, but one in position k. Similarly with a central difference

$$\frac{\partial f(x)}{\partial x_k} = \frac{f(x + \varepsilon\delta_k) - f(x - \varepsilon\delta_k)}{2\varepsilon} + O(\varepsilon^2)$$

Can we let $\varepsilon \to 0$ to make the truncation error vanish?

Errors of finite differences



Forward difference error dependence on ε (CFD case)

Finite differences for gradient computation

- Needs no knowledge of the equations or implementation, can call $f(x)$ as *black-box*.
- Needs careful setting of the stepsize ε:
- If ε is too large, there is a large truncation error.
- T.E. is $\propto O(\varepsilon)$ for forward or backward differences, one additional evaluation per design variable.
- T.E. is $\propto O(\varepsilon^2)$ for the central difference, but costs two additional evaluations per design variable.
- If ε is too small, there is a large round-off error.
- For N design variables, we need to compute the objective N times!

Finite differences for gradient computation

- Needs no knowledge of the equations or implementation, can call $f(x)$ as *black-box*.
- Needs careful setting of the stepsize ε:
- If ε is too large, there is a large truncation error.
- T.E. is $\propto O(\varepsilon)$ for forward or backward differences, one additional evaluation per design variable.
- T.E. is $\propto O(\varepsilon^2)$ for the central difference, but costs two additional evaluations per design variable.
- If ε is too small, there is a large round-off error.
- For N design variables, we need to compute the objective N times!

Finite differences for gradient computation

- Needs no knowledge of the equations or implementation, can call $f(x)$ as *black-box*.
- Needs careful setting of the stepsize ε:
- If ε is too large, there is a large truncation error.
- T.E. is $\propto O(\varepsilon)$ for forward or backward differences, one additional evaluation per design variable.
- T.E. is $\propto O(\varepsilon^2)$ for the central difference, but costs two additional evaluations per design variable.
- If ε is too small, there is a large round-off error.
- For N design variables, we need to compute the objective N times!

Finite differences for gradient computation

- Needs no knowledge of the equations or implementation, can call $f(x)$ as *black-box*.
- Needs careful setting of the stepsize ε:
- If ε is too large, there is a large truncation error.
- T.E. is $\propto O(\varepsilon)$ for forward or backward differences, one additional evaluation per design variable.
- T.E. is $\propto O(\varepsilon^2)$ for the central difference, but costs two additional evaluations per design variable.
- If ε is too small, there is a large round-off error.
- For N design variables, we need to compute the objective N times!

Finite differences for gradient computation

- Needs no knowledge of the equations or implementation, can call $f(x)$ as *black-box*.
- Needs careful setting of the stepsize ε:
- If ε is too large, there is a large truncation error.
- T.E. is $\propto O(\varepsilon)$ for forward or backward differences, one additional evaluation per design variable.
- T.E. is $\propto O(\varepsilon^2)$ for the central difference, but costs two additional evaluations per design variable.
- If ε is too small, there is a large round-off error.
- For N design variables, we need to compute the objective N times!

Finite differences for gradient computation

- Needs no knowledge of the equations or implementation, can call $f(x)$ as *black-box*.
- Needs careful setting of the stepsize ε:
- If ε is too large, there is a large truncation error.
- T.E. is $\propto O(\varepsilon)$ for forward or backward differences, one additional evaluation per design variable.
- T.E. is $\propto O(\varepsilon^2)$ for the central difference, but costs two additional evaluations per design variable.
- If ε is too small, there is a large round-off error.
- For N design variables, we need to compute the objective N times!

Finite differences for gradient computation

- Needs no knowledge of the equations or implementation, can call $f(x)$ as *black-box*.
- Needs careful setting of the stepsize ε:
- If ε is too large, there is a large truncation error.
- T.E. is $\propto O(\varepsilon)$ for forward or backward differences, one additional evaluation per design variable.
- T.E. is $\propto O(\varepsilon^2)$ for the central difference, but costs two additional evaluations per design variable.
- If ε is too small, there is a large round-off error.
- For N design variables, we need to compute the objective N times!

Background on Adjoint PDE Sensitivities

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

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

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

Background on Adjoint PDE Sensitivities

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

$$\frac{dL}{d\alpha} = \frac{\partial L}{\partial \alpha} + \frac{\partial L}{\partial U}\frac{\partial U}{\partial \alpha} = \frac{\partial L}{\partial \alpha} + g^T u$$

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

Background on Adjoint PDE Sensitivities

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.

Background on Adjoint PDE Sensitivities

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.

Background on Adjoint PDE Sensitivities

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:

$$\mathbf{A}^{-T} g = v, \quad \text{i.e.} \quad \mathbf{A}^T v = g$$

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

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:

$$\mathbf{A}^{-T} g = v, \quad \text{i.e.} \quad \mathbf{A}^T v = g$$

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

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:

$$\mathbf{A}^{-T} g = v, \quad \text{i.e.} \quad \mathbf{A}^T v = g$$

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

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:

$$\mathbf{A}^{-T} g = v, \qquad \text{i.e.} \qquad \mathbf{A}^T v = g$$

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

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:

$$\mathbf{A}^{-T} g = v, \qquad \text{i.e.} \qquad \mathbf{A}^T v = g$$

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

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:

$$\mathbf{A}^{-T} g = v, \qquad \text{i.e.} \qquad \mathbf{A}^T v = g$$

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

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:

$$\mathbf{A}^{-T} g = v, \qquad \text{i.e.} \qquad \mathbf{A}^T v = g$$

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

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:

$$\mathbf{A}^{-T} g = v, \qquad \text{i.e.} \qquad \mathbf{A}^T v = g$$

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

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.**

Advantages of adjoint sensitivities (II)

- The forward method computes a perturbed flow field u and then the change in functional as $g^T u$.
- The adjoint solution directly computes the influence v of a source term f onto the functional L.
- We then need to evaluate the source f_i due to a design perturbation α_i.
- For a single design parameter, the cost of $g^T u$ and $v^T f$ are the same.
- **Using the adjoint the cost of gradient calculations for large design problems is essentially constant.**

Advantages of adjoint sensitivities (II)

- The forward method computes a perturbed flow field u and then the change in functional as $g^T u$.
- The adjoint solution directly computes the influence v of a source term f onto the functional L.
- We then need to evaluate the source f_i due to a design perturbation α_i.
- For a single design parameter, the cost of $g^T u$ and $v^T f$ are the same.
- **Using the adjoint the cost of gradient calculations for large design problems is essentially constant.**

Advantages of adjoint sensitivities (II)

- The forward method computes a perturbed flow field u and then the change in functional as $g^T u$.
- The adjoint solution directly computes the influence v of a source term f onto the functional L.
- We then need to evaluate the source f_i due to a design perturbation α_i.
- For a single design parameter, the cost of $g^T u$ and $v^T f$ are the same.
- **Using the adjoint the cost of gradient calculations for large design problems is essentially constant.**

Advantages of adjoint sensitivities (II)

- The forward method computes a perturbed flow field u and then the change in functional as $g^T u$.
- The adjoint solution directly computes the influence v of a source term f onto the functional L.
- We then need to evaluate the source f_i due to a design perturbation α_i.
- For a single design parameter, the cost of $g^T u$ and $v^T f$ are the same.
- **Using the adjoint the cost of gradient calculations for large design problems is essentially constant.**

Advantages of adjoint sensitivities (II)

- The forward method computes a perturbed flow field u and then the change in functional as $g^T u$.
- The adjoint solution directly computes the influence v of a source term f onto the functional L.
- We then need to evaluate the source f_i due to a design perturbation α_i.
- For a single design parameter, the cost of $g^T u$ and $v^T f$ are the same.
- **Using the adjoint the cost of gradient calculations for large design problems is essentially constant.**

Outline

The adjoint method

Introduction to Algorithmic Differentiation

STAMPS - Source-Transformation Adjoint Multi-Purpose Solver

CAD-based shape optimisation

Uncertainty Quantification

Summary

Algorithmic Differentiation (AD)

- A computer program that computes a function $f(x)$ can be viewed as a sequence of simple operations such as addition, multiplication, etc:

$$f(x) = f_n(f_{n-1}(\cdots f_2(f_1(x))))$$

- We can straightforwardly compute the derivative of each of these operations and concatenate the derivatives using the chain rule.

$$\frac{\partial f(x)}{\partial x_i} = \frac{\partial f_n}{\partial f_{n-1}} \cdot \frac{\partial f_{n-1}}{\partial f_{n-2}} \cdot \ldots \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1(x)}{\partial x_i}$$

- While f_1 can only be a function of the input variables x, f_n will typically also depend on intermediate results $f_{n-1}, f_{n-2}, \ldots$.
- We can proceed to compute the derivative (automatically) instruction by instruction.

Algorithmic Differentiation (AD)

- A computer program that computes a function $f(x)$ can be viewed as a sequence of simple operations such as addition, multiplication, etc:

$$f(x) = f_n(f_{n-1}(\cdots f_2(f_1(x))))$$

- We can straightforwardly compute the derivative of each of these operations and concatenate the derivatives using the chain rule.

$$\frac{\partial f(x)}{\partial x_i} = \frac{\partial f_n}{\partial f_{n-1}} \cdot \frac{\partial f_{n-1}}{\partial f_{n-2}} \cdot \ldots \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1(x)}{\partial x_i}$$

- While f_1 can only be a function of the input variables x, f_n will typically also depend on intermediate results $f_{n-1}, f_{n-2}, \ldots$.
- We can proceed to compute the derivative (automatically) instruction by instruction.

Algorithmic Differentiation (AD)

- A computer program that computes a function $f(x)$ can be viewed as a sequence of simple operations such as addition, multiplication, etc:

$$f(x) = f_n(f_{n-1}(\cdots f_2(f_1(x))))$$

- We can straightforwardly compute the derivative of each of these operations and concatenate the derivatives using the chain rule.

$$\frac{\partial f(x)}{\partial x_i} = \frac{\partial f_n}{\partial f_{n-1}} \cdot \frac{\partial f_{n-1}}{\partial f_{n-2}} \cdot \ldots \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1(x)}{\partial x_i}$$

- While f_1 can only be a function of the input variables x, f_n will typically also depend on intermediate results $f_{n-1}, f_{n-2}, \ldots$.
- We can proceed to compute the derivative (automatically) instruction by instruction.

Algorithmic Differentiation (AD)

- A computer program that computes a function $f(x)$ can be viewed as a sequence of simple operations such as addition, multiplication, etc:

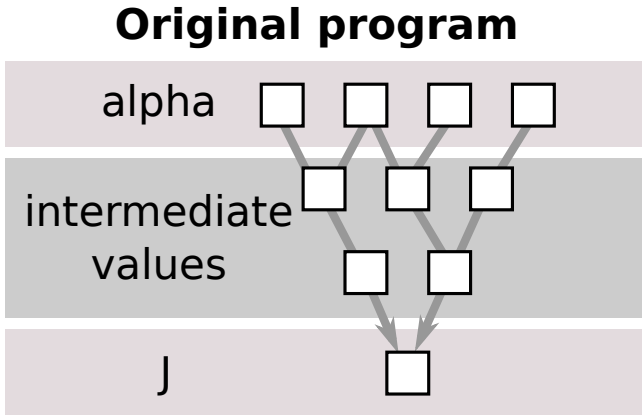
$$f(x) = f_n(f_{n-1}(\cdots f_2(f_1(x))))$$

- We can straightforwardly compute the derivative of each of these operations and concatenate the derivatives using the chain rule.

$$\frac{\partial f(x)}{\partial x_i} = \frac{\partial f_n}{\partial f_{n-1}} \cdot \frac{\partial f_{n-1}}{\partial f_{n-2}} \cdot \ldots \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1(x)}{\partial x_i}$$

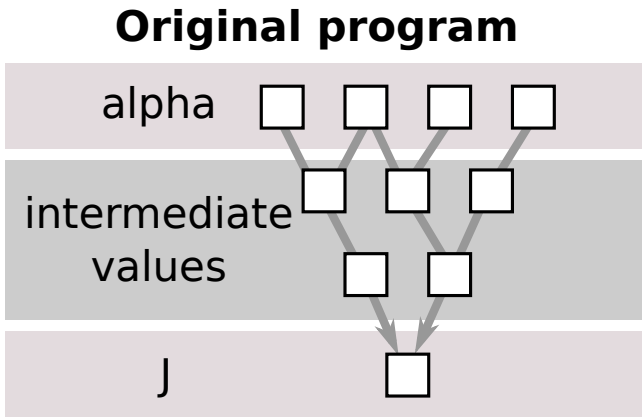
- While f_1 can only be a function of the input variables x, f_n will typically also depend on intermediate results $f_{n-1}, f_{n-2}, \ldots$.
- We can proceed to compute the derivative (automatically) instruction by instruction.

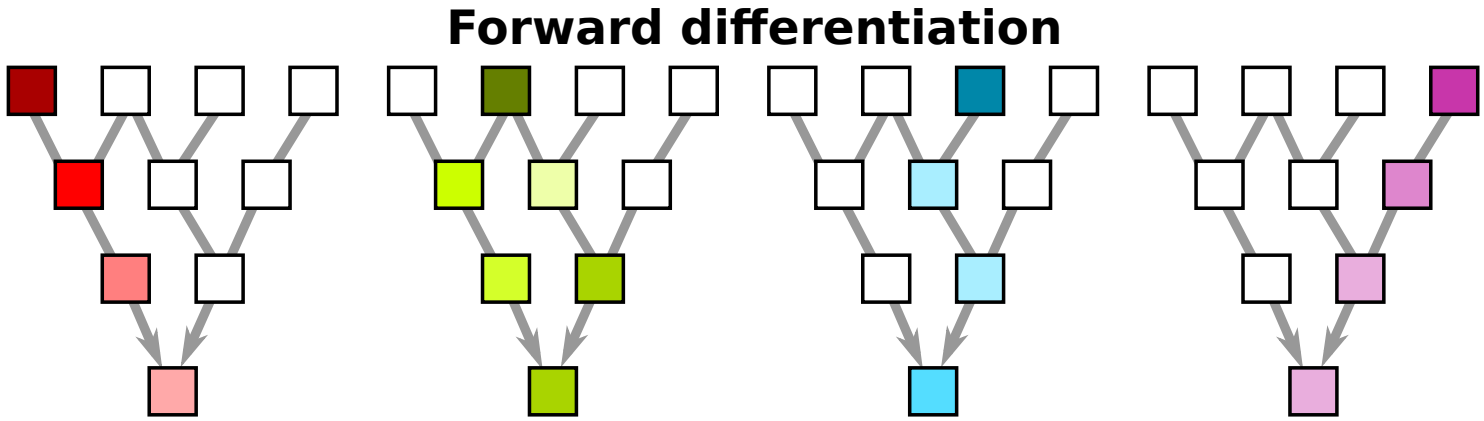
Algorithms as graphs



- Forward: propagate influence of each alpha through program
- Reverse: trace back every influence on result. One pass is enough to get all derivatives.

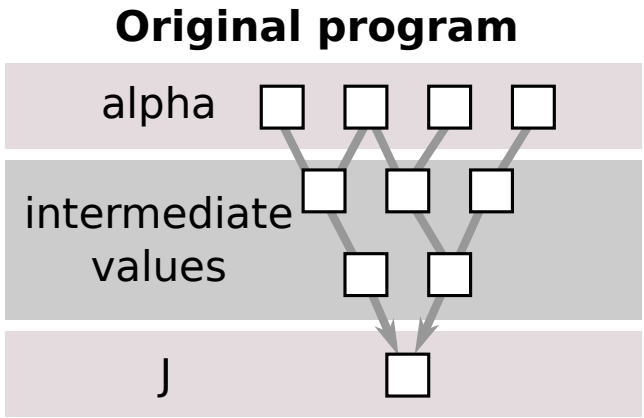
Algorithms as graphs

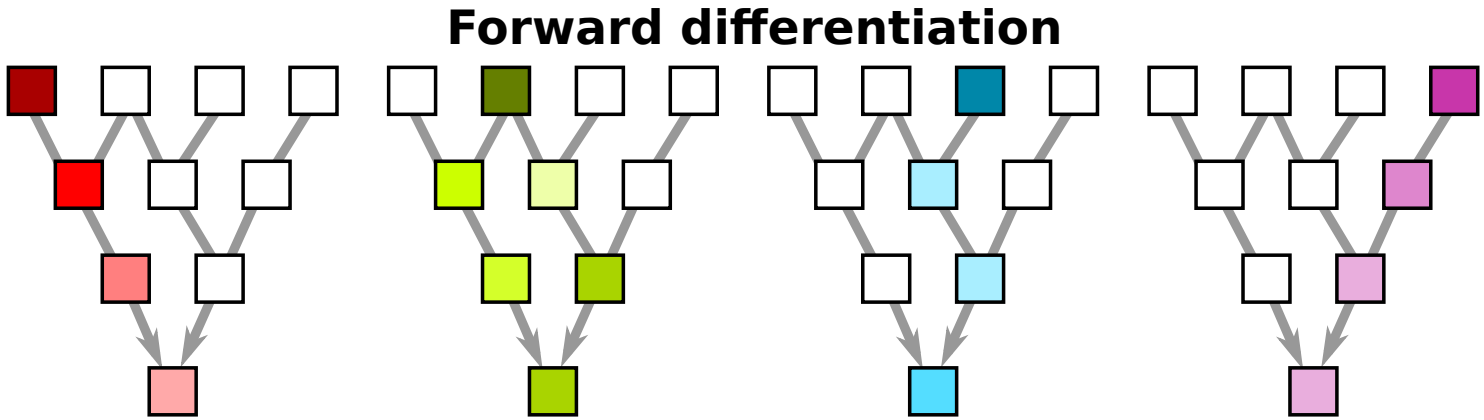




- Forward: propagate influence of each alpha through program
- Reverse: trace back every influence on result. One pass is enough to get all derivatives.

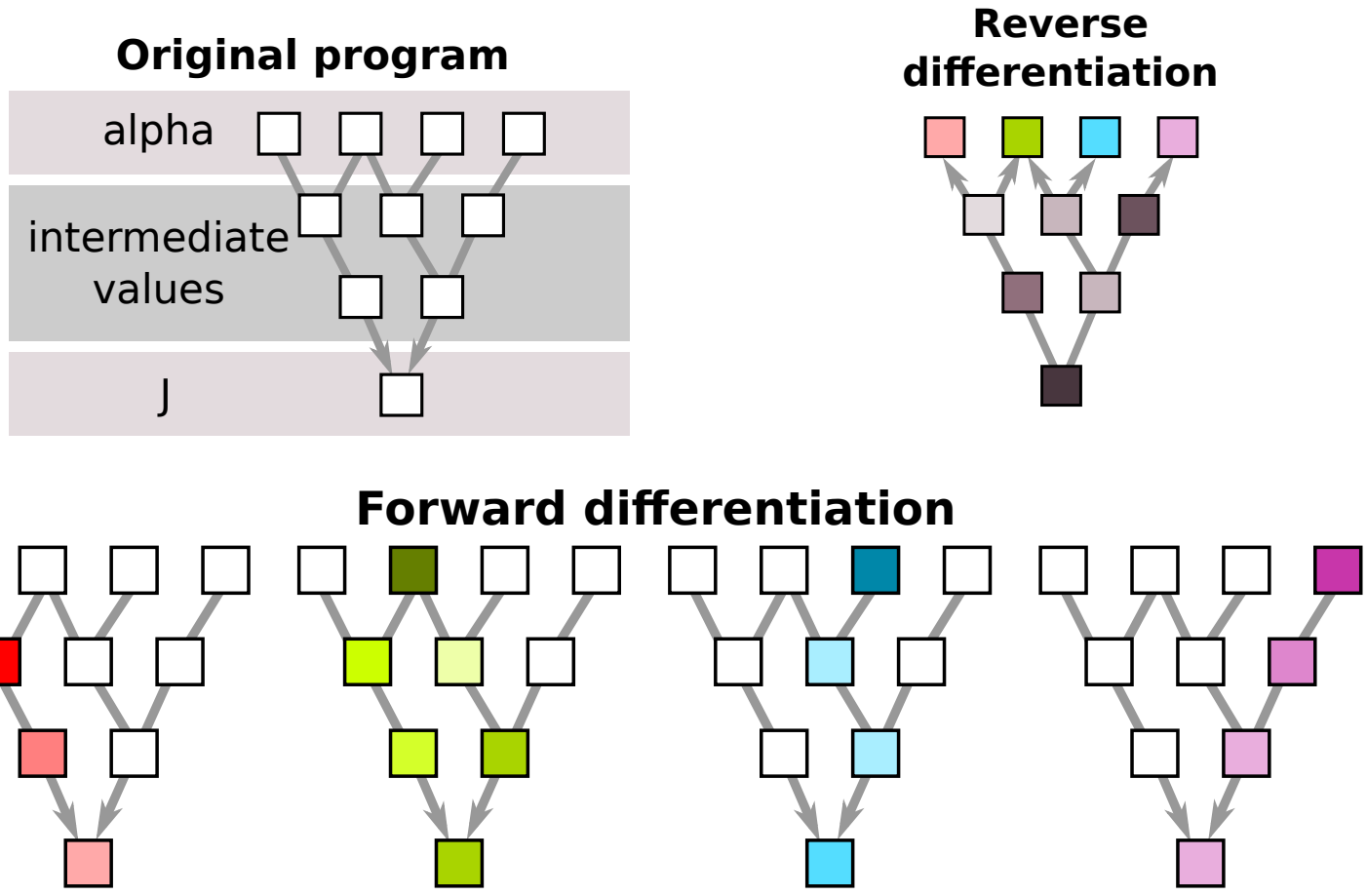
Algorithms as graphs





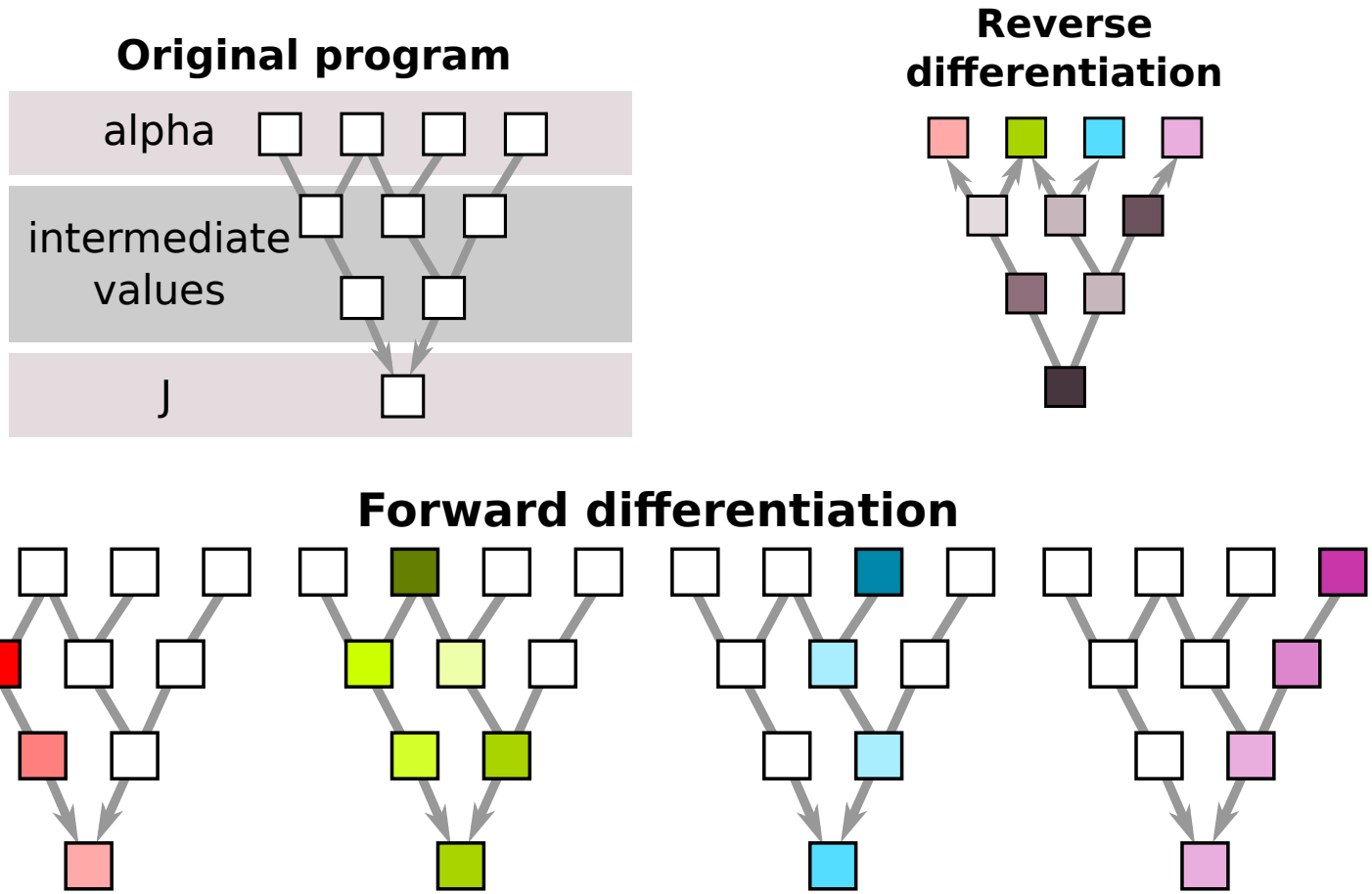
- Forward: propagate influence of each alpha through program
- Reverse: trace back every influence on result. One pass is enough to get all derivatives.

Algorithms as graphs



- Forward: propagate influence of each alpha through program
- Reverse: trace back every influence on result. One pass is enough to get all derivatives.

Algorithms as graphs



- Forward: propagate influence of each alpha through program
- Reverse: trace back every influence on result. One pass is enough to get all derivatives.

From Algorithmic to Automatic Differentiation

- Forward-mode steps through the statements in the same order, add a derivative computation statement before each primal statement.
- This is a straightforward (i.e. rigorous and stupid) process, why not have this done by software.
- The reverse-mode records all partial derivatives in each statement, then accumulates the derivatives in reverse.
- This is a straightforward (i.e. rigorous and stupid), potentially memory consuming process, why not have this done by software.

There are two main options to apply automatic differentiation:

- Operator-overloading
- Source-transformation

From Algorithmic to Automatic Differentiation

- Forward-mode steps through the statements in the same order, add a derivative computation statement before each primal statement.
- This is a straightforward (i.e. rigorous and stupid) process, why not have this done by software.
- The reverse-mode records all partial derivatives in each statement, then accumulates the derivatives in reverse.
- This is a straightforward (i.e. rigorous and stupid), potentially memory consuming process, why not have this done by software.

There are two main options to apply automatic differentiation:

- Operator-overloading
- Source-transformation

From Algorithmic to Automatic Differentiation

- Forward-mode steps through the statements in the same order, add a derivative computation statement before each primal statement.
- This is a straightforward (i.e. rigorous and stupid) process, why not have this done by software.
- The reverse-mode records all partial derivatives in each statement, then accumulates the derivatives in reverse.
- This is a straightforward (i.e. rigorous and stupid), potentially memory consuming process, why not have this done by software.

There are two main options to apply automatic differentiation:

- Operator-overloading
- Source-transformation

From Algorithmic to Automatic Differentiation

- Forward-mode steps through the statements in the same order, add a derivative computation statement before each primal statement.
- This is a straightforward (i.e. rigorous and stupid) process, why not have this done by software.
- The reverse-mode records all partial derivatives in each statement, then accumulates the derivatives in reverse.
- This is a straightforward (i.e. rigorous and stupid), potentially memory consuming process, why not have this done by software.

There are two main options to apply automatic differentiation:

- Operator-overloading
- Source-transformation

From Algorithmic to Automatic Differentiation

- Forward-mode steps through the statements in the same order, add a derivative computation statement before each primal statement.
- This is a straightforward (i.e. rigorous and stupid) process, why not have this done by software.
- The reverse-mode records all partial derivatives in each statement, then accumulates the derivatives in reverse.
- This is a straightforward (i.e. rigorous and stupid), potentially memory consuming process, why not have this done by software.

There are two main options to apply automatic differentiation:

- Operator-overloading
- Source-transformation

From Algorithmic to Automatic Differentiation

- Forward-mode steps through the statements in the same order, add a derivative computation statement before each primal statement.
- This is a straightforward (i.e. rigorous and stupid) process, why not have this done by software.
- The reverse-mode records all partial derivatives in each statement, then accumulates the derivatives in reverse.
- This is a straightforward (i.e. rigorous and stupid), potentially memory consuming process, why not have this done by software.

There are two main options to apply automatic differentiation:

- Operator-overloading
- Source-transformation

From Algorithmic to Automatic Differentiation

- Forward-mode steps through the statements in the same order, add a derivative computation statement before each primal statement.
- This is a straightforward (i.e. rigorous and stupid) process, why not have this done by software.
- The reverse-mode records all partial derivatives in each statement, then accumulates the derivatives in reverse.
- This is a straightforward (i.e. rigorous and stupid), potentially memory consuming process, why not have this done by software.

There are two main options to apply automatic differentiation:

- Operator-overloading
- Source-transformation

From Algorithmic to Automatic Differentiation

- Forward-mode steps through the statements in the same order, add a derivative computation statement before each primal statement.
- This is a straightforward (i.e. rigorous and stupid) process, why not have this done by software.
- The reverse-mode records all partial derivatives in each statement, then accumulates the derivatives in reverse.
- This is a straightforward (i.e. rigorous and stupid), potentially memory consuming process, why not have this done by software.

There are two main options to apply automatic differentiation:

- Operator-overloading
- Source-transformation

AD via Operator-Overloading

Principle:

- Most modern languages allow *operator-overloading*, i.e. to define special data-types and then define extensions of standard operations such as $*$ or $+$ for these data-types.
- E.g. we could define a derivative-enhanced double type:

```
struct {
  double val ;
  double val_d ;
} double_d
```

- An overloaded multiplication then would be:

```
double_d operator *( double_d a,double_d b ){
  double_d prod ;
  prod.val_d = a.val*b.val_d + a.val_d*b.val ;
  prod.val = a.val * b.val ;
  return ( prod ) ;
}
```

AD via Operator-Overloading

Principle:

- Most modern languages allow *operator-overloading*, i.e. to define special data-types and then define extensions of standard operations such as $*$ or $+$ for these data-types.
- E.g. we could define a derivative-enhanced double type:

```
struct {
  double val ;
  double val_d ;
} double_d
```

- An overloaded multiplication then would be:

```
double_d operator *( double_d a,double_d b ){
  double_d prod ;
  prod.val_d = a.val*b.val_d + a.val_d*b.val ;
  prod.val = a.val * b.val ;
  return ( prod ) ;
}
```

AD via Operator-Overloading

Principle:

- Most modern languages allow *operator-overloading*, i.e. to define special data-types and then define extensions of standard operations such as $*$ or $+$ for these data-types.
- E.g. we could define a derivative-enhanced double type:

```
struct {
  double val ;
  double val_d ;
} double_d
```

- An overloaded multiplication then would be:

```
double_d operator *( double_d a,double_d b ){
  double_d prod ;
  prod.val_d = a.val*b.val_d + a.val_d*b.val ;
  prod.val = a.val * b.val ;
  return ( prod ) ;
}
```

AD via Operator-Overloading

- Operator-overloading very naturally gives rise to a forward-mode differentiation.
- All operators need overloading, all simple data-types such as `double` promoted to enhanced ones `double_d`.
- For reverse mode we need create a *tape* of operations and operands which is then run backwards at the end.

Properties of operator-overloading AD

- Works in most cases out of the box. Often easy to apply.
- High memory requirements due to large tapes.
- The tape is difficult to analyse or inspect, limited possibilities to assemble differentiated parts in other code.
- The tape contains run-time analysis, only required code branches are differentiated.
- All `val` are calculated, whether or not needed to form `val.d`. Static compile-time optimisation is not possible.
- S-T AD usually outperforms O-O AD.

AD via Operator-Overloading

- Operator-overloading very naturally gives rise to a forward-mode differentiation.
- All operators need overloading, all simple data-types such as `double` promoted to enhanced ones `double_d`.
- For reverse mode we need create a *tape* of operations and operands which is then run backwards at the end.

Properties of operator-overloading AD

- Works in most cases out of the box. Often easy to apply.
- High memory requirements due to large tapes.
- The tape is difficult to analyse or inspect, limited possibilities to assemble differentiated parts in other code.
- The tape contains run-time analysis, only required code branches are differentiated.
- All `val` are calculated, whether or not needed to form `val.d`. Static compile-time optimisation is not possible.
- S-T AD usually outperforms O-O AD.

AD via Operator-Overloading

- Operator-overloading very naturally gives rise to a forward-mode differentiation.
- All operators need overloading, all simple data-types such as `double` promoted to enhanced ones `double_d`.
- For reverse mode we need create a *tape* of operations and operands which is then run backwards at the end.

Properties of operator-overloading AD

- Works in most cases out of the box. Often easy to apply.
- High memory requirements due to large tapes.
- The tape is difficult to analyse or inspect, limited possibilities to assemble differentiated parts in other code.
- The tape contains run-time analysis, only required code branches are differentiated.
- All `val` are calculated, whether or not needed to form `val.d`. Static compile-time optimisation is not possible.
- S-T AD usually outperforms O-O AD.

AD via Operator-Overloading

- Operator-overloading very naturally gives rise to a forward-mode differentiation.
- All operators need overloading, all simple data-types such as `double` promoted to enhanced ones `double_d`.
- For reverse mode we need create a *tape* of operations and operands which is then run backwards at the end.

Properties of operator-overloading AD

- Works in most cases out of the box. Often easy to apply.
- High memory requirements due to large tapes.
- The tape is difficult to analyse or inspect, limited possibilities to assemble differentiated parts in other code.
- The tape contains run-time analysis, only required code branches are differentiated.
- All `val` are calculated, whether or not needed to form `val.d`. Static compile-time optimisation is not possible.
- S-T AD usually outperforms O-O AD.

AD via Operator-Overloading

- Operator-overloading very naturally gives rise to a forward-mode differentiation.
- All operators need overloading, all simple data-types such as `double` promoted to enhanced ones `double_d`.
- For reverse mode we need create a *tape* of operations and operands which is then run backwards at the end.

Properties of operator-overloading AD

- Works in most cases out of the box. Often easy to apply.
- High memory requirements due to large tapes.
- The tape is difficult to analyse or inspect, limited possibilities to assemble differentiated parts in other code.
- The tape contains run-time analysis, only required code branches are differentiated.
- All `val` are calculated, whether or not needed to form `val.d`. Static compile-time optimisation is not possible.
- S-T AD usually outperforms O-O AD.

AD via Operator-Overloading

- Operator-overloading very naturally gives rise to a forward-mode differentiation.
- All operators need overloading, all simple data-types such as `double` promoted to enhanced ones `double_d`.
- For reverse mode we need create a *tape* of operations and operands which is then run backwards at the end.

Properties of operator-overloading AD

- Works in most cases out of the box. Often easy to apply.
- High memory requirements due to large tapes.
- The tape is difficult to analyse or inspect, limited possibilities to assemble differentiated parts in other code.
- The tape contains run-time analysis, only required code branches are differentiated.
- All `val` are calculated, whether or not needed to form `val.d`. Static compile-time optimisation is not possible.
- S-T AD usually outperforms O-O AD.

AD via Operator-Overloading

- Operator-overloading very naturally gives rise to a forward-mode differentiation.
- All operators need overloading, all simple data-types such as `double` promoted to enhanced ones `double_d`.
- For reverse mode we need create a *tape* of operations and operands which is then run backwards at the end.

Properties of operator-overloading AD

- Works in most cases out of the box. Often easy to apply.
- High memory requirements due to large tapes.
- The tape is difficult to analyse or inspect, limited possibilities to assemble differentiated parts in other code.
- The tape contains run-time analysis, only required code branches are differentiated.
- All `val` are calculated, whether or not needed to form `val.d`. Static compile-time optimisation is not possible.
- S-T AD usually outperforms O-O AD.

AD via Operator-Overloading

- Operator-overloading very naturally gives rise to a forward-mode differentiation.
- All operators need overloading, all simple data-types such as `double` promoted to enhanced ones `double_d`.
- For reverse mode we need create a *tape* of operations and operands which is then run backwards at the end.

Properties of operator-overloading AD

- Works in most cases out of the box. Often easy to apply.
- High memory requirements due to large tapes.
- The tape is difficult to analyse or inspect, limited possibilities to assemble differentiated parts in other code.
- The tape contains run-time analysis, only required code branches are differentiated.
- All `val` are calculated, whether or not needed to form `val.d`. Static compile-time optimisation is not possible.
- S-T AD usually outperforms O-O AD.

AD via Operator-Overloading

- Operator-overloading very naturally gives rise to a forward-mode differentiation.
- All operators need overloading, all simple data-types such as `double` promoted to enhanced ones `double_d`.
- For reverse mode we need create a *tape* of operations and operands which is then run backwards at the end.

Properties of operator-overloading AD

- Works in most cases out of the box. Often easy to apply.
- High memory requirements due to large tapes.
- The tape is difficult to analyse or inspect, limited possibilities to assemble differentiated parts in other code.
- The tape contains run-time analysis, only required code branches are differentiated.
- All `val` are calculated, whether or not needed to form `val.d`. Static compile-time optimisation is not possible.
- S-T AD usually outperforms O-O AD.

AD via Operator-Overloading

- Operator-overloading very naturally gives rise to a forward-mode differentiation.
- All operators need overloading, all simple data-types such as `double` promoted to enhanced ones `double_d`.
- For reverse mode we need create a *tape* of operations and operands which is then run backwards at the end.

Properties of operator-overloading AD

- Works in most cases out of the box. Often easy to apply.
- High memory requirements due to large tapes.
- The tape is difficult to analyse or inspect, limited possibilities to assemble differentiated parts in other code.
- The tape contains run-time analysis, only required code branches are differentiated.
- All `val` are calculated, whether or not needed to form `val.d`. Static compile-time optimisation is not possible.
- S-T AD usually outperforms O-O AD.

AD via source transformation

Procedure:

- *Parse* the statements in the primal source code
- then add the necessary statements to produce modified source code
- then compile the modified source code.

There are a number of source-transformation AD tools:
We use Tapenade (INRIA):

- Fortran or C. Combined mode under development.
- Forward and reverse,
- most mature and most popular tool,
- free one-year academic licenses.

AD via source transformation

Procedure:

- *Parse* the statements in the primal source code
- then add the necessary statements to produce modified source code
- then compile the modified source code.

There are a number of source-transformation AD tools:
We use Tapenade (INRIA):

- Fortran or C. Combined mode under development.
- Forward and reverse,
- most mature and most popular tool,
- free one-year academic licenses.

AD via source transformation

Procedure:

- *Parse* the statements in the primal source code
- then add the necessary statements to produce modified source code
- then compile the modified source code.

There are a number of source-transformation AD tools:
We use Tapenade (INRIA):

- Fortran or C. Combined mode under development.
- Forward and reverse,
- most mature and most popular tool,
- free one-year academic licenses.

AD via source transformation

Procedure:

- *Parse* the statements in the primal source code
- then add the necessary statements to produce modified source code
- then compile the modified source code.

There are a number of source-transformation AD tools:
We use Tapenade (INRIA):

- Fortran or C. Combined mode under development.
- Forward and reverse,
- most mature and most popular tool,
- free one-year academic licenses.

Properties of source transformation AD

Advantages/Disadvantages:

- Compile-time parsing can only take account of information available at compile-time (i.e. information embedded in the code structure), it is oblivious of run-time effect such as values of pointers.
- The entire code needs differentiating, regardless whether or not parts of the code will be used at run-time, unless pragma-hidden.
- Compiler optimises differentiated source code.
- Diff'ed source code can be analysed, to inform a rewrite of the primal to improve performance.
- Diff'ed source code modules can easily be assembled with non- or hand-differentiated code to optimise memory and runtime.

Properties of source transformation AD

Advantages/Disadvantages:

- Compile-time parsing can only take account of information available at compile-time (i.e. information embedded in the code structure), it is oblivious of run-time effect such as values of pointers.
- The entire code needs differentiating, regardless whether or not parts of the code will be used at run-time, unless pragma-hidden.
- Compiler optimises differentiated source code.
- Diff'ed source code can be analysed, to inform a rewrite of the primal to improve performance.
- Diff'ed source code modules can easily be assembled with non- or hand-differentiated code to optimise memory and runtime.

Properties of source transformation AD

Advantages/Disadvantages:

- Compile-time parsing can only take account of information available at compile-time (i.e. information embedded in the code structure), it is oblivious of run-time effect such as values of pointers.
- The entire code needs differentiating, regardless whether or not parts of the code will be used at run-time, unless pragma-hidden.
- Compiler optimises differentiated source code.
- Diff'ed source code can be analysed, to inform a rewrite of the primal to improve performance.
- Diff'ed source code modules can easily be assembled with non- or hand-differentiated code to optimise memory and runtime.

Properties of source transformation AD

Advantages/Disadvantages:

- Compile-time parsing can only take account of information available at compile-time (i.e. information embedded in the code structure), it is oblivious of run-time effect such as values of pointers.
- The entire code needs differentiating, regardless whether or not parts of the code will be used at run-time, unless pragma-hidden.
- Compiler optimises differentiated source code.
- Diff'ed source code can be analysed, to inform a rewrite of the primal to improve performance.
- Diff'ed source code modules can easily be assembled with non- or hand-differentiated code to optimise memory and runtime.

Properties of source transformation AD

Advantages/Disadvantages:

- Compile-time parsing can only take account of information available at compile-time (i.e. information embedded in the code structure), it is oblivious of run-time effect such as values of pointers.
- The entire code needs differentiating, regardless whether or not parts of the code will be used at run-time, unless pragma-hidden.
- Compiler optimises differentiated source code.
- Diff'ed source code can be analysed, to inform a rewrite of the primal to improve performance.
- Diff'ed source code modules can easily be assembled with non- or hand-differentiated code to optimise memory and runtime.

Properties of source transformation AD

Advantages/Disadvantages:

- Compile-time parsing can only take account of information available at compile-time (i.e. information embedded in the code structure), it is oblivious of run-time effect such as values of pointers.
- The entire code needs differentiating, regardless whether or not parts of the code will be used at run-time, unless pragma-hidden.
- Compiler optimises differentiated source code.
- Diff'ed source code can be analysed, to inform a rewrite of the primal to improve performance.
- Diff'ed source code modules can easily be assembled with non- or hand-differentiated code to optimise memory and runtime.

The need for open-source adjoint CFD solvers

Existing open-source adjoint CFD solvers:

- OpenFOAM: incompressible flow solver, continuous adjoint. Needs substantial expertise to adjoint models, to overcome stability issues.
- SU2: compressible flow solver, continuous adjoint. Needs substantial expertise to adjoint models, to overcome stability issues.
- SU2: compressible flow solver, discrete adjoint using operator-overloading tool Codipack.
  - Adjoint codes in tape/call-stack, not readable for the non-expert developer.
  - Memory requirements substantially improved over the years, but still multiple of the flow solver.
  - Obtaining similar performance for new models only by substantial expertise in Codipack.

Rationale for STAMPS

- provide a run-time and memory efficient open-source adjoint solver.

The need for open-source adjoint CFD solvers

Existing open-source adjoint CFD solvers:

- OpenFOAM: incompressible flow solver, continuous adjoint. Needs substantial expertise to adjoint models, to overcome stability issues.
- SU2: compressible flow solver, continuous adjoint. Needs substantial expertise to adjoint models, to overcome stability issues.
- SU2: compressible flow solver, discrete adjoint using operator-overloading tool Codipack.
  - Adjoint coding in tape/callback, not readable for the non-expert developer.
  - Memory requirements substantially improved over the years, but still multiple of the flow solver.
  - Obtaining similar performance for new models very time consuming, substantial expertise in Codipack.

Rationale for STAMPS

- provide a run-time and memory efficient open-source adjoint solver.

The need for open-source adjoint CFD solvers

Existing open-source adjoint CFD solvers:

- OpenFOAM: incompressible flow solver, continuous adjoint. Needs substantial expertise to adjoint models, to overcome stability issues.
- SU2: compressible flow solver, continuous adjoint. Needs substantial expertise to adjoint models, to overcome stability issues.
- SU2: compressible flow solver, discrete adjoint using operator-overloading tool Codipack.
  - Adjoint code is 'in tape/call stack', not readable to the non-expert developer.
  - Memory requirements substantially improved over the years, but still multiples of the flow solver.
  - Obtaining similar performance for new models may need substantial expertise in Codipack.

Rationale for STAMPS

- provide a run-time and memory efficient open-source adjoint solver.

The need for open-source adjoint CFD solvers

Existing open-source adjoint CFD solvers:

- SU2: compressible flow solver, discrete adjoint using operator-overloading tool Codipack.
  - Adjoint code is 'in tape/call stack', not readable to the non-expert developer.
  - Memory requirements substantially improved over the years, but still multiples of the flow solver.
  - Obtaining similar performance for new models may need substantial expertise in Codipack.

Rationale for STAMPS

- provide a run-time and memory efficient open-source adjoint solver.
- use source-transformation AD to enable the user to work with the adjoint code at multiple levels.
- provide a framework for fully automated adjoint code derivation of specified classes of algorithms.

The need for open-source adjoint CFD solvers

Existing open-source adjoint CFD solvers:

- SU2: compressible flow solver, discrete adjoint using operator-overloading tool Codipack.
  - Adjoint code is 'in tape/call stack', not readable to the non-expert developer.
  - Memory requirements substantially improved over the years, but still multiples of the flow solver.
  - Obtaining similar performance for new models may need substantial expertise in Codipack.

Rationale for STAMPS

- provide a run-time and memory efficient open-source adjoint solver.
- use source-transformation AD to enable the user to work with the adjoint code at multiple levels.
- provide a framework for fully automated adjoint code derivation of specified classes of algorithms.

The need for open-source adjoint CFD solvers

Existing open-source adjoint CFD solvers:

- SU2: compressible flow solver, discrete adjoint using operator-overloading tool Codipack.
  - Adjoint code is 'in tape/call stack', not readable to the non-expert developer.
  - Memory requirements substantially improved over the years, but still multiples of the flow solver.
  - Obtaining similar performance for new models may need substantial expertise in Codipack.

Rationale for STAMPS

- provide a run-time and memory efficient open-source adjoint solver.
- use source-transformation AD to enable the user to work with the adjoint code at multiple levels.
- provide a framework for fully automated adjoint code derivation of specified classes of algorithms.

The need for open-source adjoint CFD solvers

Existing open-source adjoint CFD solvers:

- SU2: compressible flow solver, discrete adjoint using operator-overloading tool Codipack.
  - Adjoint code is 'in tape/call stack', not readable to the non-expert developer.
  - Memory requirements substantially improved over the years, but still multiples of the flow solver.
  - Obtaining similar performance for new models may need substantial expertise in Codipack.

Rationale for STAMPS

- provide a run-time and memory efficient open-source adjoint solver.
- use source-transformation AD to enable the user to work with the adjoint code at multiple levels.
- provide a framework for fully automated adjoint code derivation of specified classes of algorithms.

The need for open-source adjoint CFD solvers

Existing open-source adjoint CFD solvers:

- SU2: compressible flow solver, discrete adjoint using operator-overloading tool Codipack.
  - Adjoint code is 'in tape/call stack', not readable to the non-expert developer.
  - Memory requirements substantially improved over the years, but still multiples of the flow solver.
  - Obtaining similar performance for new models may need substantial expertise in Codipack.

Rationale for STAMPS

- provide a run-time and memory efficient open-source adjoint solver.
- use source-transformation AD to enable the user to work with the adjoint code at multiple levels.
- provide a framework for fully automated adjoint code derivation of specified classes of algorithms.

The need for open-source adjoint CFD solvers

Existing open-source adjoint CFD solvers:

- SU2: compressible flow solver, discrete adjoint using operator-overloading tool Codipack.
  - Adjoint code is 'in tape/call stack', not readable to the non-expert developer.
  - Memory requirements substantially improved over the years, but still multiples of the flow solver.
  - Obtaining similar performance for new models may need substantial expertise in Codipack.

Rationale for STAMPS

- provide a run-time and memory efficient open-source adjoint solver.
- use source-transformation AD to enable the user to work with the adjoint code at multiple levels.
- provide a framework for fully automated adjoint code derivation of specified classes of algorithms.

The need for open-source adjoint CFD solvers

Existing open-source adjoint CFD solvers:

- SU2: compressible flow solver, discrete adjoint using operator-overloading tool Codipack.
  - Adjoint code is 'in tape/call stack', not readable to the non-expert developer.
  - Memory requirements substantially improved over the years, but still multiples of the flow solver.
  - Obtaining similar performance for new models may need substantial expertise in Codipack.

Rationale for STAMPS

- provide a run-time and memory efficient open-source adjoint solver.
- use source-transformation AD to enable the user to work with the adjoint code at multiple levels.
- provide a framework for fully automated adjoint code derivation of specified classes of algorithms.

The need for open-source adjoint CFD solvers

Existing open-source adjoint CFD solvers:

- SU2: compressible flow solver, discrete adjoint using operator-overloading tool Codipack.
  - Adjoint code is 'in tape/call stack', not readable to the non-expert developer.
  - Memory requirements substantially improved over the years, but still multiples of the flow solver.
  - Obtaining similar performance for new models may need substantial expertise in Codipack.

Rationale for STAMPS

- provide a run-time and memory efficient open-source adjoint solver.
- use source-transformation AD to enable the user to work with the adjoint code at multiple levels.
- provide a framework for fully automated adjoint code derivation of specified classes of algorithms.

Assembling efficient discrete adjoints

- We use a combination of AD and hand-differentiation:
  - Use AD for core routines like residual, gradients, limiters, fluxes...
  - Use hand-assembly for time-stepping, geometric multigrid and distributed-memory parallel communication
- Derivatives are a linearisation at a particular flow state. If the flow is at a steady state, only the final converged solution is needed to compute derivatives
- If the flow is unsteady (or the solver doesn't find a steady state), back-propagation of derivatives requires intermediate flow states. This typically requires large amount of memory, but can be reduced with check-pointing.
- In the first instance, focus on steady-state.

Assembling efficient discrete adjoints

- We use a combination of AD and hand-differentiation:
  - Use AD for core routines like residual, gradients, limiters, fluxes...
  - Use hand-assembly for time-stepping, geometric multigrid and distributed-memory parallel communication
- Derivatives are a linearisation at a particular flow state. If the flow is at a steady state, only the final converged solution is needed to compute derivatives
- If the flow is unsteady (or the solver doesn't find a steady state), back-propagation of derivatives requires intermediate flow states. This typically requires large amount of memory, but can be reduced with check-pointing.
- In the first instance, focus on steady-state.

Assembling efficient discrete adjoints

- We use a combination of AD and hand-differentiation:
  - Use AD for core routines like residual, gradients, limiters, fluxes...
  - Use hand-assembly for time-stepping, geometric multigrid and distributed-memory parallel communication
- Derivatives are a linearisation at a particular flow state. If the flow is at a steady state, only the final converged solution is needed to compute derivatives
- If the flow is unsteady (or the solver doesn't find a steady state), back-propagation of derivatives requires intermediate flow states. This typically requires large amount of memory, but can be reduced with check-pointing.
- In the first instance, focus on steady-state.

Assembling efficient discrete adjoints

- We use a combination of AD and hand-differentiation:
  - Use AD for core routines like residual, gradients, limiters, fluxes...
  - Use hand-assembly for time-stepping, geometric multigrid and distributed-memory parallel communication
- Derivatives are a linearisation at a particular flow state. If the flow is at a steady state, only the final converged solution is needed to compute derivatives
- If the flow is unsteady (or the solver doesn't find a steady state), back-propagation of derivatives requires intermediate flow states. This typically requires large amount of memory, but can be reduced with check-pointing.
- In the first instance, focus on steady-state.

QMUL use of AD

Source-transformation with Fortran90 (Tapenade, INRIA)

- Application of source-transformation AD to CFD in F90: STAMPS
- Advances with S-T AD tools (Tapenade) to improve robustness, efficiency.
- AD-derived Jacobians used in robust and stable iterative schemes (JT-KIRK, full Newton)
- Gradient-enabled NURBS-kernel NSPCC
- CSM solver Calculix: forward and reverse.

Operator-overloading in C++ (ADOL-C, Univ Paderborn)

- Differentiated OAD system OpenOpenCascade for vector and reverse.
- Differentiated turbomachinery CAD tool PADRAM (Rolls-Royce) forward.

QMUL use of AD

Source-transformation with Fortran90 (Tapenade, INRIA)

- Application of source-transformation AD to CFD in F90: STAMPS
- Advances with S-T AD tools (Tapenade) to improve robustness, efficiency.
- AD-derived Jacobians used in robust and stable iterative schemes (JT-KIRK, full Newton)
- Gradient-enabled NURBS-kernel NSPCC
- CSM solver Calculix: forward and reverse.

Operator-overloading in C++ (ADOL-C, Univ Paderborn)

- Differentiated CAD system OpenCascade, forward and reverse
- Differentiated turbomachinery CAD tool CADO (VKI), forward.

Outline

The adjoint method

Introduction to Algorithmic Differentiation

STAMPS - Source-Transformation Adjoint Multi-Purpose Solver

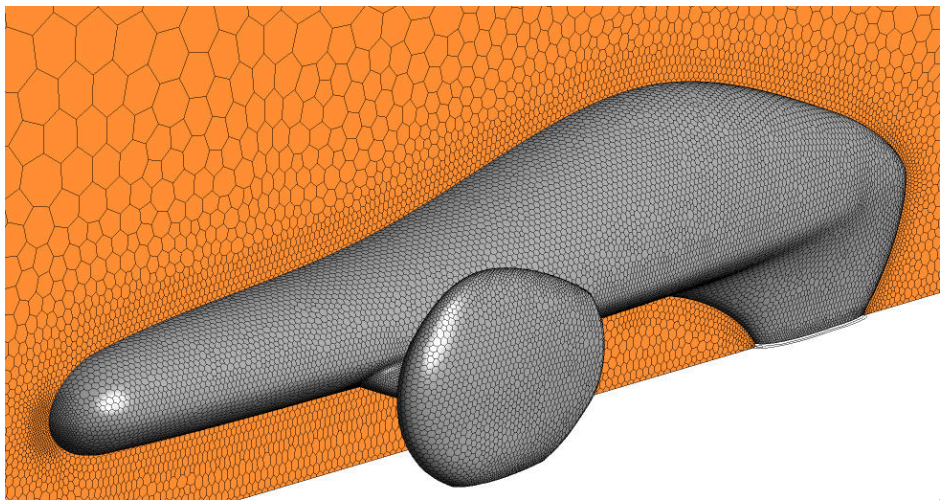
CAD-based shape optimisation

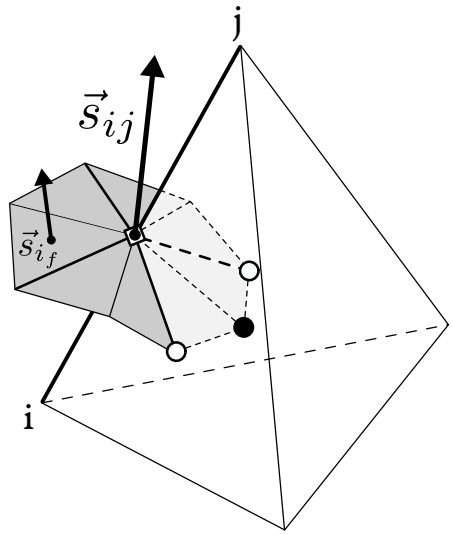
Uncertainty Quantification

Summary

STAMPS: discretisation

Source-**T**ransformation **A**djoint **M**ulti-**P**arametrisation, (Physics, Parallelism) **S**olver





- Unstructured 3-D finite volume, vertex-centred solver.
- Physics: inviscid, laminar, RANS-turbulent ideal gas.
- Mesh-deformation coupled with a variety of geometric parametrisations
- Interfaces for FSI, CHT.

STAMPS: discretisation

Typical finite-volume compressible flow discretisation:

- compressible formulation with Roe and AUSM+ fluxes, MUSCL reconstruction up to second order accuracy
- node-centred discretisation, edge-based fluxes, edge- and cell-based gradients,
- Spalart-Allmaras turbulence model,
- explicit, block-Jacobi and implicit (JT-KIRK)[1]. timestepping for steady-state and unsteady flows (BDF2)
- GMRES + ILU preconditioner.
- Parallelisation with MPI.

[1] Xu, Müller: JT-KIRK, JCP 2015

STAMPS: design capabilities

specifically designed as a discrete adjoint CFD solver:

- discrete adjoint solver: derivatives are consistent with the flow solver: linear properties such as spectral radius of Jacobian are guaranteed.
- fully differentiable with AD Tool Tapenade (Inria, France) in tangent and adjoint mode: build of the adjoint code is completely automated.
- uses efficient source-transformation AD: memory use is no more than 20% larger than the primal (CFD).
- coupled with a number of design parametrisation tools: node-based, NURBS-CAD-based and parameter-CAD-based.
- coupling with Calculix structures solver for FSI and CHT is currently undertaken.
- Adjoint-based mesh adaptation is currently being developed.

STAMPS: design capabilities

specifically designed as a discrete adjoint CFD solver:

- discrete adjoint solver: derivatives are consistent with the flow solver: linear properties such as spectral radius of Jacobian are guaranteed.
- fully differentiable with AD Tool Tapenade (Inria, France) in tangent and adjoint mode: build of the adjoint code is completely automated.
- uses efficient source-transformation AD: memory use is no more than 20% larger than the primal (CFD).
- coupled with a number of design parametrisation tools: node-based, NURBS-CAD-based and parameter-CAD-based.
- coupling with Calculix structures solver for FSI and CHT is currently undertaken.
- Adjoint-based mesh adaptation is currently being developed.

STAMPS: design capabilities

specifically designed as a discrete adjoint CFD solver:

- discrete adjoint solver: derivatives are consistent with the flow solver: linear properties such as spectral radius of Jacobian are guaranteed.
- fully differentiable with AD Tool Tapenade (Inria, France) in tangent and adjoint mode: build of the adjoint code is completely automated.
- uses efficient source-transformation AD: memory use is no more than 20% larger than the primal (CFD).
- coupled with a number of design parametrisation tools: node-based, NURBS-CAD-based and parameter-CAD-based.
- coupling with Calculix structures solver for FSI and CHT is currently undertaken.
- Adjoint-based mesh adaptation is currently being developed.

STAMPS: design capabilities

specifically designed as a discrete adjoint CFD solver:

- discrete adjoint solver: derivatives are consistent with the flow solver: linear properties such as spectral radius of Jacobian are guaranteed.
- fully differentiable with AD Tool Tapenade (Inria, France) in tangent and adjoint mode: build of the adjoint code is completely automated.
- uses efficient source-transformation AD: memory use is no more than 20% larger than the primal (CFD).
- coupled with a number of design parametrisation tools: node-based, NURBS-CAD-based and parameter-CAD-based.
- coupling with Calculix structures solver for FSI and CHT is currently undertaken.
- Adjoint-based mesh adaptation is currently being developed.

STAMPS: design capabilities

specifically designed as a discrete adjoint CFD solver:

- discrete adjoint solver: derivatives are consistent with the flow solver: linear properties such as spectral radius of Jacobian are guaranteed.
- fully differentiable with AD Tool Tapenade (Inria, France) in tangent and adjoint mode: build of the adjoint code is completely automated.
- uses efficient source-transformation AD: memory use is no more than 20% larger than the primal (CFD).
- coupled with a number of design parametrisation tools: node-based, NURBS-CAD-based and parameter-CAD-based.
- coupling with Calculix structures solver for FSI and CHT is currently undertaken.
- Adjoint-based mesh adaptation is currently being developed.

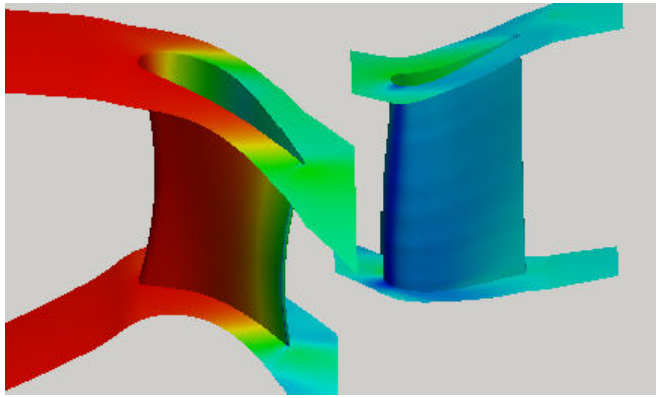
STAMPS: design capabilities

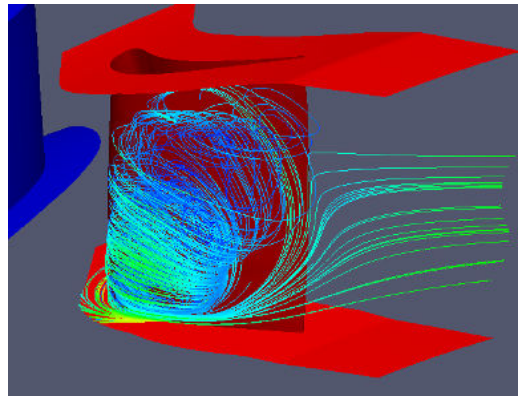
specifically designed as a discrete adjoint CFD solver:

- discrete adjoint solver: derivatives are consistent with the flow solver: linear properties such as spectral radius of Jacobian are guaranteed.
- fully differentiable with AD Tool Tapenade (Inria, France) in tangent and adjoint mode: build of the adjoint code is completely automated.
- uses efficient source-transformation AD: memory use is no more than 20% larger than the primal (CFD).
- coupled with a number of design parametrisation tools: node-based, NURBS-CAD-based and parameter-CAD-based.
- coupling with Calculix structures solver for FSI and CHT is currently undertaken.
- Adjoint-based mesh adaptation is currently being developed.

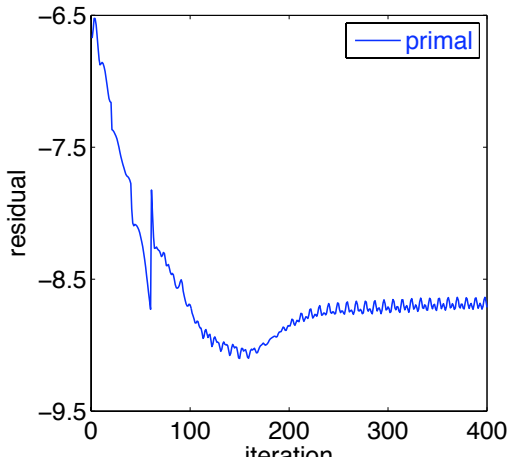
Convergence of the flow solver to limit cycles

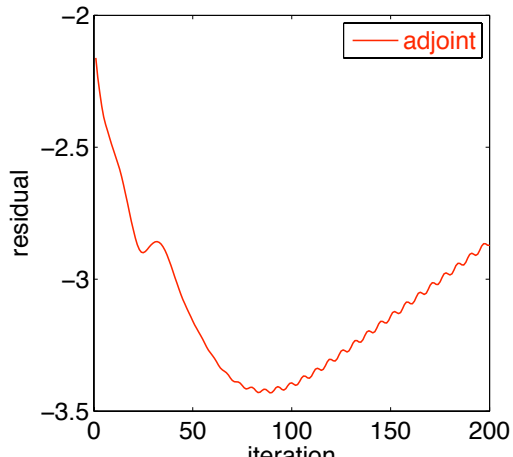
A major problem with adjoint solvers is robustness.





Turbomachinery case in off-design condition, convergence of the CFD (left) to limit cycles, divergence of the adjoint solver (right).

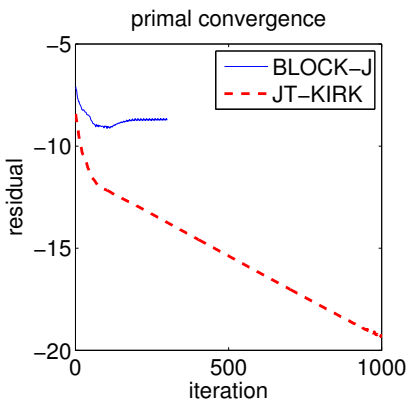


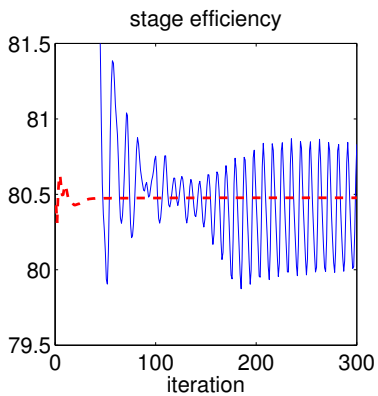


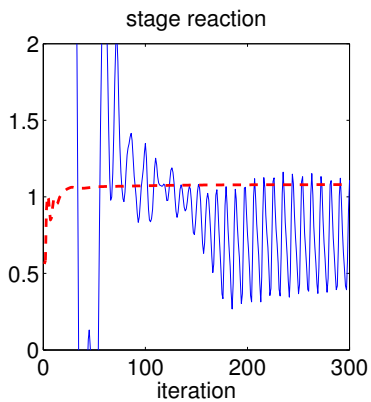
New iterative schemes for stable adjoints

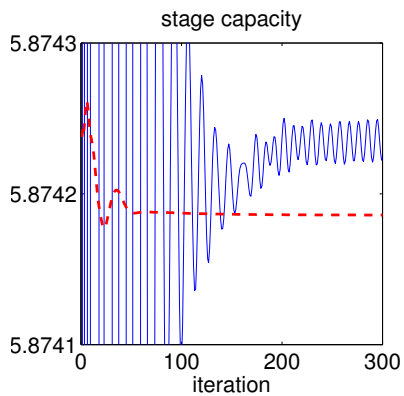
In collaboration with Rolls Royce the group developed the more stable JT-KIRK time-stepping scheme[2] that is

- more efficient in runtime for the flow solver
- more stable in achieving full convergence for flow and adjiont.
- Typical cost functions such as efficiency, reaction, capacity converge much more rapidly to steady-state.





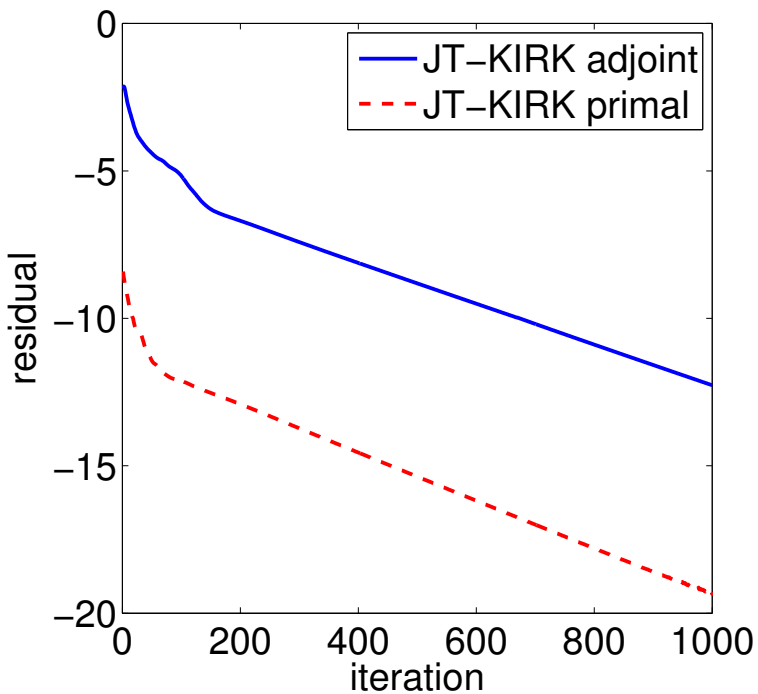




[2] S. Xu et al, “Stabilisation of discrete steady adjoint solvers”, JCP 2015

Stable adjoints: essential for industrial optimisation

- Most importantly, convergence of the discrete adjoint can be achieved even for mildly unsteady flow situations.
- This is an essential ingredient for industrial application of gradient-based optimisation using adjoint methods.



Convergence history of both JT-KIRK primal and adjoint solvers.

Framework for automatic application of S-T AD

Fully automated differentiation in tangent and reverse mode for

- fully coupled residual evaluation
- transport equations
- ILU precond. using AD'ed Jacobians
- Surface sensitivity projection
- adhering to coding templates ensures AD'ability
- two-layer halo MPI parallelisation: no MPI comm inside the FPI loop, no need to differentiate through MPI calls.
- Extensive use of Multi-Activity mode in Tapenade to derive efficient code for specialised derivative instances.

Framework for automatic application of S-T AD

Fully automated differentiation in tangent and reverse mode for

- fully coupled residual evaluation
- transport equations
- ILU precond. using AD'ed Jacobians
- Surface sensitivity projection
- adhering to coding templates ensures AD'ability
- two-layer halo MPI parallelisation: no MPI comm inside the FPI loop, no need to differentiate through MPI calls.
- Extensive use of Multi-Activity mode in Tapenade to derive efficient code for specialised derivative instances.

Framework for automatic application of S-T AD

Fully automated differentiation in tangent and reverse mode for

- fully coupled residual evaluation
- transport equations
- ILU precond. using AD'ed Jacobians
- Surface sensitivity projection
- adhering to coding templates ensures AD'ability
- two-layer halo MPI parallelisation: no MPI comm inside the FPI loop, no need to differentiate through MPI calls.
- Extensive use of Multi-Activity mode in Tapenade to derive efficient code for specialised derivative instances.

Framework for automatic application of S-T AD

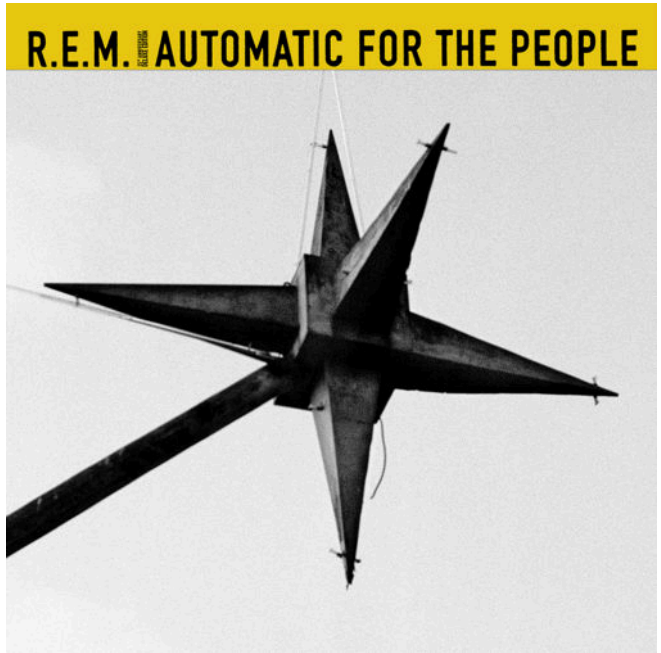
Fully automated differentiation in tangent and reverse mode for

- fully coupled residual evaluation
- transport equations
- ILU precond. using AD'ed Jacobians
- Surface sensitivity projection
- adhering to coding templates ensures AD'ability
- two-layer halo MPI parallelisation: no MPI comm inside the FPI loop, no need to differentiate through MPI calls.
- Extensive use of Multi-Activity mode in Tapenade to derive efficient code for specialised derivative instances.

Framework for automatic application of S-T AD

Fully automated differentiation in tangent and reverse mode for

- fully coupled residual evaluation
- transport equations
- ILU precond. using AD'ed Jacobians
- Surface sensitivity projection
- adhering to coding templates ensures AD'ability
- two-layer halo MPI parallelisation: no MPI comm inside the FPI loop, no need to differentiate through MPI calls.
- Extensive use of Multi-Activity mode in Tapenade to derive efficient code for specialised derivative instances.



AD: 'brute-force' iterators

Simplified compressible fixed-point iterator

```
call initialise_flow ( ←U )
call metrics ( →X, ←Nrm )
do nIter = 1,mIt
  call residual ( →U, →Nrm, ←R )
  call update ( →R, ⇌U )
  call push_to_tape ( →U )
end do
call cost_fun ( →U, →Nrm, ←J )
```

Adjoint iterator derived from 'brute-force' AD

```
call initialise_flow ( ←Ū=0 )
call cost_fun ( ←Ū, ←Nrm̄, 1 )
do nIter = mIt,-1,-1
  call pop_from_tape ( →U )
  call update ( ←R̄, ⇌Ū )
  call residual ( ←Ū, ←Nrm̄, →R̄ )
end do
call metrics ( ←X̄, →Nrm̄ )
```

AD: 'brute-force' iterators

Simplified compressible fixed-point iterator

```
call initialise_flow ( ←U )
call metrics ( →X, ←Nrm )
do nIter = 1,mIt
  call residual ( →U, →Nrm, ←R )
  call update ( →R, ⇌U )
  call push_to_tape ( →U )
end do
call cost_fun ( →U, →Nrm, ←J )
```

Adjoint iterator derived from 'brute-force' AD

```
call initialise_flow ( ←Ū=0 )
call cost_fun ( ←Ū, ←Nrm̄, 1 )
do nIter = mIt,-1,-1
  call pop_from_tape ( ←U )
  call update ( ←R̄, ⇌Ū )
  call residual ( ←Ū, ←Nrm̄, →R̄ )
end do
call metrics ( ←X̄, →Nrm̄ )
```

AD: 'brute-force' iterators

Simplified compressible fixed-point iterator

```
call initialise_flow ( ←U )
call metrics ( →X, ←Nrm )
do nIter = 1,mIt
  call residual ( →U, →Nrm, ←R )
  call update ( →R, ⇌U )
  call push_to_tape ( →U )
end do
call cost_fun ( →U, →Nrm, ←J )
```

Adjoint iterator derived from 'brute-force' AD

```
call initialise_flow ( ←Ū=0 )
call cost_fun ( ←Ū, ←Nrm̄, 1 )
do nIter = mIt,-1,-1
  call pop_from_tape ( ←U )
  call update ( ←R̄, ⇌Ū )
  call residual ( ←Ū, ←Nrm̄, →R̄ )
end do
call metrics ( ←X̄, →Nrm̄ )
```

AD: 'brute-force' fixed-point iterators

Simplified compressible fixed-point iterator

```
call initialise_flow ( ←U )
call metrics ( →X, ←Nrm )
do nIter = 1,mIt
  call residual ( →U, →Nrm, ←R )
  call update ( →R, ⇌U )
end do
call cost_fun ( →U, →Nrm, ←J )
```

Adjoint iterator derived from 'steady-state' AD

```
call initialise_flow ( ←Ū=0 )
call cost_fun ( ←Ū, ←Nrm̄, 1 )
do nIter = mIt,-1,-1
  call update ( ←R̄, ⇌Ū )
  call residual ( ←Ū, ←Nrm̄, →R̄ )
end do
call metrics ( ←X̄, →Nrm̄ )
```

AD: 'brute-force' fixed-point iterators

Simplified compressible fixed-point iterator

```
call initialise_flow ( ←U )
call metrics ( →X, ←Nrm )
do nIter = 1,mIt
  call residual ( →U, →Nrm, ←R )
  call update ( →R, ⇌U )
end do
call cost_fun ( →U, →Nrm, ←J )
```

Adjoint iterator derived from 'steady-state' AD

```
call initialise_flow ( ←Ū=0 )
call cost_fun ( ←Ū, ←Nrm̄, 1 )
do nIter = mIt,-1,-1
  call update ( ←R̄, ⇌Ū )
  call residual ( ←Ū, ←Nrm̄, →R̄ )
end do
call metrics ( ←X̄, →Nrm̄ )
```

Adjoint fixed-point iterators with 'reverse accumulation'

Simplified compressible fixed-point iterator

```
call initialise_flow ( ←U )
call metrics ( →X, ←Nrm )
do nIter = 1,mIt
  call residual ( →U, →Nrm, ←R )
  call update ( →R, ⇌U )
end do
call cost_fun ( →U, →Nrm, ←J )
```

Adjoint iterator derived from the primal time-stepping (PTS)

```
call cost_fun_b ( ←g, ←Nrm_b, 1 )
do nIter = 1,mIt
  call residual_u_b ( ←R_b, →U_b )
  R_b = R_b - g
  call update_b ( →R_b, ⇌U_b )
end do
call residual_nrm_b ( →U_b, ←Nrm_b )
call metrics_b ( ←X_b, →Nrm_b )
```

Adjoint fixed-point iterators with 'reverse accumulation'

Simplified compressible fixed-point iterator

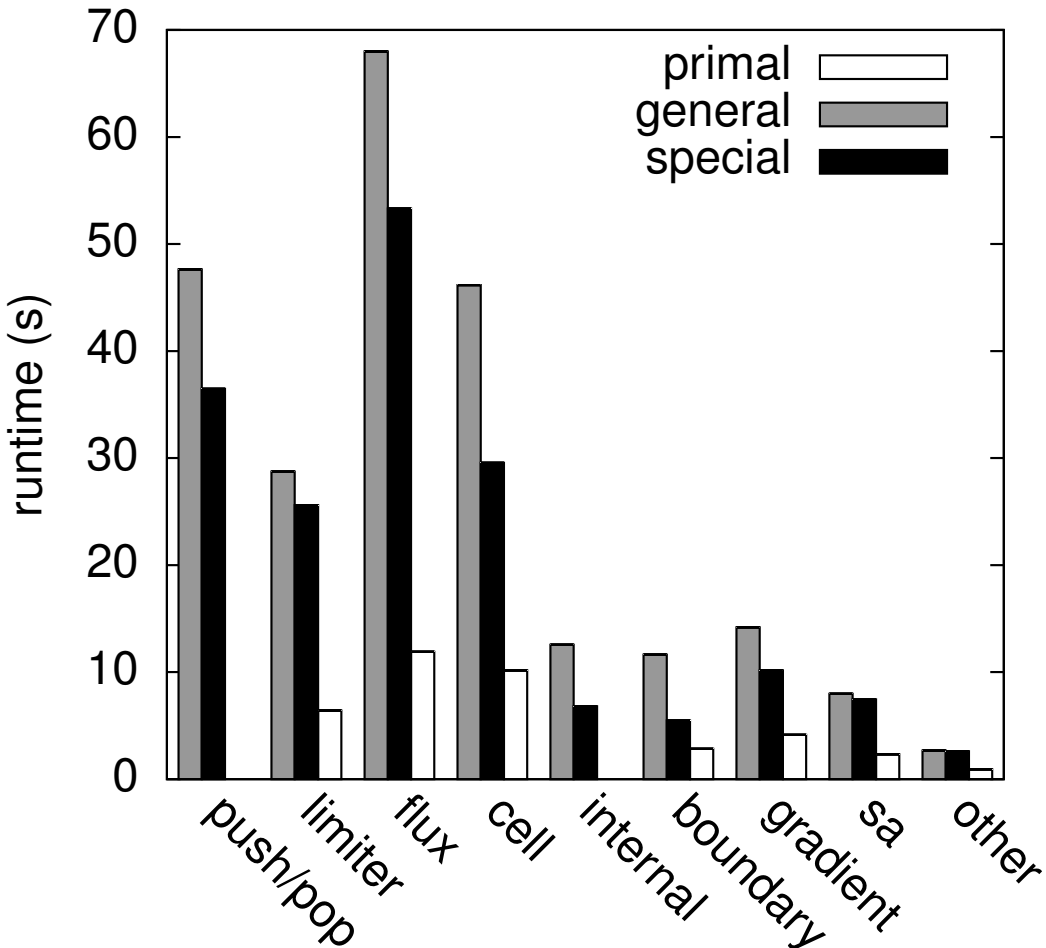
```
call initialise_flow ( ←U )
call metrics ( →X, ←Nrm )
do nIter = 1,mIt
  call residual ( →U, →Nrm, ←R )
  call update ( →R, ⇌U )
end do
call cost_fun ( →U, →Nrm, ←J )
```

Adjoint iterator derived from the primal time-stepping (PTS)

```
call cost_fun_b ( ←g, ←Nrm_b, 1 )
do nIter = 1,mIt
  call residual_u_b ( ←R_b, →U_b )
  R_b = R_b - g
  call update_b ( →R_b, ⇌U_b )
end do
call residual_nrm_b ( →U_b, ←Nrm_b )
call metrics_b ( ←X_b, →Nrm_b )
```

Multi-target AD

- Joint development with INRIA: create specialised derivative code depending on context
- Example: Residual comp. differentiated wrt `U` in iter. loop, and wrt `Nrm` via separate call upon loop exit.



CPU and memory performance of multi-target AD

Runtime and memory performance of general and specialised (multi-target) differentiation.

	runtime	runtime (rel.)	memory	memory (rel.)
primal	211.1s	1	360.93MB	1
general	328.8s	1.56	431.68MB	1.20
special	249.1s	1.18	432.62MB	1.20
change	**-32%**		**0.2%**	

Peak memory use (measured with valgrind/massif)

Case	flow Gb	adj. Gb	ratio
flatPlate, 2D quad, visc	0.217	0.260	1.20
rae2822, 2D quad, inv	0.199	0.231	1.16
DeathStar, 3D unstr, inv	0.331	0.368	1.12
TUB Stator, 3D hexa, visc	5.98	6.81	1.14

CPU and memory performance of multi-target AD

Runtime and memory performance of general and specialised (multi-target) differentiation.

	runtime	runtime (rel.)	memory	memory (rel.)
primal	211.1s	1	360.93MB	1
general	328.8s	1.56	431.68MB	1.20
special	249.1s	1.18	432.62MB	1.20
change	**-32%**		**0.2%**	

Peak memory use (measured with valgrind/massif)

Case	flow Gb	adj. Gb	ratio
flatPlate, 2D quad, visc	0.217	0.260	1.20
rae2822, 2D quad, inv	0.199	0.231	1.16
DeathStar, 3D unstr, inv	0.331	0.368	1.12
TUB Stator, 3D hexa, visc	5.98	6.81	1.14

Outline

The adjoint method

Introduction to Algorithmic Differentiation

STAMPS - Source-Transformation Adjoint Multi-Purpose Solver

CAD-based shape optimisation

Uncertainty Quantification

Summary

CAD parametrisation with gradient-based optimisation

To compute sensitivities of design variables α, complete with the chain rule. In tangent mode:

$$\frac{\partial J}{\partial \alpha} = \underbrace{\frac{\partial J}{\partial R}}_{\text{Tangent}} \underbrace{\frac{\partial R}{\partial X_V}}_{\text{Metrics}} \underbrace{\frac{\partial X_V}{\partial X_S}}_{\text{Mesh}} \underbrace{\frac{\partial X_S}{\partial \alpha}}_{\text{Parametrisation}}$$

Can be evaluated analytic, or with finite differences, AD, complex ...

Since $N_\alpha << N_S << N_V$, efficiency demands reverse differentiation:

$$\frac{\partial J}{\partial \alpha}^T = \underbrace{\frac{\partial X_S}{\partial \alpha}^T}_{\text{Parametrisation}} \underbrace{\frac{\partial X_V}{\partial X_S}^T}_{\text{Mesh}} \underbrace{\frac{\partial R}{\partial X_V}^T}_{\text{Metrics}} \underbrace{\frac{\partial J}{\partial R}^T}_{\text{Adjoint}}$$

Can be evaluated analytic, or with AD. Not with F-D, complex ..

CAD parametrisation with gradient-based optimisation

To compute sensitivities of design variables α, complete with the chain rule. In tangent mode:

$$\frac{\partial J}{\partial \alpha} = \underbrace{\frac{\partial J}{\partial R}}_{\text{Tangent}} \underbrace{\frac{\partial R}{\partial X_V}}_{\text{Metrics}} \underbrace{\frac{\partial X_V}{\partial X_S}}_{\text{Mesh}} \underbrace{\frac{\partial X_S}{\partial \alpha}}_{\text{Parametrisation}}$$

Can be evaluated analytic, or with finite differences, AD, complex ...

Since $N_\alpha << N_S << N_V$, efficiency demands reverse differentiation:

$$\frac{\partial J}{\partial \alpha}^T = \underbrace{\frac{\partial X_S}{\partial \alpha}^T}_{\text{Parametrisation}} \underbrace{\frac{\partial X_V}{\partial X_S}^T}_{\text{Mesh}} \underbrace{\frac{\partial R}{\partial X_V}^T}_{\text{Metrics}} \underbrace{\frac{\partial J}{\partial R}^T}_{\text{Adjoint}}$$

Can be evaluated analytic, or with AD. Not with F-D, complex ..

Challenges for CAD-based optimisation

- We need to compute the gradient of surface node movement w.r.t. design variable, $\frac{\partial X_S}{\partial \alpha}$.
- This derivative must be computed over all computational steps, typically a long list of successive operations.
- The chain of geometric computations contain many non-differentiable operations such as e.g. Booleans. Analytic derivatives may be difficult to define.
- None of the commercial CAD systems compute derivatives.

Challenges for CAD-based optimisation

- We need to compute the gradient of surface node movement w.r.t. design variable, $\frac{\partial X_S}{\partial \alpha}$.
- This derivative must be computed over all computational steps, typically a long list of successive operations.
- The chain of geometric computations contain many non-differentiable operations such as e.g. Booleans. Analytic derivatives may be difficult to define.
- None of the commercial CAD systems compute derivatives.

Challenges for CAD-based optimisation

- We need to compute the gradient of surface node movement w.r.t. design variable, $\frac{\partial X_S}{\partial \alpha}$.
- This derivative must be computed over all computational steps, typically a long list of successive operations.
- The chain of geometric computations contain many non-differentiable operations such as e.g. Booleans. Analytic derivatives may be difficult to define.
- None of the commercial CAD systems compute derivatives.

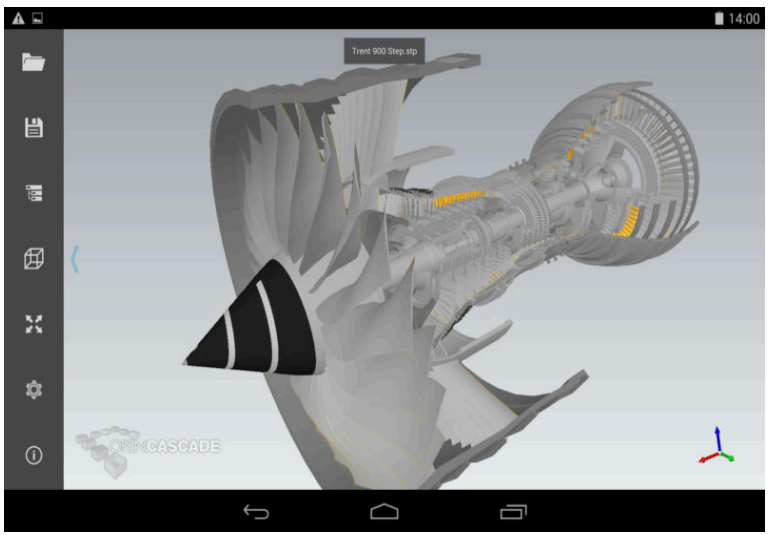
Challenges for CAD-based optimisation

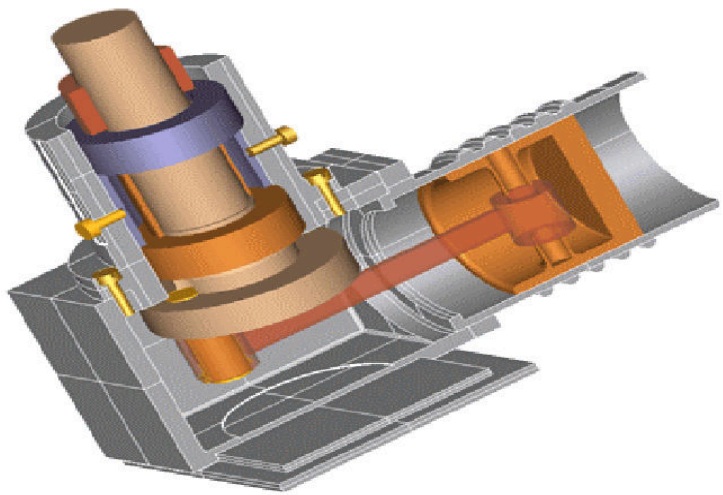
- We need to compute the gradient of surface node movement w.r.t. design variable, $\frac{\partial X_S}{\partial \alpha}$.
- This derivative must be computed over all computational steps, typically a long list of successive operations.
- The chain of geometric computations contain many non-differentiable operations such as e.g. Booleans. Analytic derivatives may be difficult to define.
- None of the commercial CAD systems compute derivatives.

Open Cascade Technology

Open CASCADE[3] Technology (OCCT) is an open source C++ library, consisting of thousands of classes and providing solutions in the areas of:

- Surface and solid modelling: to model any type of object,
- 3D and 2D visualisation: to display and animate objects,
- Data exchange (import and export standard CAD formats) and tree-like data model.





AD on Open Cascace CAD kernel (scalar)

Joint work with Prof. A.Walther, M. Banovic, Paderborn Univ.

Automatic **D**ifferentiation by **O**ver**L**oading in **C++**

ADOL-C uses **operator overloading** concept to compute first and higher derivatives of vector functions that are written in C or C++.

Operator overloading (Scalar mode)

```
1   class myadouble{
2     double value;
3     double ADvalue;
4     // multiplication ...
5     inline myadouble operator * (const myadouble& a) const {
6       myadouble tmp;
7       tmp.value = value * a.value;
8       tmp.ADvalue = ADvalue * a.value + value * a.ADvalue;
9       return tmp;
10    }
11  };
```

AD on Open Cascace CAD kernel (vector)

Joint work with Prof. A.Walther, M. Banovic, Paderborn Uni.

Automatic **D**ifferentiation by **O**ver**L**oading in **C++**

ADOL-C uses **operator overloading** concept to compute first and higher derivatives of vector functions that are written in C or C++.

Operator overloading (**Vector mode)**

```
1       class myadouble{
2         double value;
3         double *ADvalue = new double[NUMBER_OF_DIRECTIONS];
4         // multiplication ...
5         inline myadouble operator * (const myadouble& a) const {
6           myadouble tmp;
7           tmp.value = value * a.value;
8           for(size_t i = 0; i < NUMBER_OF_DIRECTIONS; ++i)
9           tmp.ADvalue[i] = ADvalue[i] * a.value + value * a.
                ADvalue[i];
10          return tmp;
11        }
12      };
```

Calculating the CAD sensitivities (II)

ADOL-C vector mode

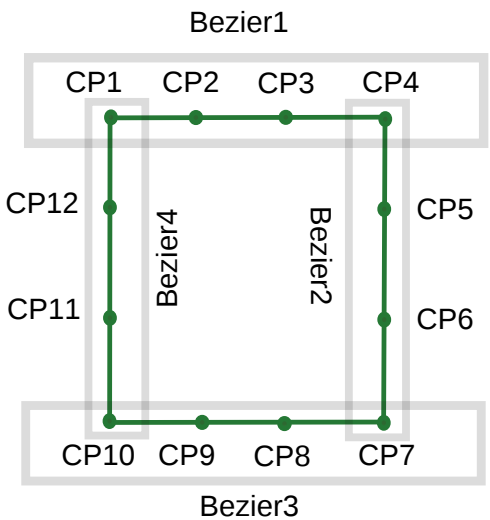
- Number of design parameters = `NUMBER_OF_DIRECTIONS`.
- Derivaties w.r.t. all design parameters are evaluated with just one code run.
- Computational cost and memory requirements are dependent on the `NUMBER_OF_DIRECTIONS`.

Example of sensitivities calculated using AD vector mode

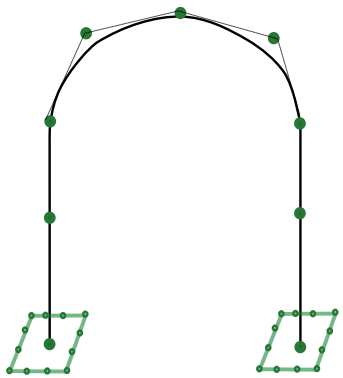
Explicit parametrisation for turbo-machinery U-Bend

The generic slice consists of 4 Bézier curves; it is swept orthogonally along a planar pathline.

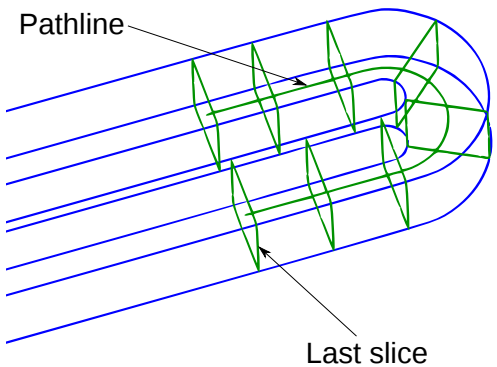
Section/Slice Definition



Pathline BSpline

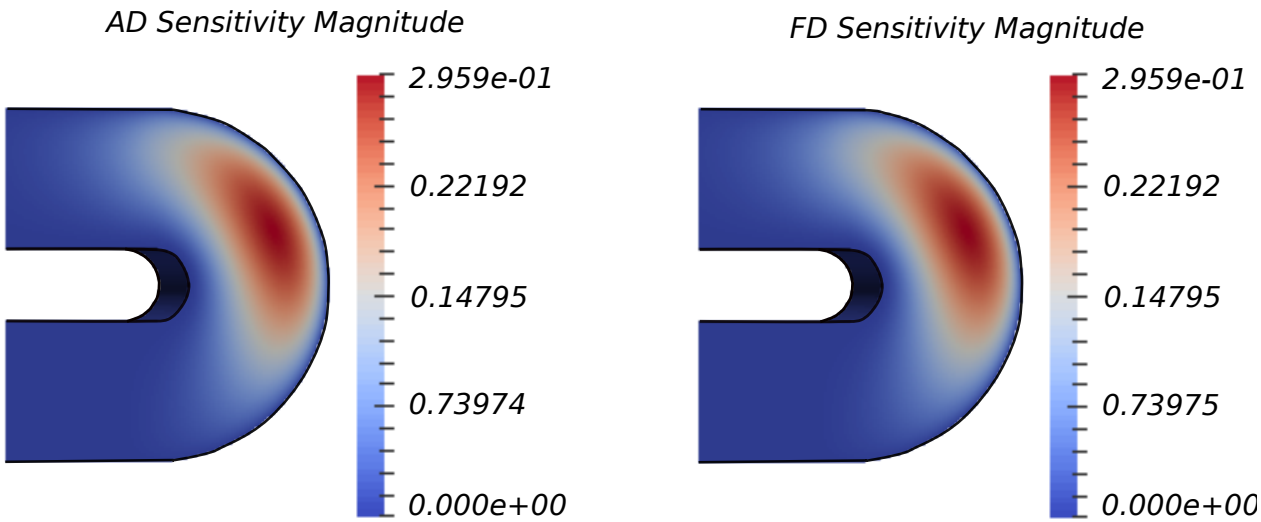


OCCT ThruSection Algo

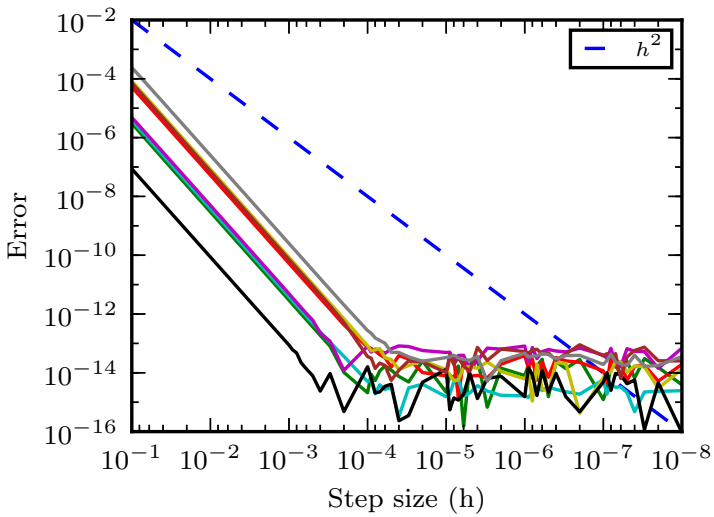


- Each control point of the section is defined by a law of evolution along a B-spline curve along the pathline.
- The 96 coordinates of its control points are the **design parameters**.

Gradient comparison

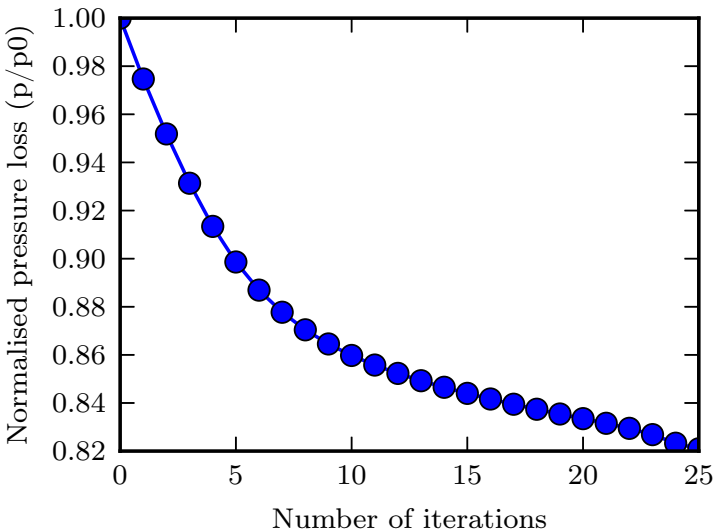


U-part shape sensitivities using AD (left), F-D (right).

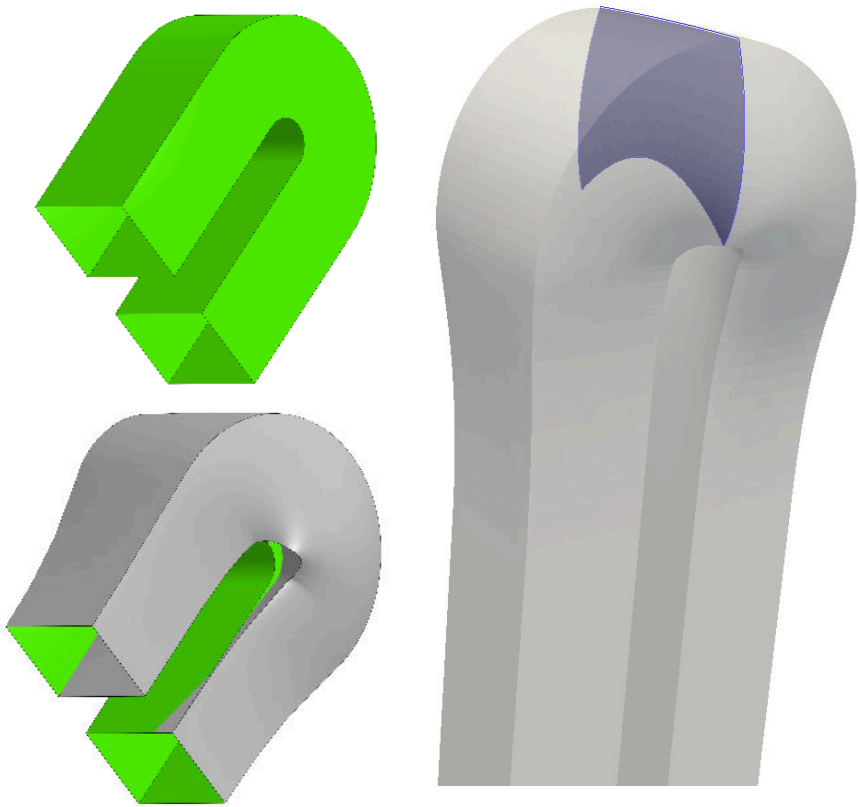


Taylor test (extrapolation of AD gradients vs. undivided difference) for eight U-bend surface point coordinates

U-Bend optimisation: optimised shape

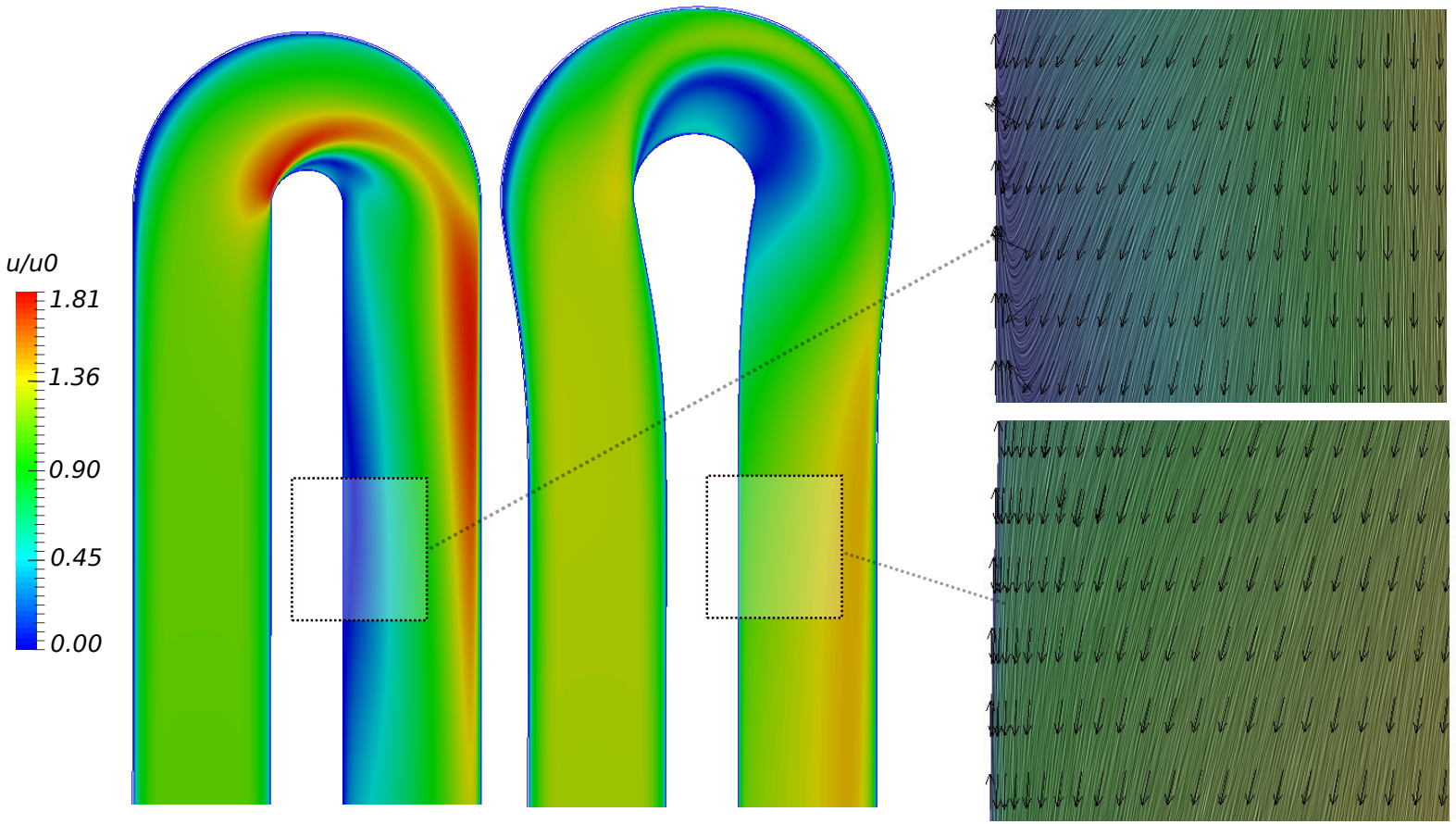


Optimisation algorithm: BFGS.



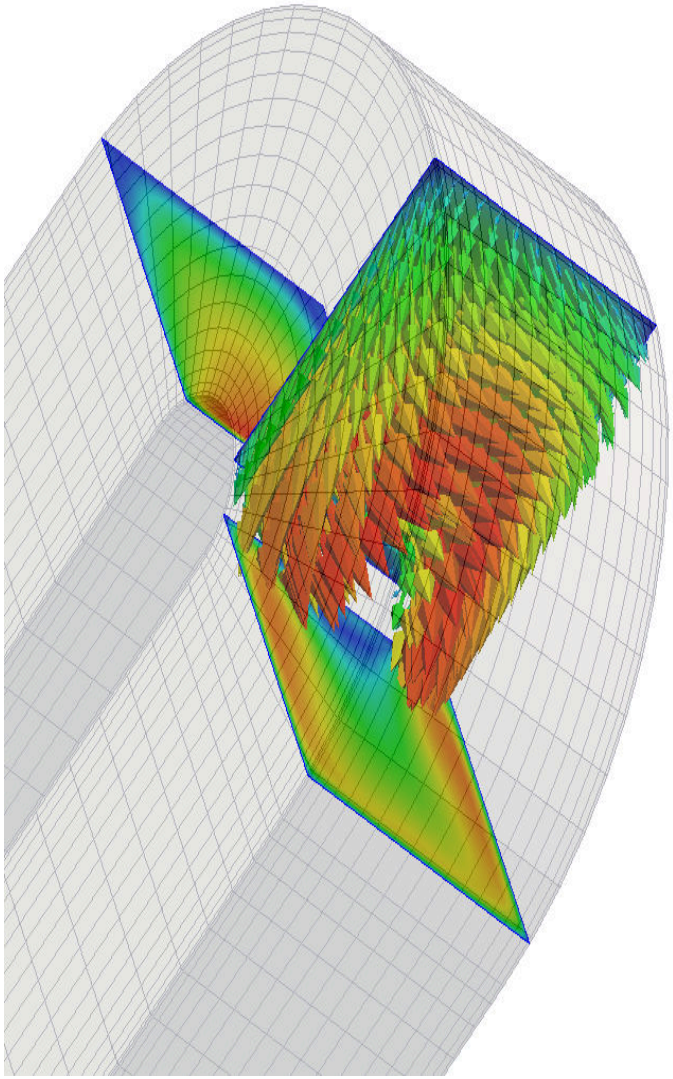
Initial and final shape.

U-Bend optimisation, flowfield

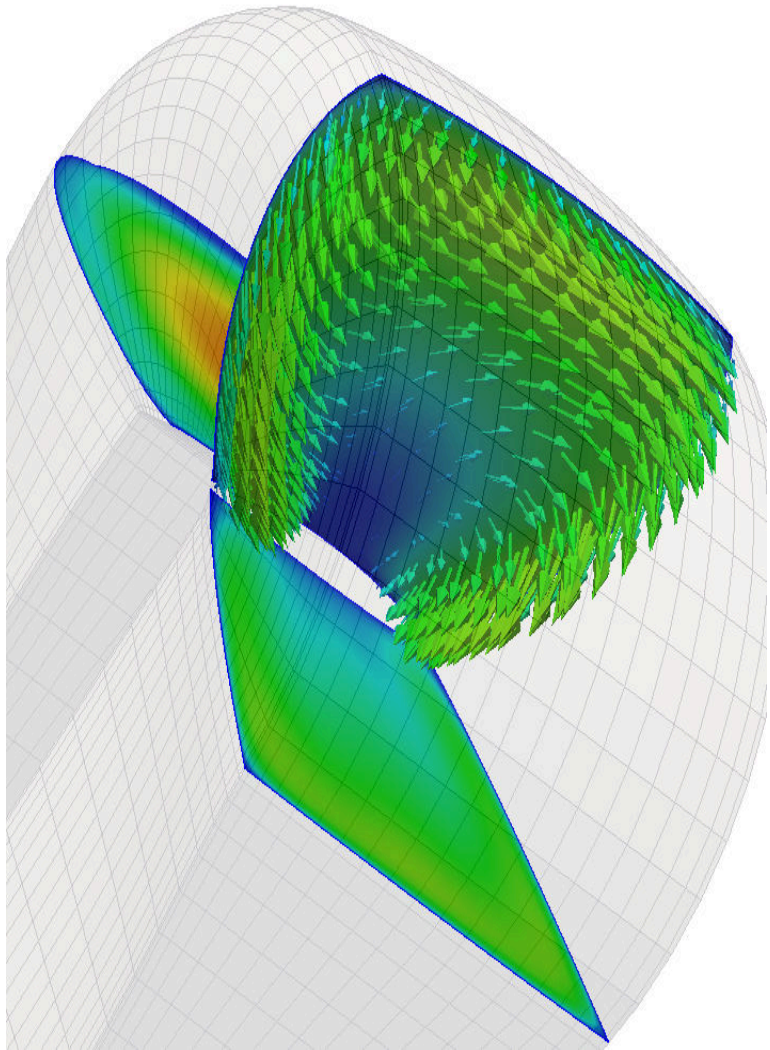


Left: Baseline and Optimised Mid-span Velocity Magnitude;
Right: Flow Streamlines in the outlet leg

U-Bend optimisation: velocity vectors at mid-turn

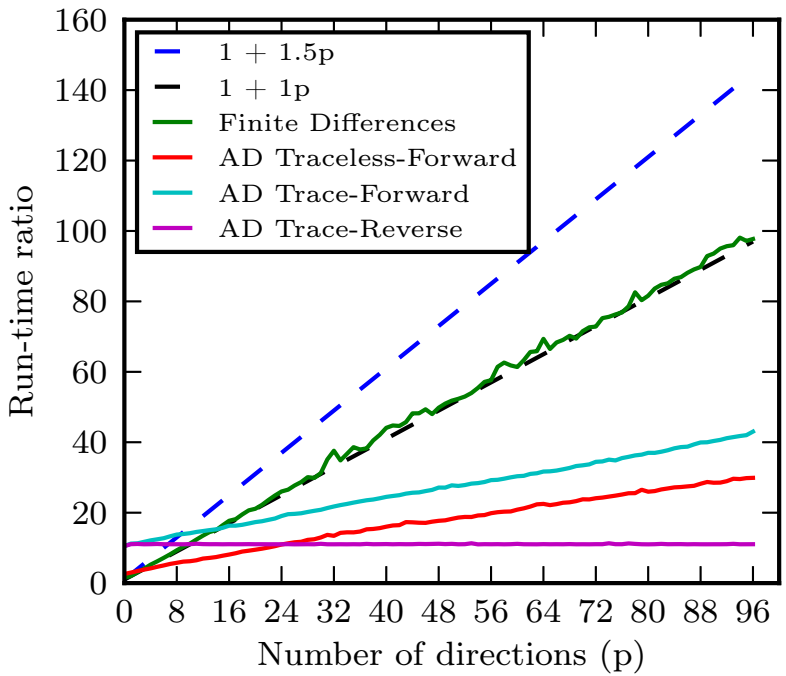


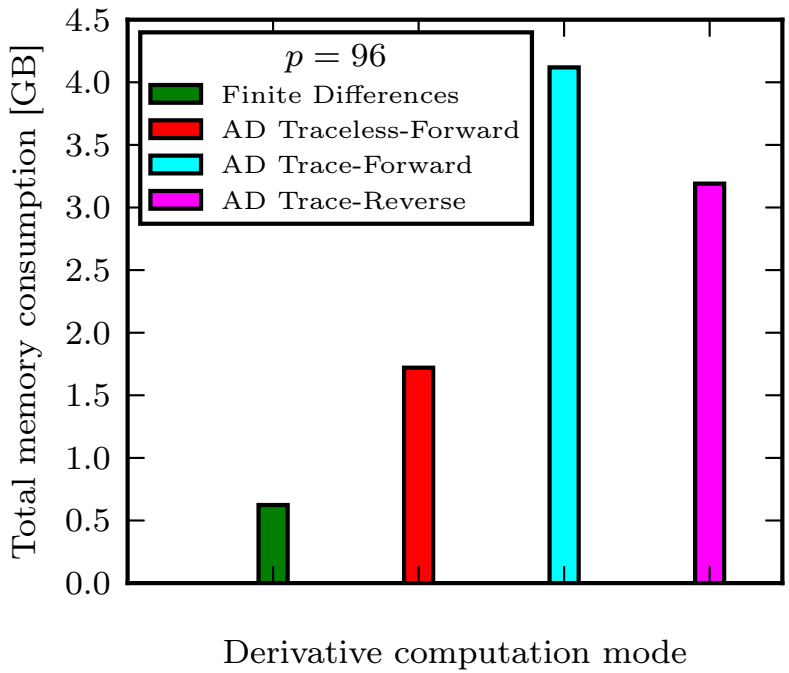
Baseline



Optimised

Computational cost: runtime and memory

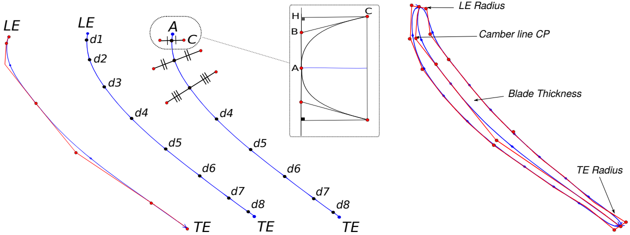




Summary of run-time ratios (left) and total memory requirements (right) for U-bend example

Explicit Parametrisation for TUB Stator

2-D blade: camberline → le/te radii → pressure/suction curve

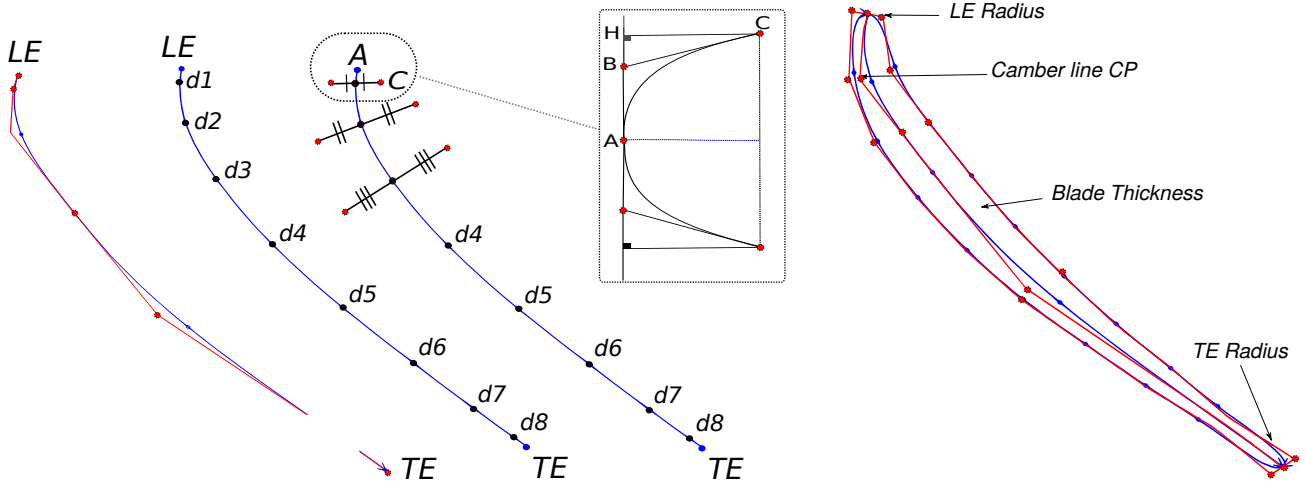


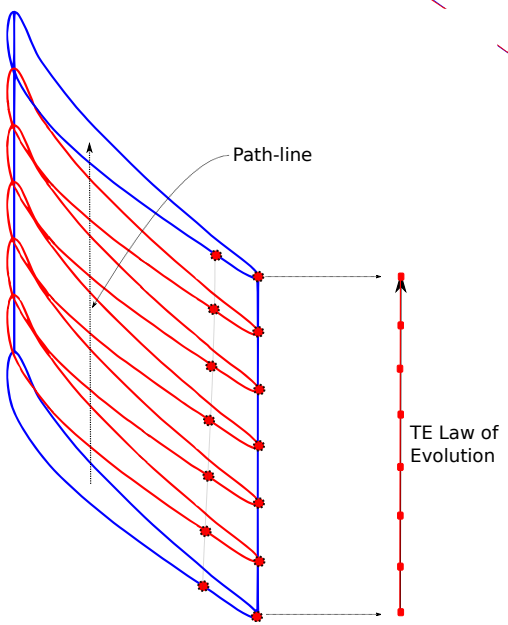
3-D blade: loft 2-D blade using laws of evolution along spline curves

- Design variables are CP of our parametrised curves and laws of evolution, 23*8=184 DoF
- We can control thickness with bounding values

Explicit Parametrisation for TUB Stator

2-D blade: camberline → le/te radii → pressure/suction curve



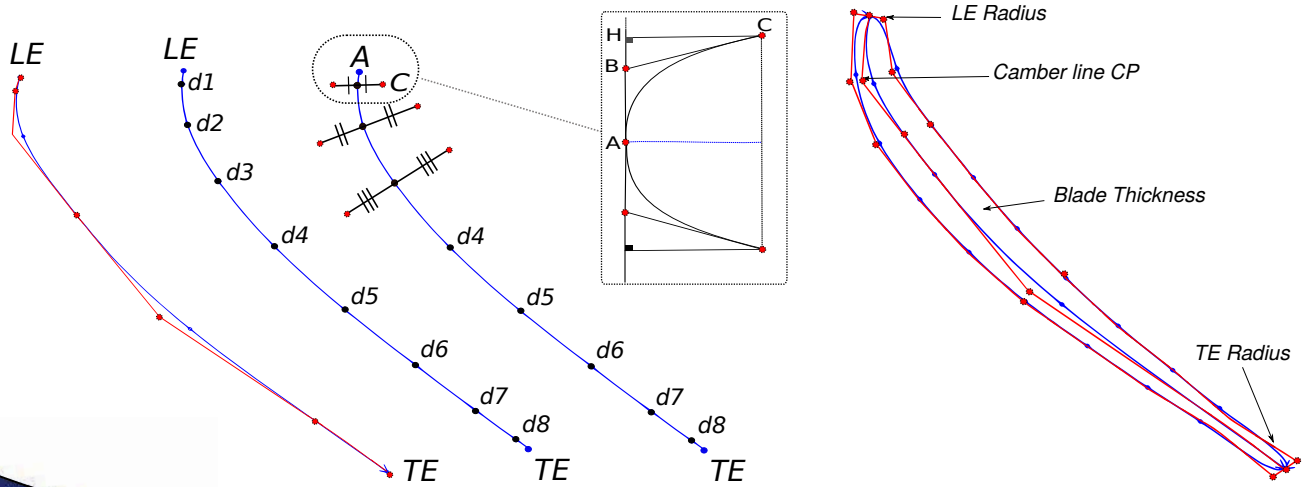


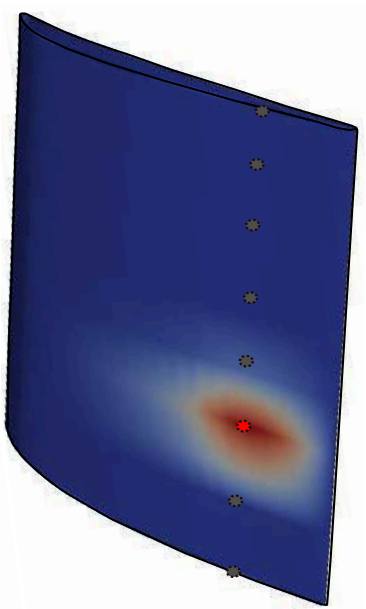
3-D blade: loft 2-D blade using laws of evolution along spline curves

- Design variables are CP of our parametrised curves and laws of evolution, 23*8=184 DoF
- We can control thickness with bounding values

Explicit Parametrisation for TUB Stator

2-D blade: camberline → le/te radii → pressure/suction curve



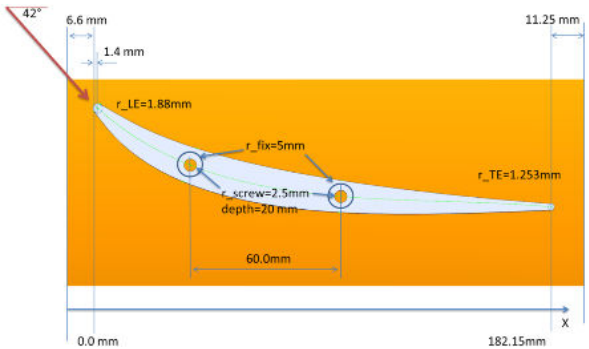


3-D blade: loft 2-D blade using laws of evolution along spline curves

- Design variables are CP of our parametrised curves and laws of evolution, 23*8=184 DoF
- We can control thickness with bounding values

Differentiating assembly constraints

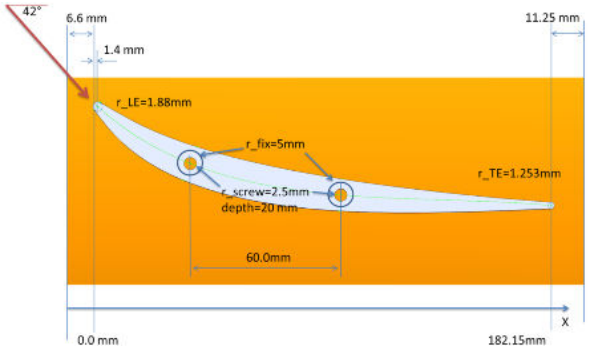
Benchmark testcase requires space for mounting bolts inside the blade.



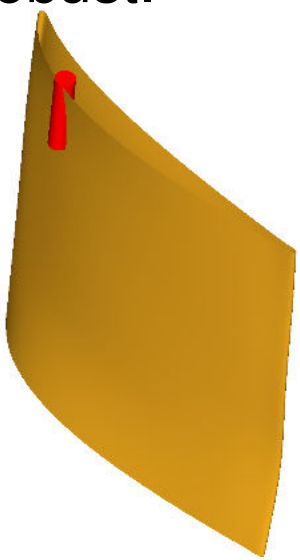
A smooth constraint objective is to compute the volume of the bolt cylinder outside the blade, but the geometric algorithm is non-robust.

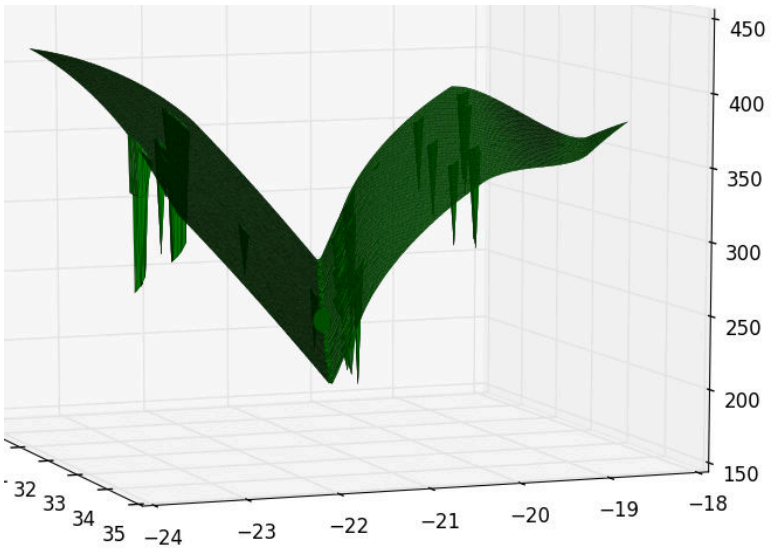
Differentiating assembly constraints

Benchmark testcase requires space for mounting bolts inside the blade.



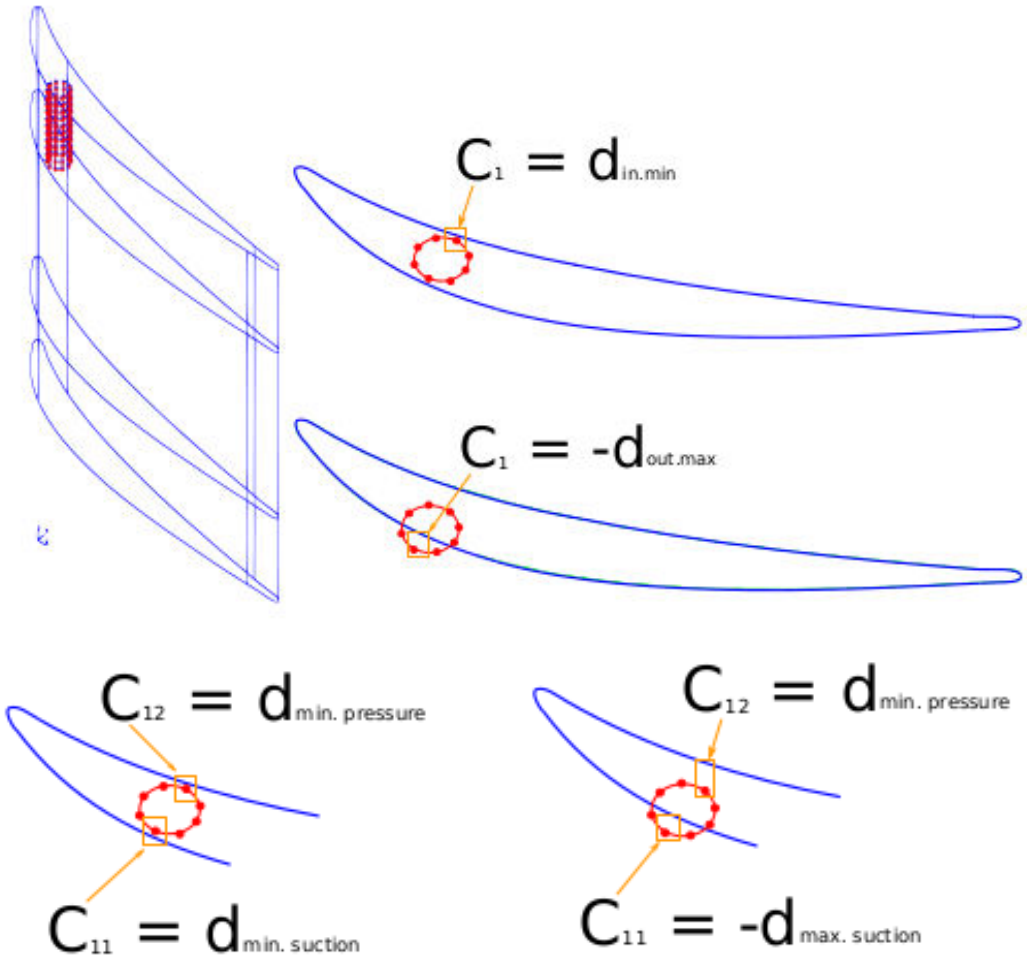
A smooth constraint objective is to compute the volume of the bolt cylinder outside the blade, but the geometric algorithm is non-robust.



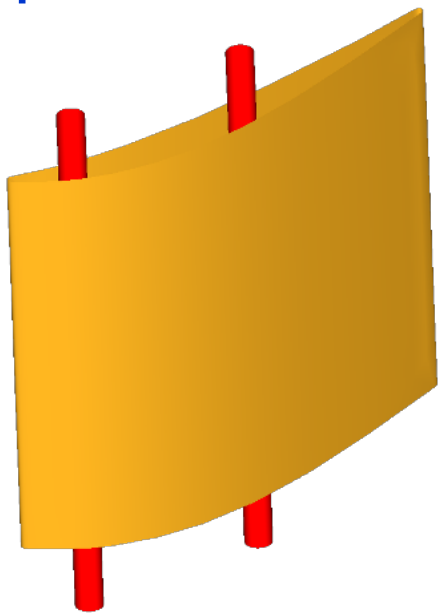


Differentiating assembly constraints

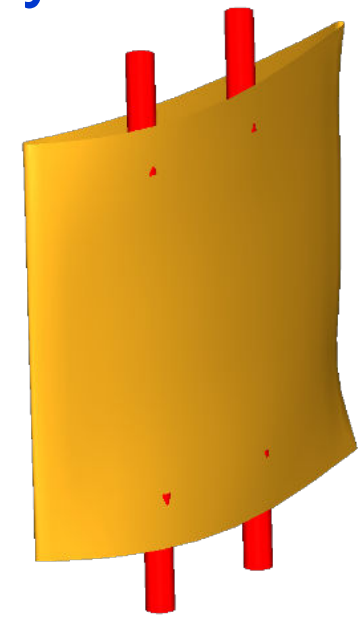
Alternative: evaluate a signed distance to the blade of sampled points on the cylinder



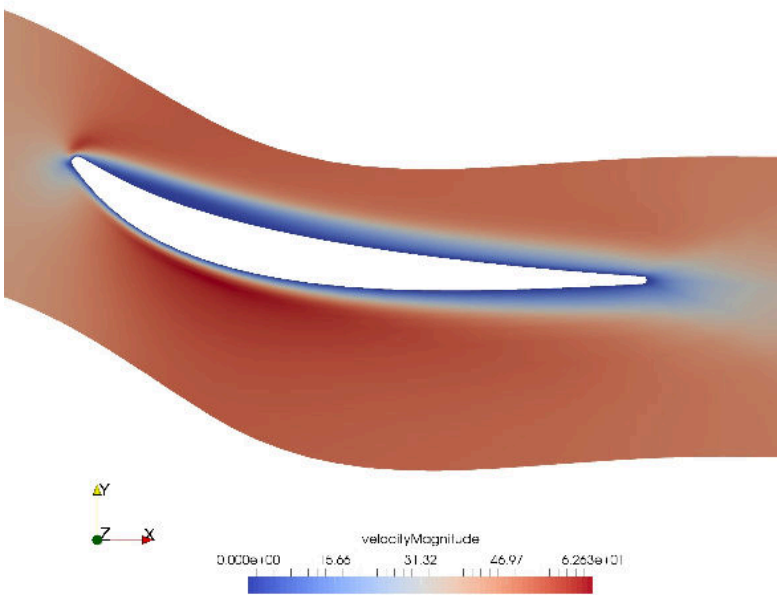
Optimised blade with assembly constraints

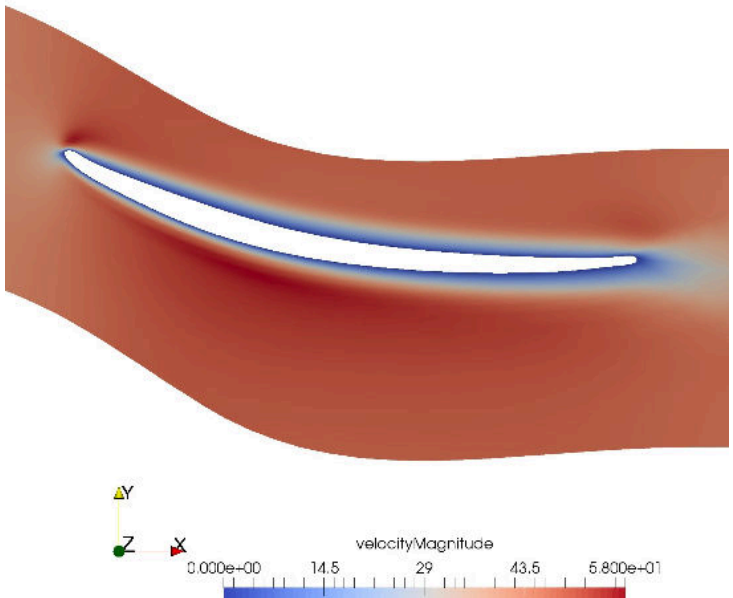


initial



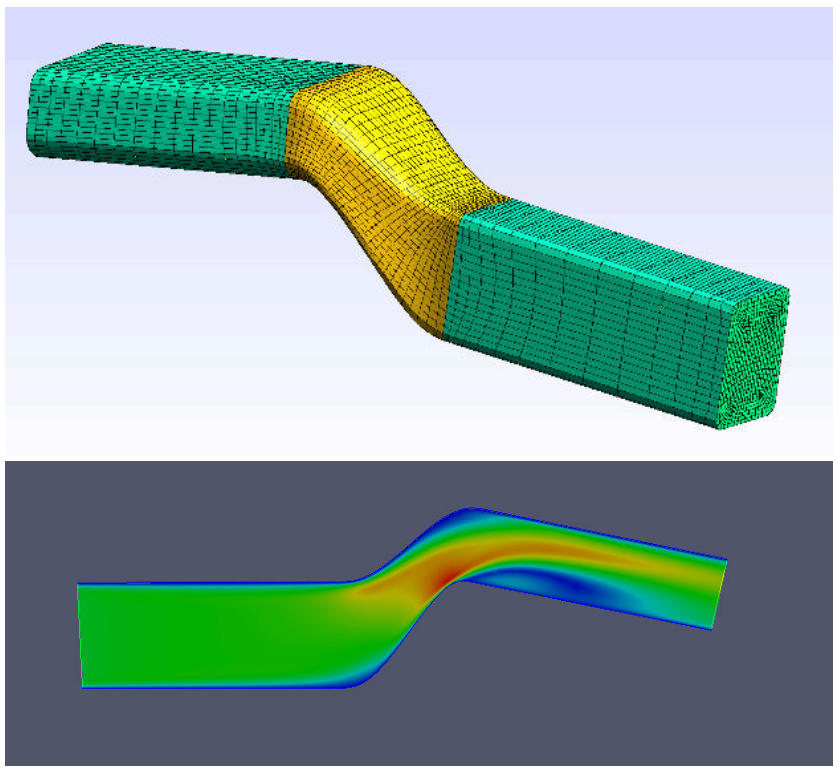
optimised



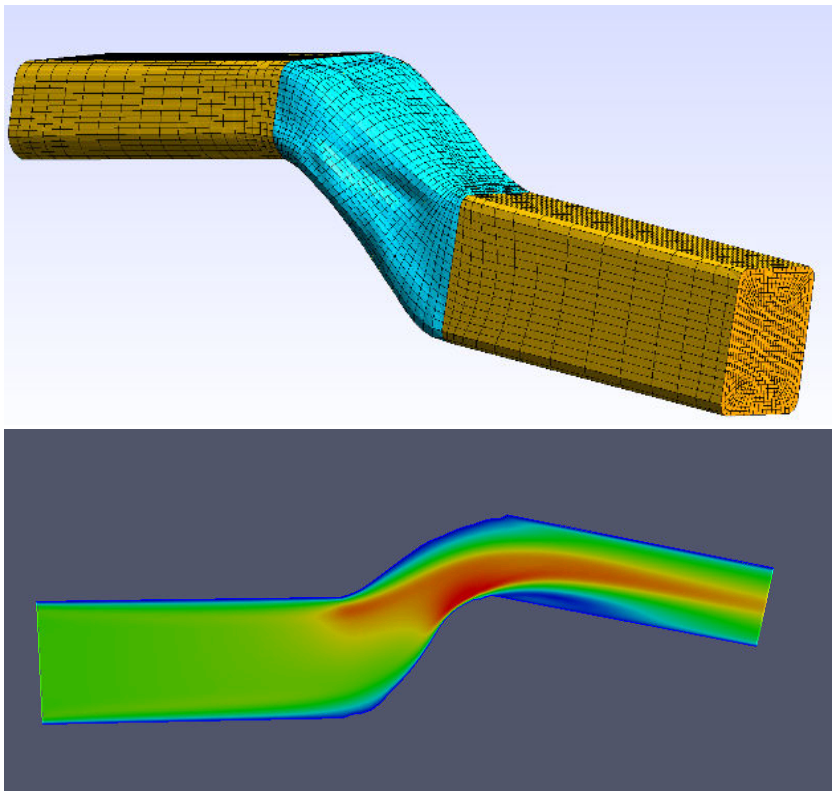


Node-based optimisation: mesh and velocity magnitude

Initial S-bend



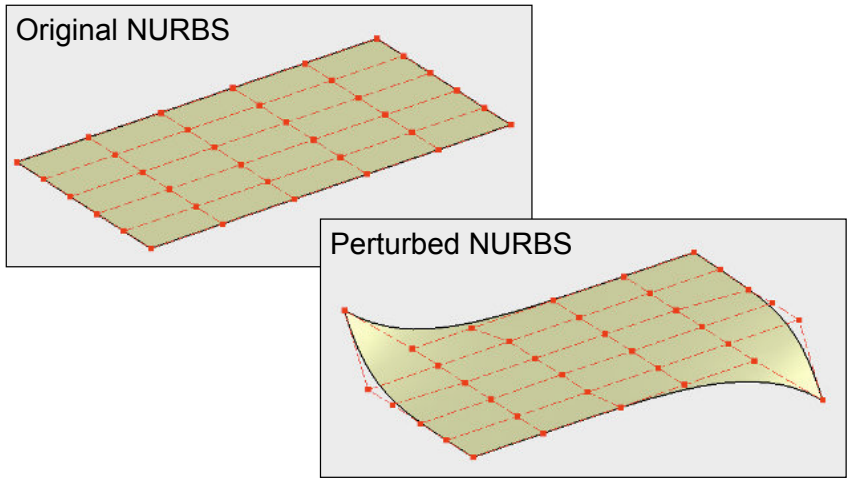
Node-based optimisation



Unintuitive shape changes, how to capture this for CAD? How to impose geometrical constraints?

CAD-based parametrisation using NURBS

To integrate num. optimisation into the design chain, the optimised shape must exist in a CAD format.



- NURBS-based approach: use the position of the control points and weights of the NURBS as design variables.
- The difficulty is to maintain geometric continuity across patch interfaces for complex CAD geometries.
- QMUL group developed the NSPCC methodology to ensure G0 (watertight), G1 (tangency) or G2 (curvature) continuity.

(NSPCC: NURBS-based Parametrisation with Complex Constraints)

NSPCC: NURBS-based parametrisation with complex constraints

When using more than one or unconstrained patches, the parametrisation needs to ensure that patches have G0 (watertight), G1 (tangency) or G2 (curvature) continuity:

- Standard methods define geometric constraints on control point positions, e.g. first control point inside needs to be in the tangency plane for G1.
- This is not practical for general cases.

Alternatively:

- iterate on control point positions to satisfy constraints
- evaluate constraints at test-points distributed along patch boundaries/intersections.

NSPCC: NURBS-based parametrisation with complex constraints

When using more than one or unconstrained patches, the parametrisation needs to ensure that patches have G0 (watertight), G1 (tangency) or G2 (curvature) continuity:

- Standard methods define geometric constraints on control point positions, e.g. first control point inside needs to be in the tangency plane for G1.
- This is not practical for general cases.

Alternatively:

- iterate on control point positions to satisfy constraints
- evaluate constraints at test-points distributed along patch boundaries/intersections.

NSPCC: NURBS-based parametrisation with complex constraints

When using more than one or unconstrained patches, the parametrisation needs to ensure that patches have G0 (watertight), G1 (tangency) or G2 (curvature) continuity:

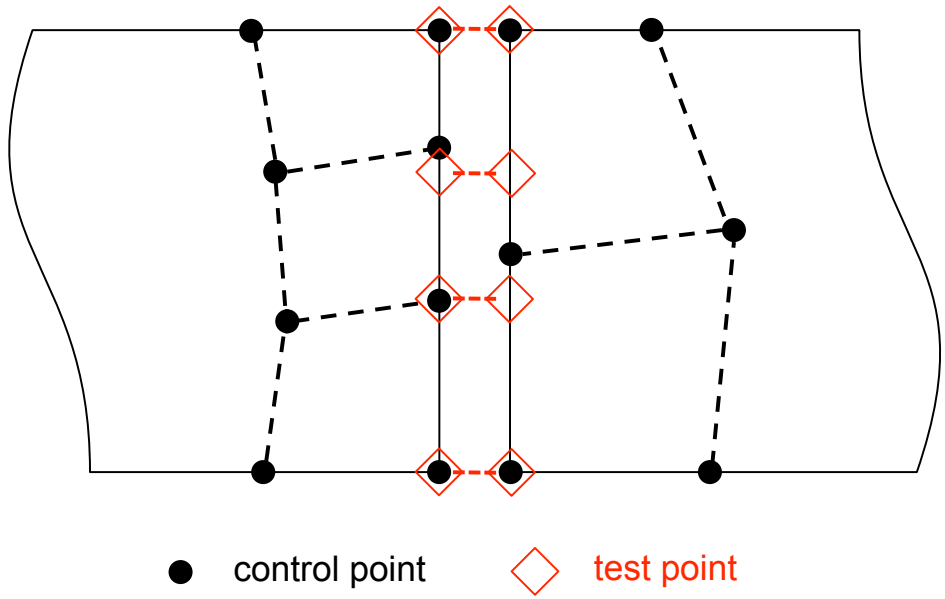
- Standard methods define geometric constraints on control point positions, e.g. first control point inside needs to be in the tangency plane for G1.
- This is not practical for general cases.

Alternatively:

- iterate on control point positions to satisfy constraints
- evaluate constraints at test-points distributed along patch boundaries/intersections.

Continuity constraint

- Continuity is required along joint edges between patches



- Constraint functions are evaluated at test points

$$G_0 = (X_s)_L - (X_s)_R \text{ linear}$$

$$G_1 = (\vec{\tau})_L \times (\vec{\tau})_R \text{ non-linear}$$

$$G_2 = (\vec{k})_L - (\vec{k})_R \text{ non-linear}$$

and are required to maintain their initial values at each design step.

Continuity constraint II

Constraint functions are linearised w.r.t. control points

$$\delta G = \frac{\partial G}{\partial P}\delta P + \text{h.o.t.}$$

Let the perturbation δP lie in the null-space of $\partial G/\partial P$,

$$\delta P = \ker C \cdot \delta\alpha$$

A basis $\ker C$ for the null-space of C is computed using SVD.

G_1 and G_2 will only be approximately zero after perturbation, then the range space is used to take a normal step to recover the non-linear constraints.

Continuity constraint II

Constraint functions are linearised w.r.t. control points

$$\delta G = \frac{\partial G}{\partial P}\delta P + \text{h.o.t.}$$

Let the perturbation δP lie in the null-space of $\partial G/\partial P$,

$$\delta P = \ker C \cdot \delta\alpha$$

A basis $\ker C$ for the null-space of C is computed using SVD.

G_1 and G_2 will only be approximately zero after perturbation, then the range space is used to take a normal step to recover the non-linear constraints.

Continuity constraint II

Constraint functions are linearised w.r.t. control points

$$\delta G = \frac{\partial G}{\partial P}\delta P + \text{h.o.t.}$$

Let the perturbation δP lie in the null-space of $\partial G/\partial P$,

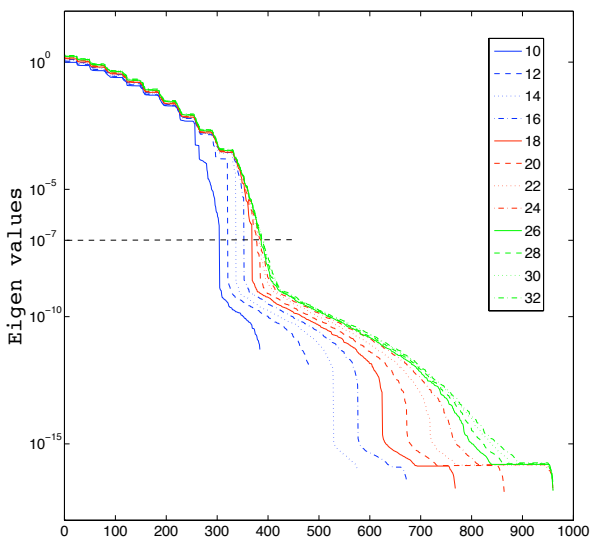
$$\delta P = \ker C \cdot \delta\alpha$$

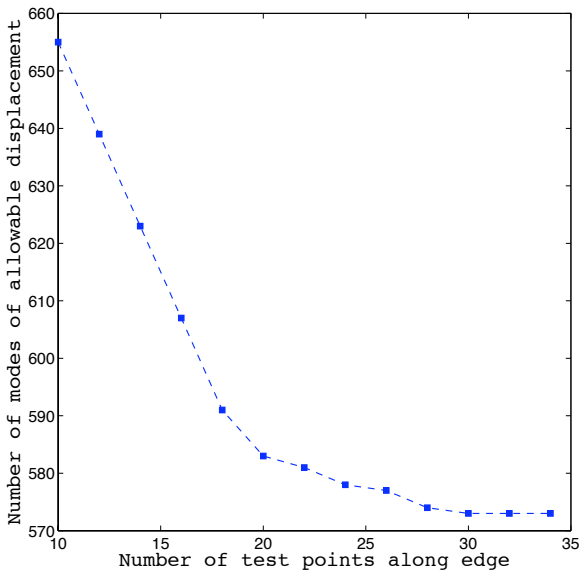
A basis $\ker C$ for the null-space of C is computed using SVD.

G_1 and G_2 will only be approximately zero after perturbation, then the range space is used to take a normal step to recover the non-linear constraints.

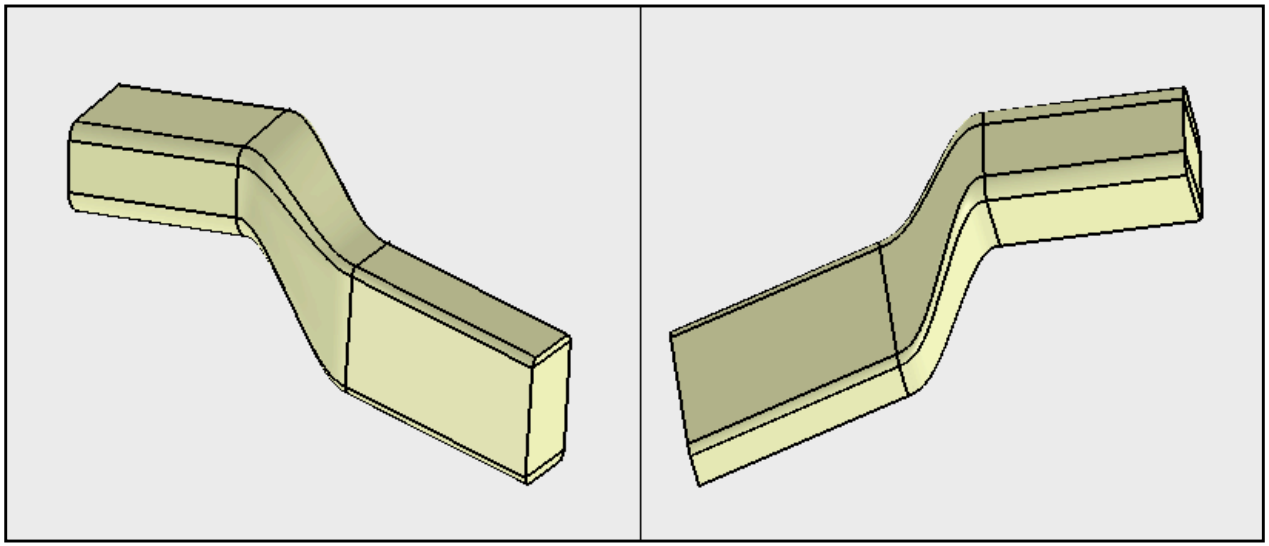
Continuity constraint III

- The number of non-zero singular values determines the design space: the SVD computes an orthogonal basis, which significantly improves convergence rate
- The cut-off value for ‘zero’ singular allows to select the size of the design space.



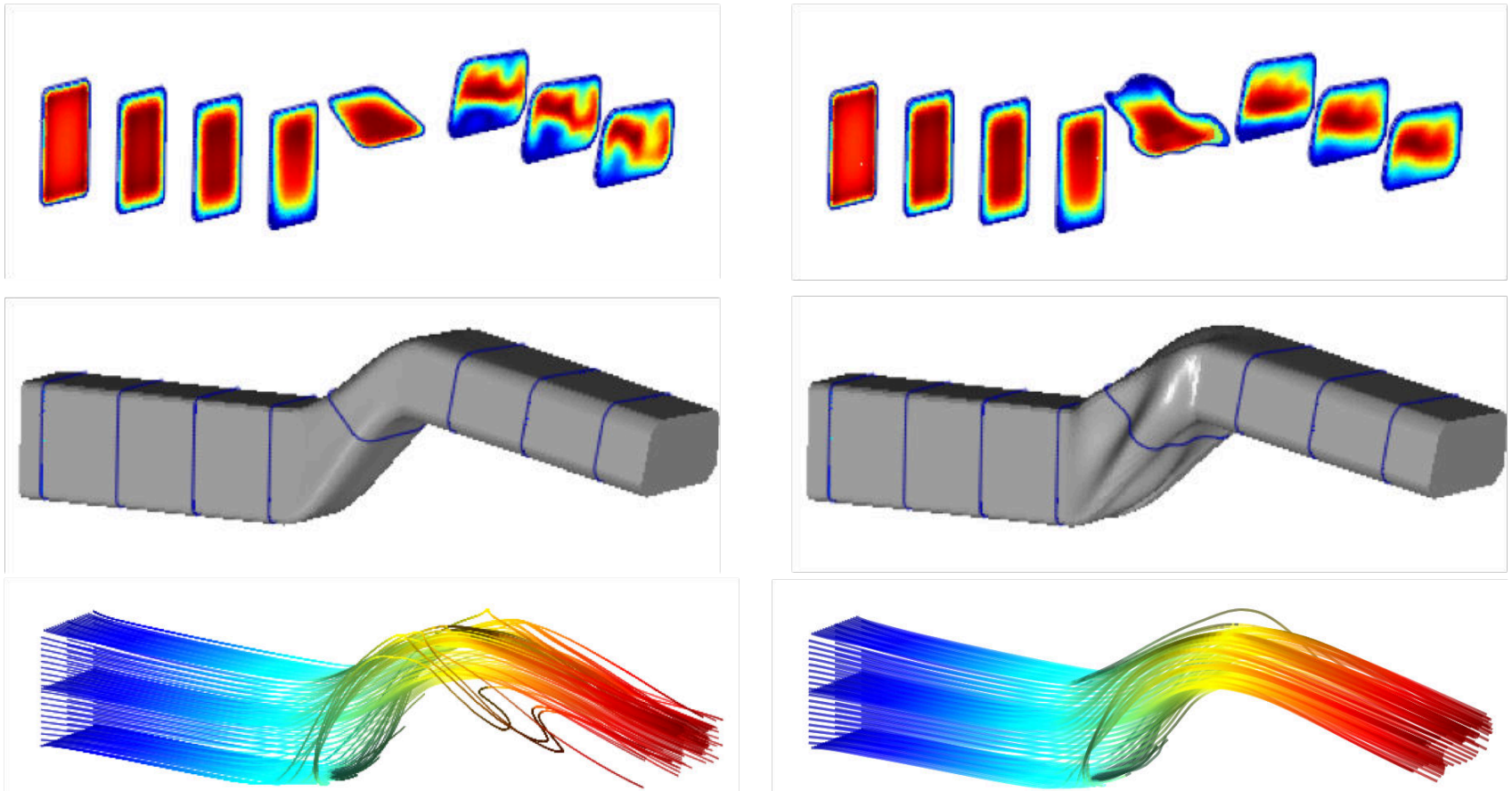


Optimisation of a Volkswagen S-Bend climate duct



- 30 NURBS patches, only cranked centre section is allowed to move. 640 control points, 1920 DoF.
- Imposing G1 continuity constraints leaves a design space of 570 modes.
- Objective function: minimise pressure loss from inlet to outlet.

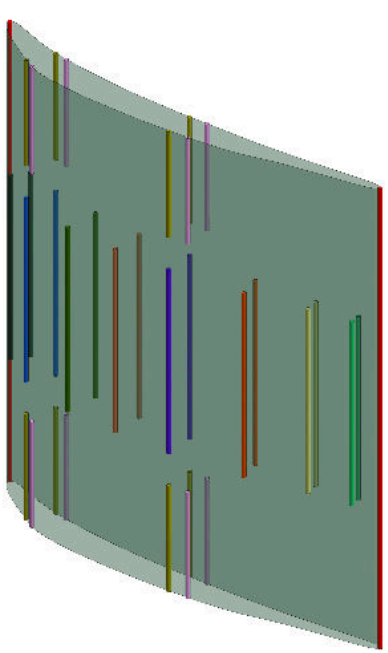
S-Bend: Improvement in flow field

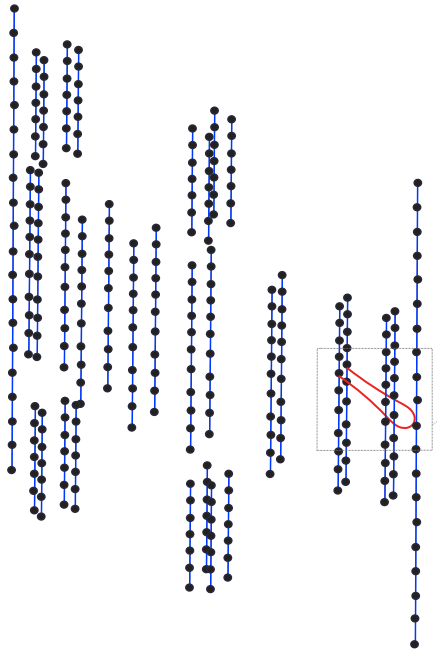


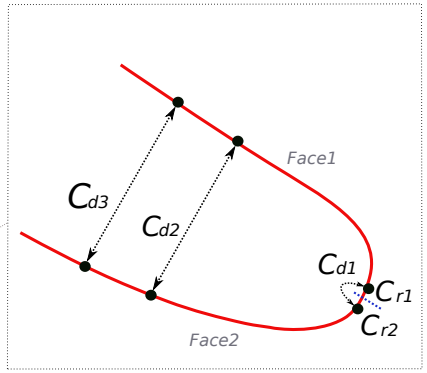
The optimisation produces strake-like features which very effectively suppress secondary flow motion.

Implicit Parametrisation in OCCT using NURBS patches

Geometric constraints evaluated at test points







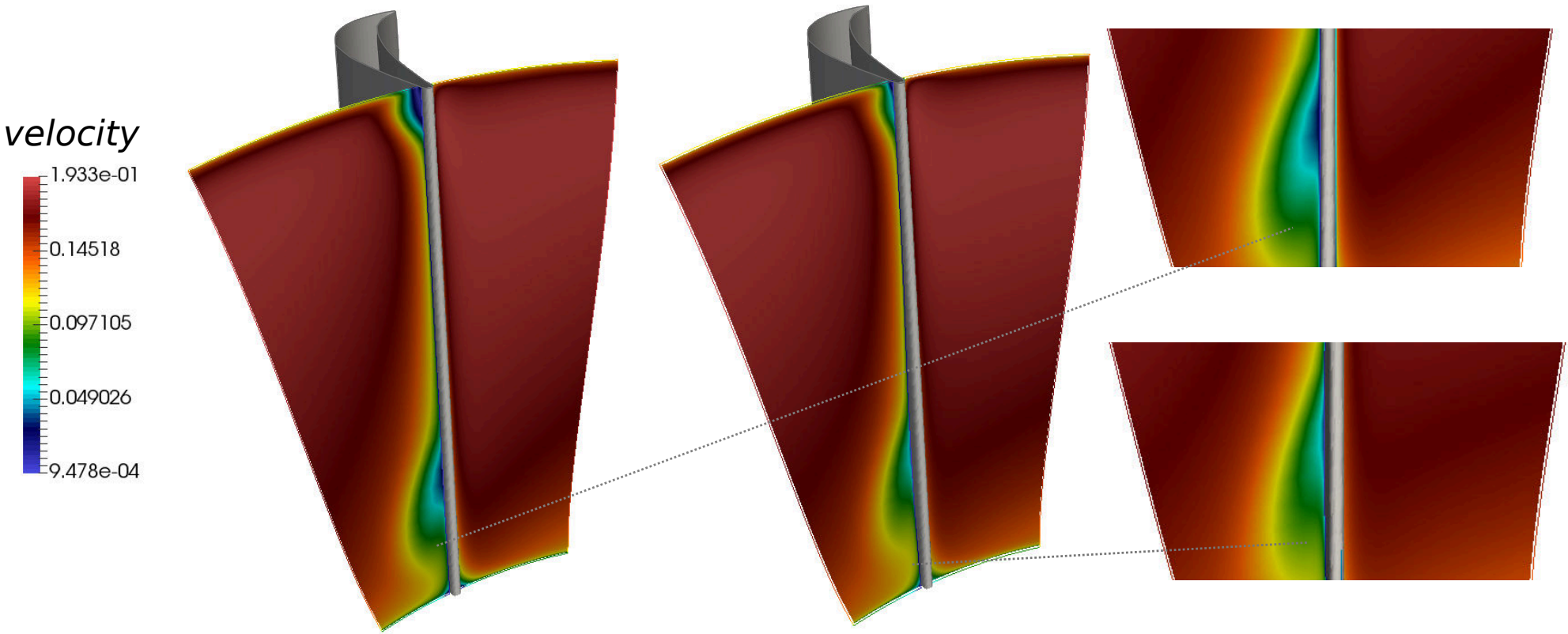
Control Points Updates and Constraints Matrix (SVD)

$$C = \frac{\partial C_x}{\partial P} = \begin{pmatrix} \frac{\partial C_{d1}}{\partial P_1} & \frac{\partial C_{d2}}{\partial P_1} & \cdots & \frac{\partial C_{cr1}}{\partial P_1} & \frac{\partial C_{cr2}}{\partial P_1} & \cdots & \frac{\partial C_{cr}}{\partial P_1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial C_{d1}}{\partial P_N} & \frac{\partial C_{d2}}{\partial P_N} & \cdots & \frac{\partial C_{cr1}}{\partial P_N} & \frac{\partial C_{cr2}}{\partial P_N} & \cdots & \frac{\partial C_{cr}}{\partial P_N} \end{pmatrix} \quad (1)$$

✓Constraints $P^{n+1} = P^n + t \cdot \ker(C) \left[(\nabla J) \ker(C) \right]^T \quad (6)$

TUB Stator Optimisation results using NSPCC parametrisation

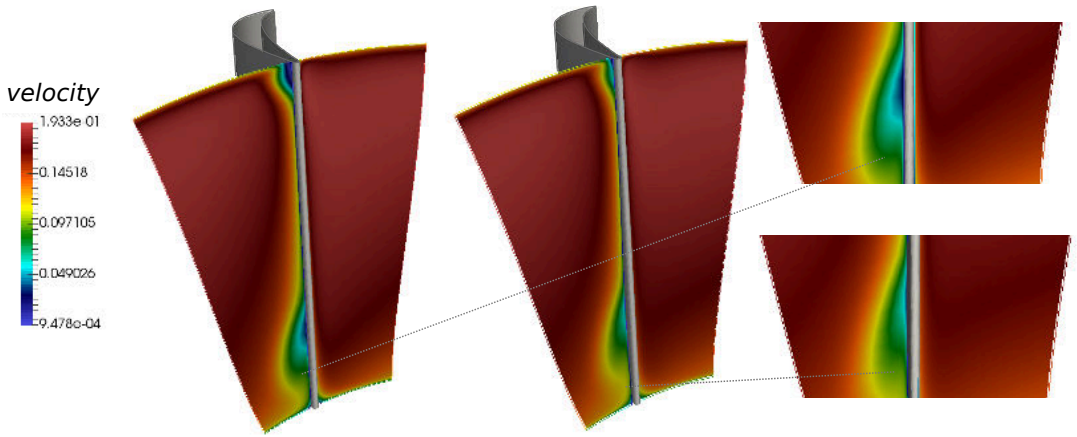
Optimised CFD flow, block-structured mesh, 0.4M hexahedra.



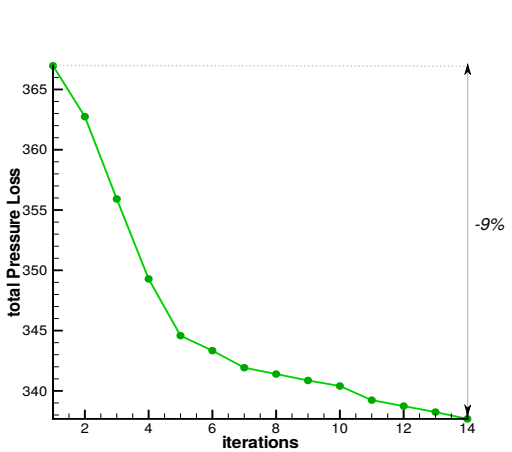
Optimisation History

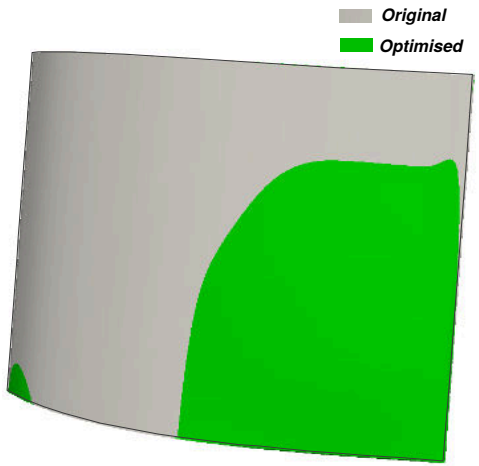
TUB Stator Optimisation results using NSPCC parametrisation

Optimised CFD flow, block-structured mesh, 0.4M hexahedra.



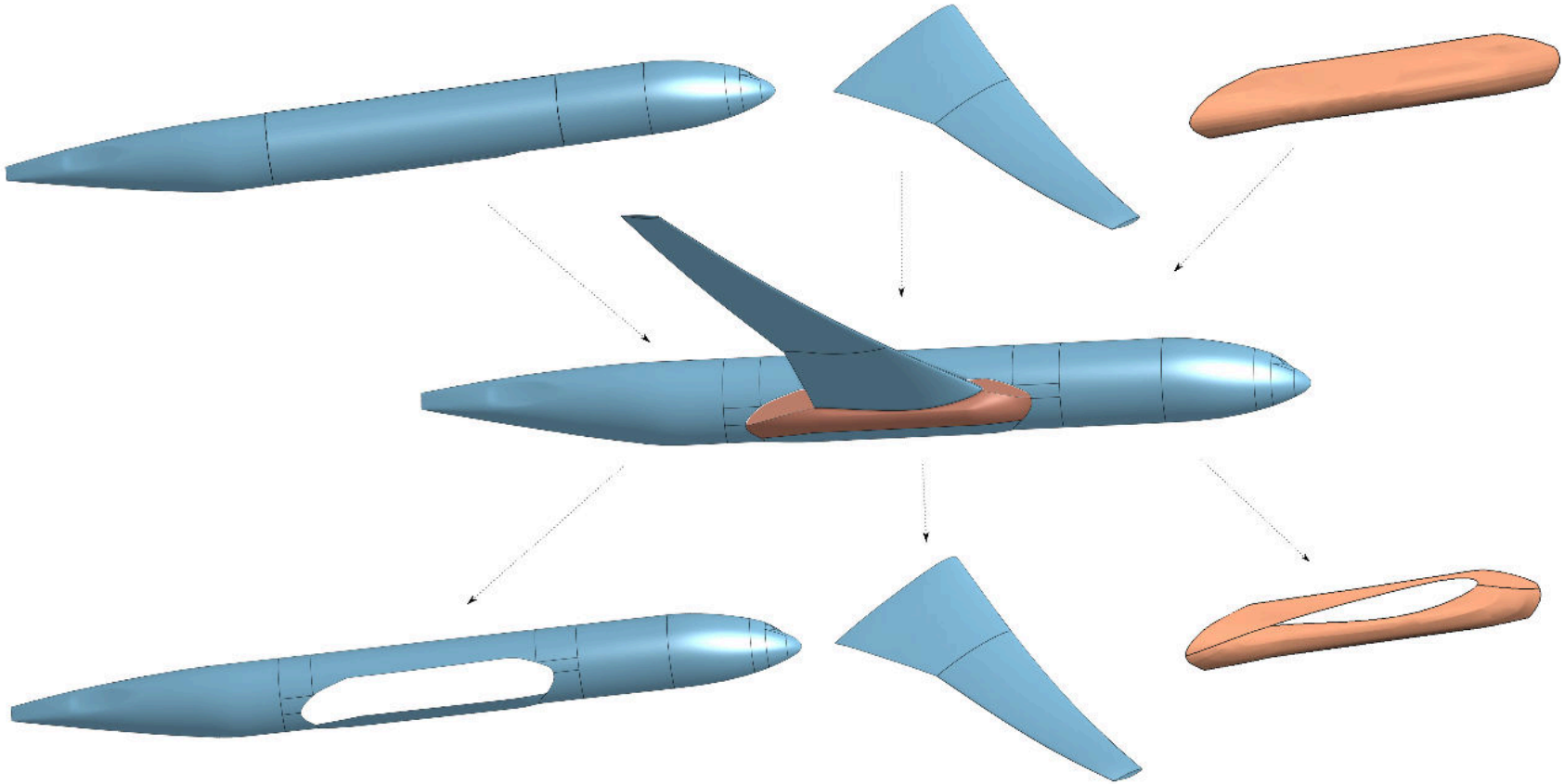
Optimisation History





CAD Boolean Operations and Trimmed Faces

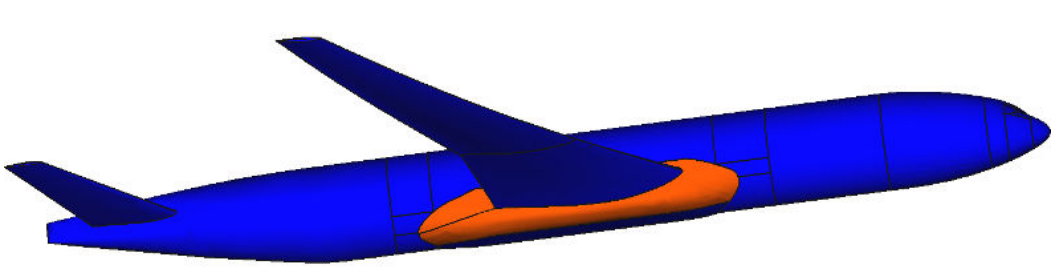
Typical CAD Workflow: CRM

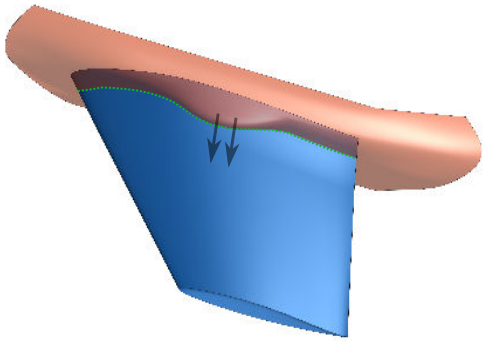


- Independent parts are trimmed after Boolean Operations (Fuse, Common): Surface-Surface intersections
- We want to optimise geometries with trimmed patches
- We want to find places where trims should be performed

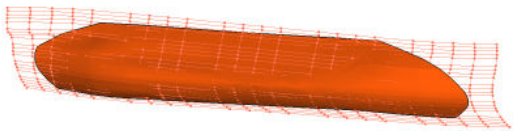
Testcase: Wing-fairing intersection

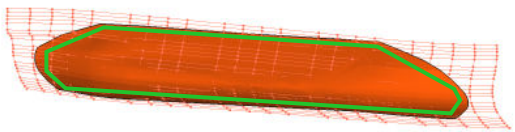
Design area (orange)





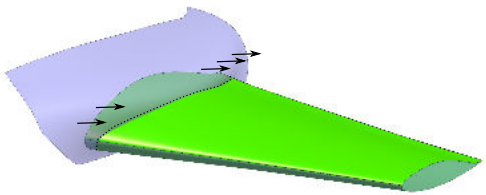
NURBS Parametrisation

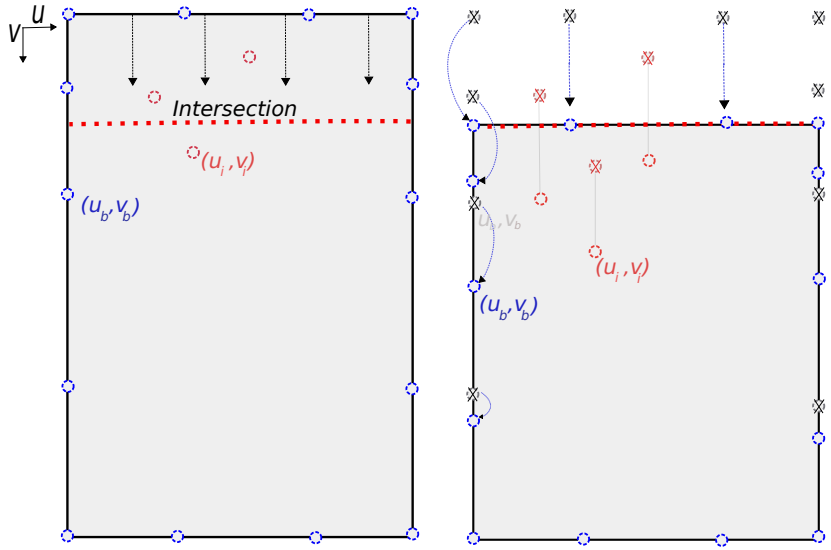




- Fairing design: 484 Control Points
- 169 Control Points allowed to move = 507 design variables
- Continuity with Fuselage

Mesh Morphing





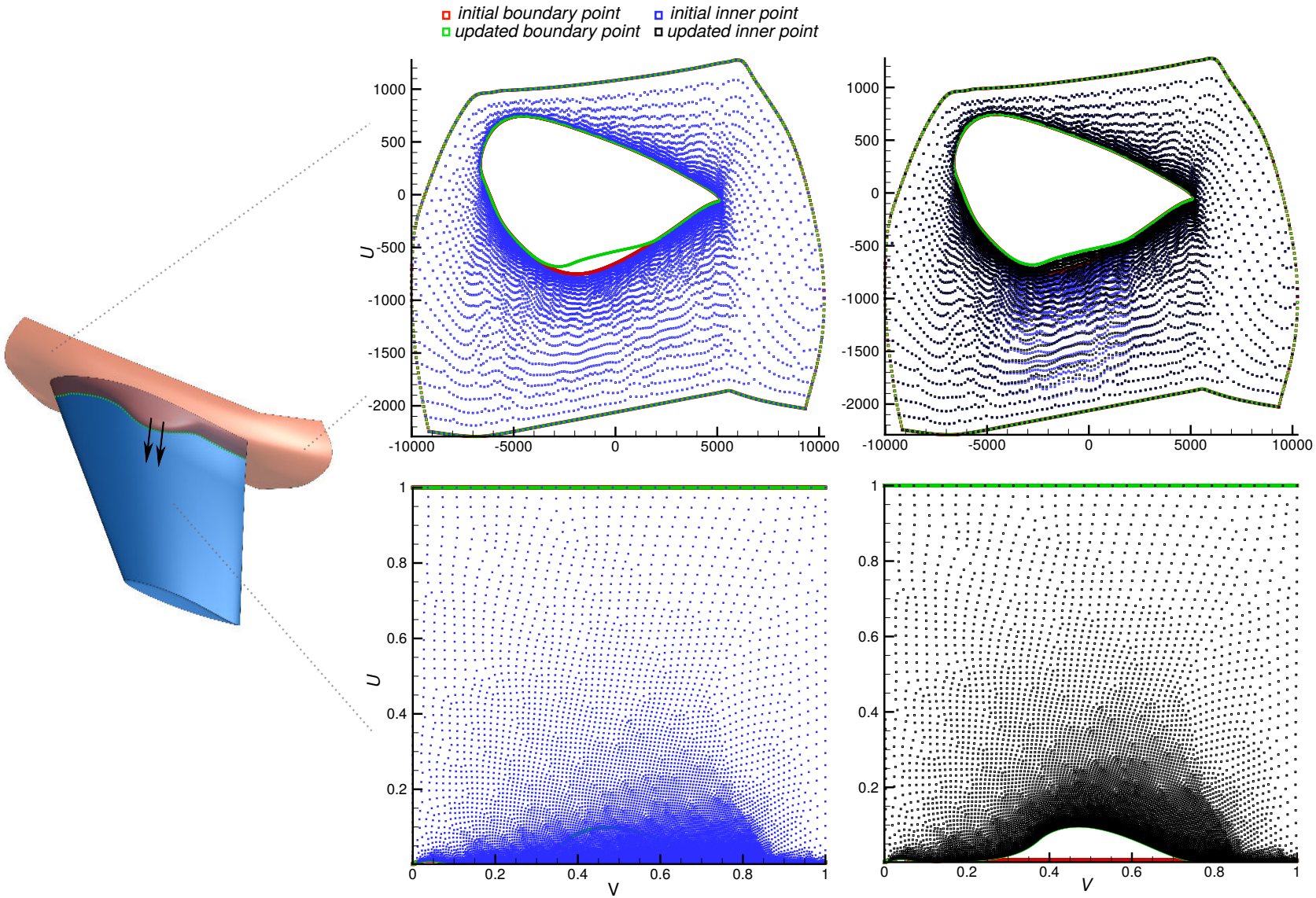
0. Mesh point inversions: Find $(\mathrm{Face}_{CAD}, u, v)$

CAD-Mesh Movement Algorithm:

1. Change Control Points P of the fairing patch
2. Recalculate new intersection curve
3. Sample curve to define displacement $(\delta u_b, \delta v_b)$: 1-D
4. Updated mesh points (u_i, v_i) in parametric space: 2-D IDW
5. Project to Cartesian space $(x, y, z) = S(u, v)$
6. Propagate surface movement into volume: 3-D IDW

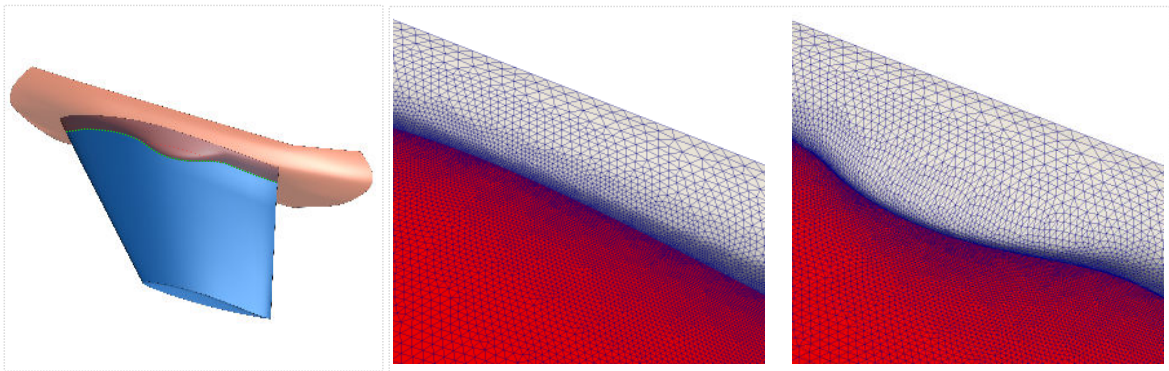
Parametric 2-D Mesh Movement

2-D IDW on Fuselage and the wing

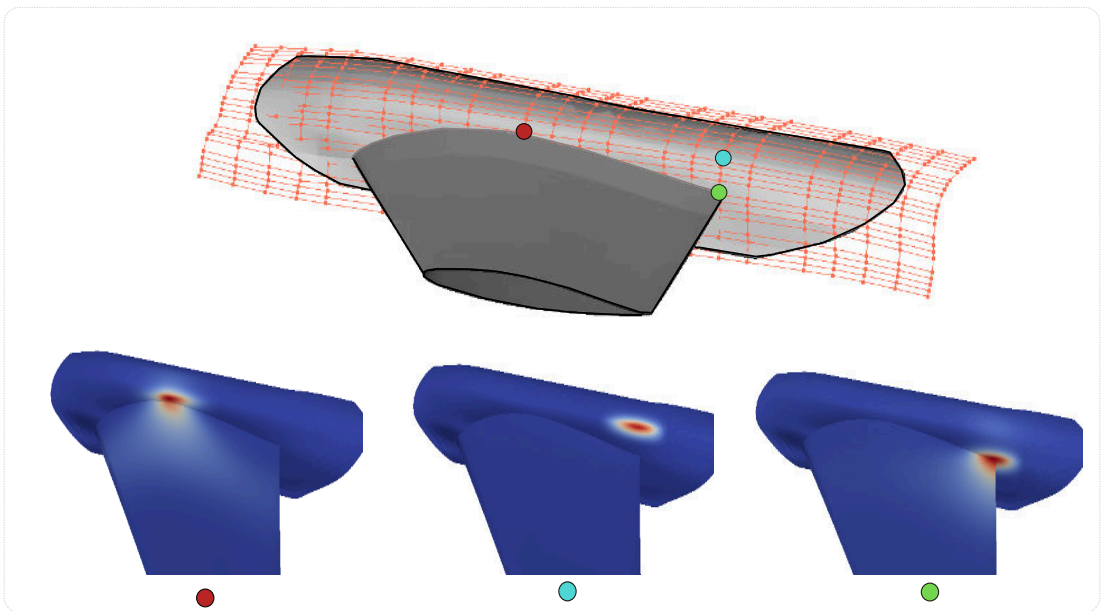


Computing CAD sensitivities for trimmed patches

3-D Mesh Movement

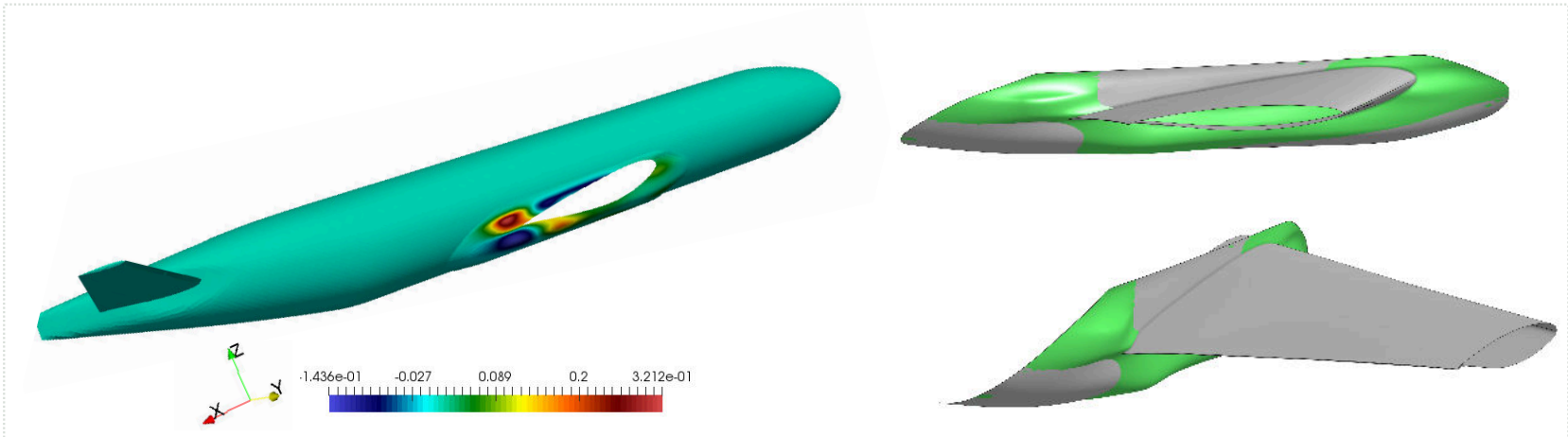


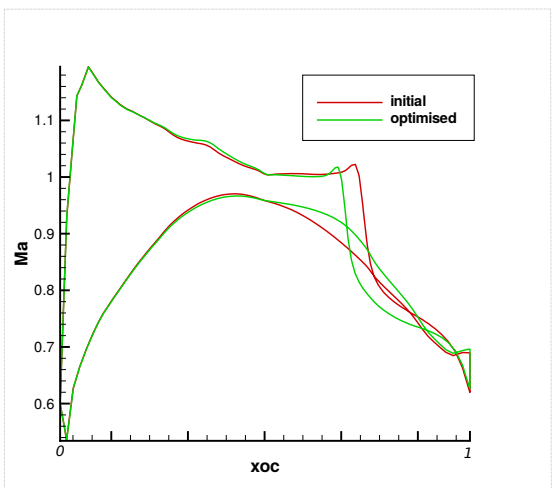
Algorithm Intersections Sensitivity: Move CPs, AD OCCT intersection, mesh ($1D \rightarrow 2D \rightarrow 3D$)

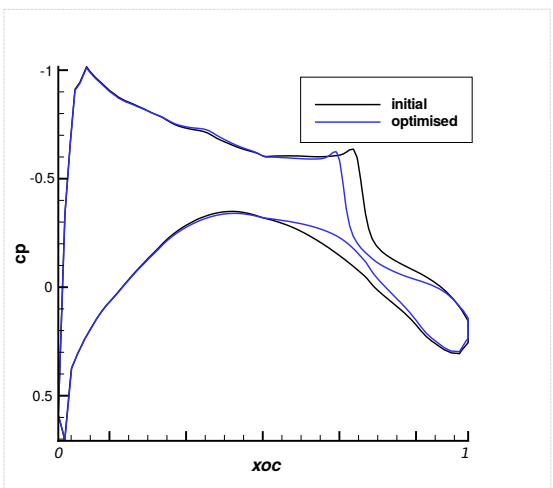


NASA CRM Results

Non-trivial updated intersection







Result: $\frac{C_L}{C_D}$: + 1.8%, C_L : -4%, C_D: -6%.

Outline

The adjoint method

Introduction to Algorithmic Differentiation

STAMPS - Source-Transformation Adjoint Multi-Purpose Solver

CAD-based shape optimisation

Uncertainty Quantification

Summary

Multi-fidelity Monte Carlo (control variate)

$$J_p^{CV} = J_p^{MC} - \beta \left(G_q^{MC} - \mathbb{E}[G] \right)$$

J_p^{CV}: Control variate estimator J_p^{MC}: Monte Carlo estimator

β: Control variate parameter G_q^{MC}: Approximate model estimator

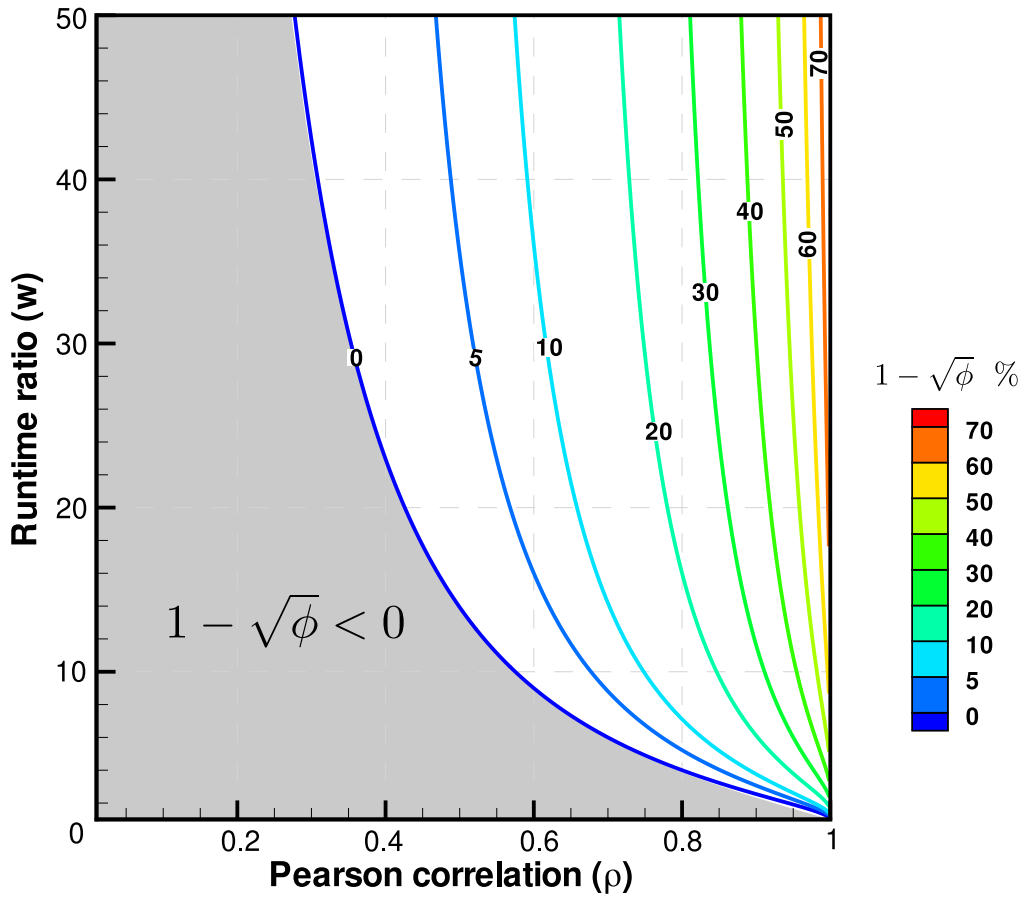
$\mathbb{E}[G]$: Mean value of G

- Run q samples of approximate or low fidelity (LF) model G
- Run p samples of high fidelity (HF) model J, ($q > p$)
- Use β to correct errors in G (G should be cheap to eval)
- Possible to obtain optimal sampling using a cost/accuracy model

Multi-fidelity error estimation

Estimated % reduction in RMSE ($\sqrt{\text{MSE}}$) = $1 - \sqrt{\phi}$ %

- $1 - \sqrt{\phi} < 0$ cost of CV more than MC
- Model correlation $\rho > 0.3$ (at least)
- Reasonably high w necessary
- Even with $\rho \approx 0.98$ and $w \approx 50$ only 70% reduction possible!



Critical requirements for low-fidelity models

To obtain an efficient multi-fidelity Monte Carlo method, we need

- high runtime gain when switching to LF model:
  $w \approx 30 - 50$ (setup cost + evaluation cost)
- good HF-LF model correlation:
  $\rho > 0.5$
- when the number of uncertainties d is large:) preservation of d-independent convergence rate of MC.

Inexpensive Monte Carlo as low-fidelity model

Inexpensive Monte Carlo of Ghate and Giles

$$J(u_\delta, \alpha_\delta) \approx J(u_\delta^*, \alpha_\delta) - v_\delta^{*T} R(u_\delta^*, \alpha_\delta) + \ldots$$
$$\ldots \mathcal{O}\left(\max\left(||u_\delta^* - u_\delta||^2, ||v_\delta^* - v_\delta||^2\right)\right)$$

- Perturbations about a mean state $u \rightarrow u_\delta$ and mean parameter $\alpha \rightarrow \alpha_\delta$
- Adjoint error correction (of Pierce and Giles) to obtain approximation to QoI J

D. Ghate, PhD thesis, 2013, Univ. Oxford

Inexpensive Monte Carlo

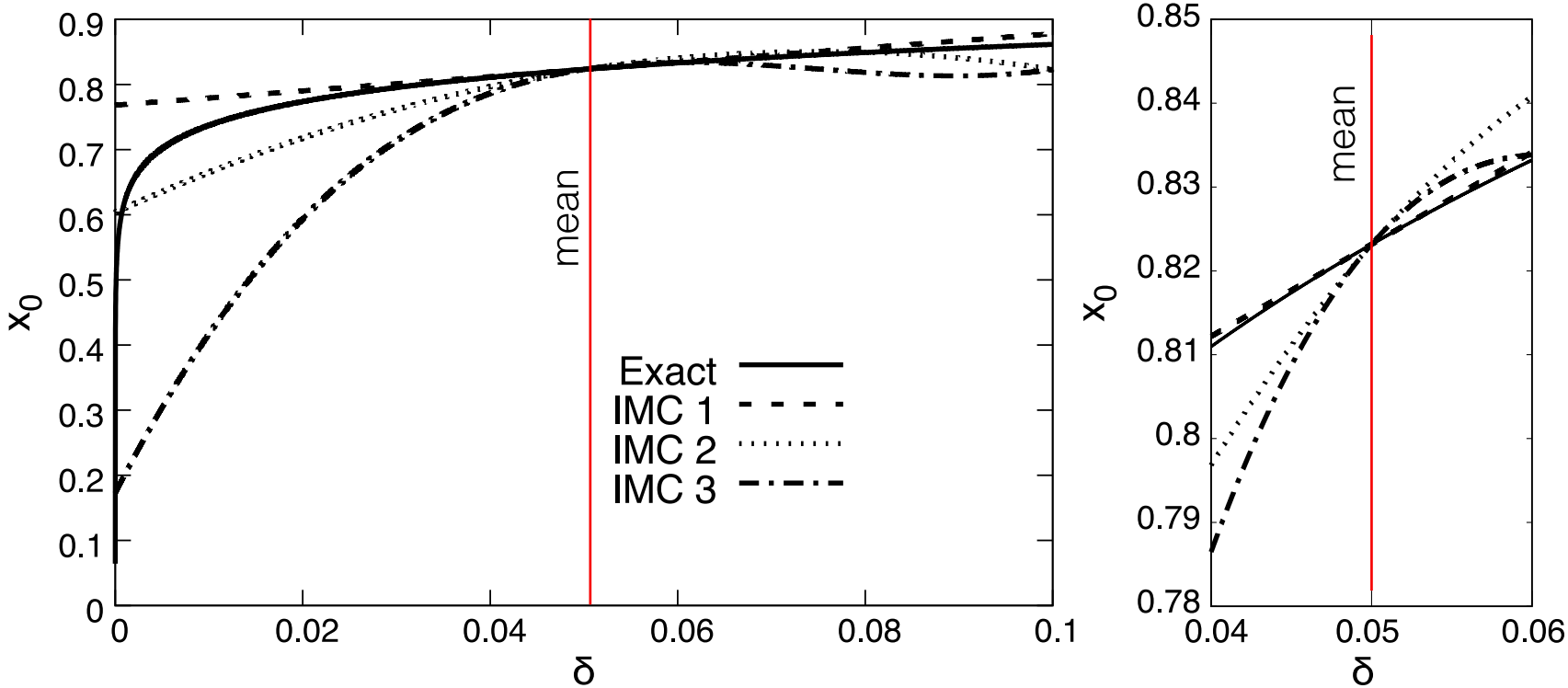
$$J(u_\delta, \alpha_\delta) \approx J(u_\delta^*, \alpha_\delta) - {v_\delta^*}^T R(u_\delta^*, \alpha_\delta)$$

IMC 1: $u_\delta^* = u,\ v_\delta^* = v$
One adjoint solution at mean state

IMC 2: $u_\delta^* = u + \frac{du}{d\alpha}(\alpha_\delta - \alpha),\ v_\delta^* = v$
IMC 1 cost + d tangent-linear solution

IMC 3: $u_\delta^* = u + \frac{du}{d\alpha}(\alpha_\delta - \alpha),\ v_\delta^* = v + \frac{dv}{d\alpha}(\alpha_\delta - \alpha)$
IMC 2 cost + Hessian solution

- Regularity of QoI and model critical to IMC 2/3
- Residual evaluation captures weak non-linearity
- Possible to obtain full *pdf* (unlike Moment method)



Model	$\mathbb{E}[x_0]$ (% *error*)	$\sigma(x_0)$ (% *error*)	$\rho(x_0)$
Exact	0.80728	5.28691×10^{-2}	-
IMC 1	0.82335 $(+2\%)$	3.17072×10^{-2} (-40%)	0.8760
IMC 2	0.78691 (-2.5%)	7.11764×10^{-2} $(+35\%)$	0.9446
IMC 3	0.71554 (-11%)	1.71379×10^{-1} $(+225\%)$	0.9484

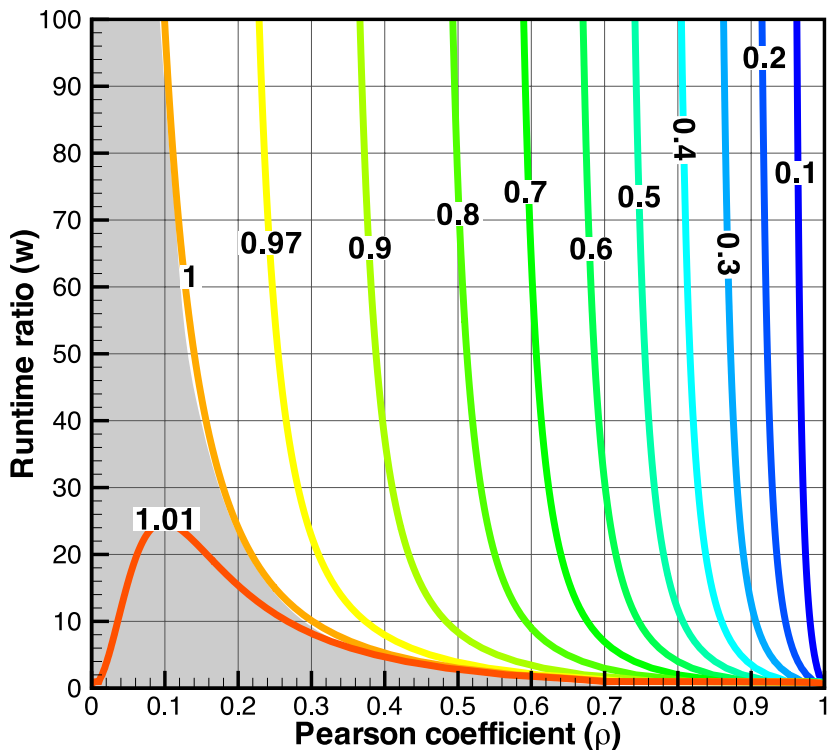
FastUQ: Multi-level Multi-fidelity MC

Combining multi-fidelity with a multi-level MC framework yields further reduction in cost, (extension of work by Geraci)

$$\text{Optimal samples } (l) = \frac{2}{\epsilon^2} \sum_{k=0}^{L} \left[\sqrt{\mathbb{V}ar(Y_k) c_k} \; \Lambda_k \left(r_k^{opt} \right) \right] \sqrt{\frac{\mathbb{V}ar(Y_l)}{c_l}}$$

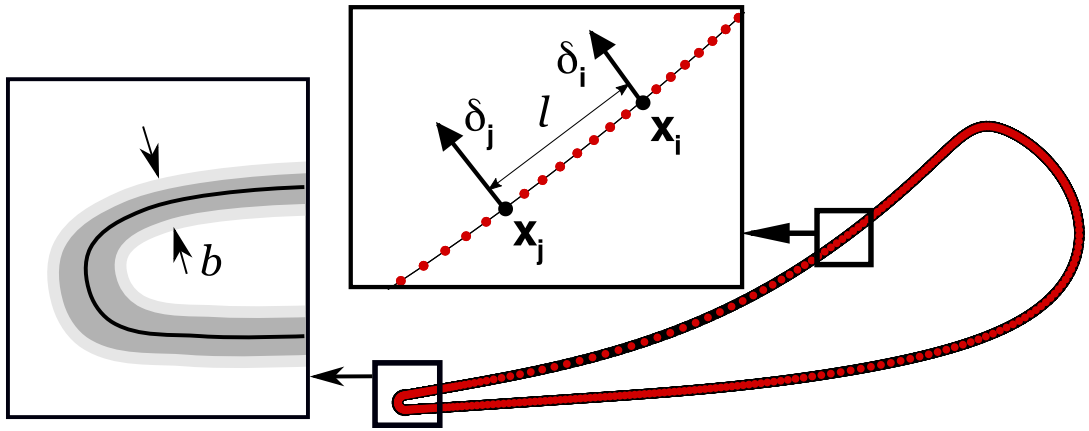
- $\Lambda_k \left(r_k^{opt} \right)$ additional parameter to the MLMC due to multi-fidelity (MF)
- Reduction at every level
- No gain in grey region using MF
- $\phi = (1 + \frac{r^{opt}}{w})\Lambda$

Geraci, CTR Report, 2015



Contours of $\Lambda = \Lambda(r^{opt}(w, \rho))$

UQ of surface variations (LS89 cascade)



- Gaussian process model to represent surface variations (e.g. mfr'ing, normal to surface) of LS89 Turbine cascade

$$\mathbf{x}_\delta = \underbrace{\mathbf{x}}_{\text{Mean surface}} + \underbrace{\delta(\mathbf{x})}_{\text{Zero-mean Gaussian process}} \underbrace{\hat{n}}_{\text{Surface normal}}$$

- Squared exponential correlation function for GP using perturbation height b and length $l \rightarrow \mathbf{c}_{ij} = b^2 \exp\left(-\frac{||x_i - x_j||^2}{2l^2}\right)$

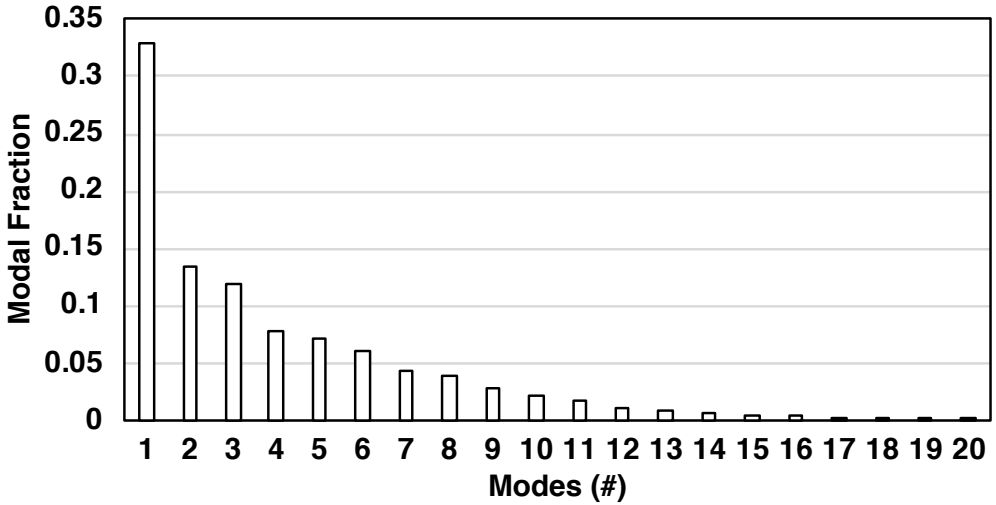


Figure: Modal fraction: LS89 turbine cascade first 20 PCA modes

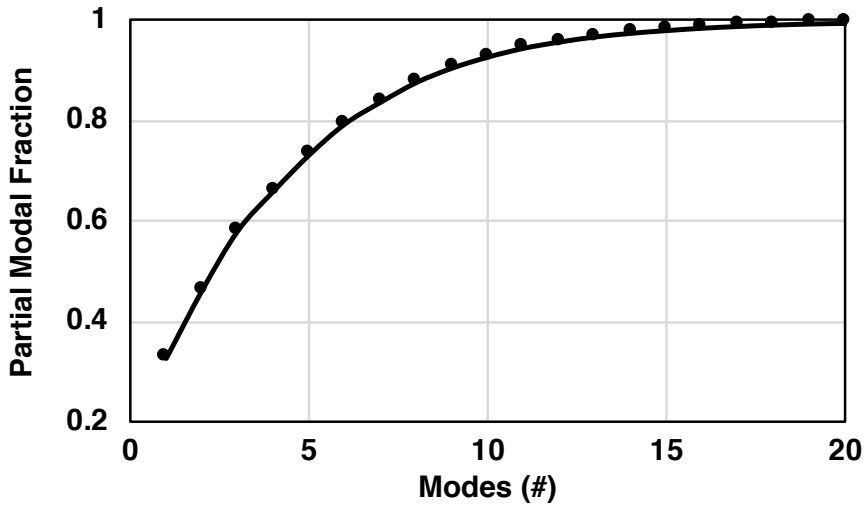


Figure: Partial modal fraction: LS89 turbine cascade first 20 PCA modes

Goal-based truncation of PCA modes

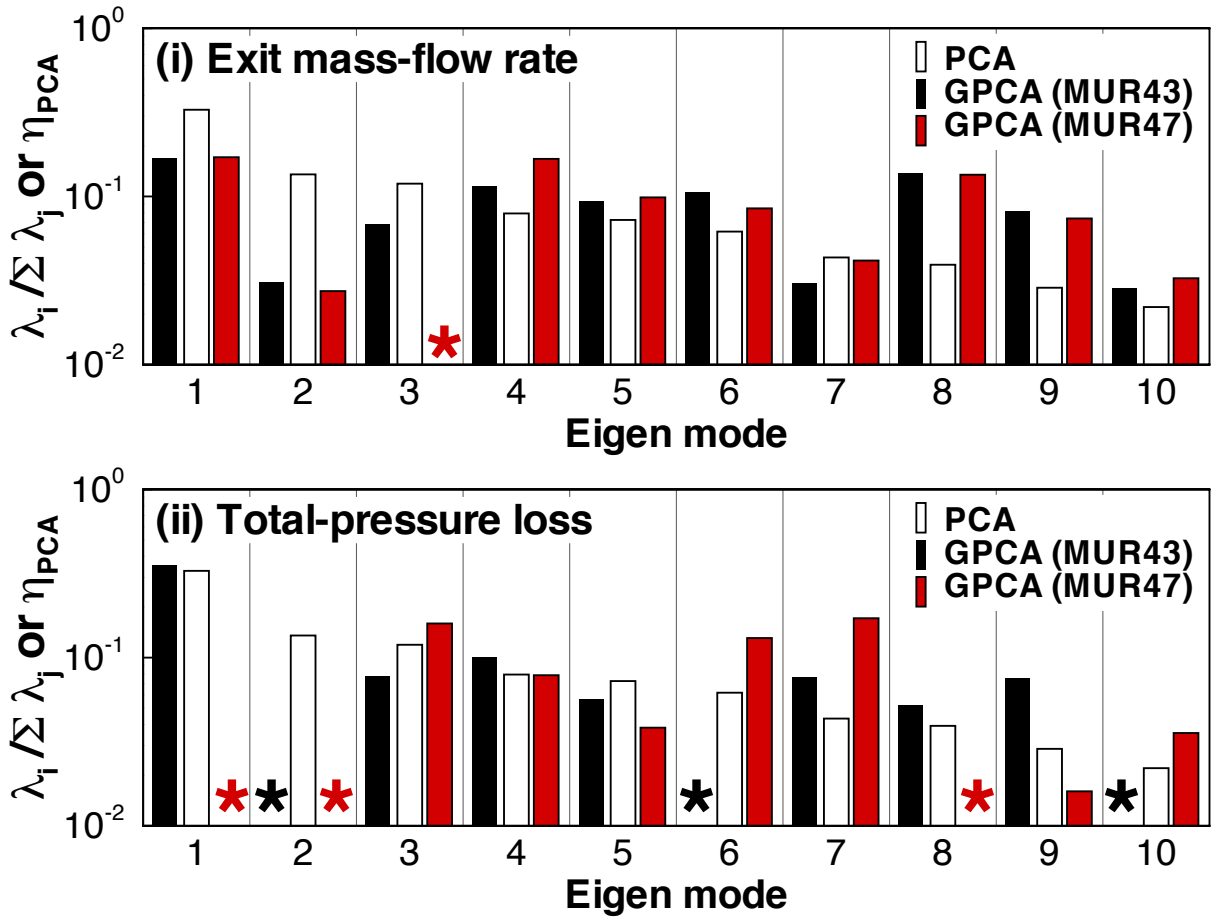
- Goal based PCA magnitude (PCA dot-product with adjoint sensitivity)

$$\eta_{PCA_i} = \lambda_i \frac{dJ}{d\mathbf{z}_i} \approx \lambda_i \frac{dJ}{dx} \cdot \mathbf{z}_i$$

- New modal and partial fraction definition

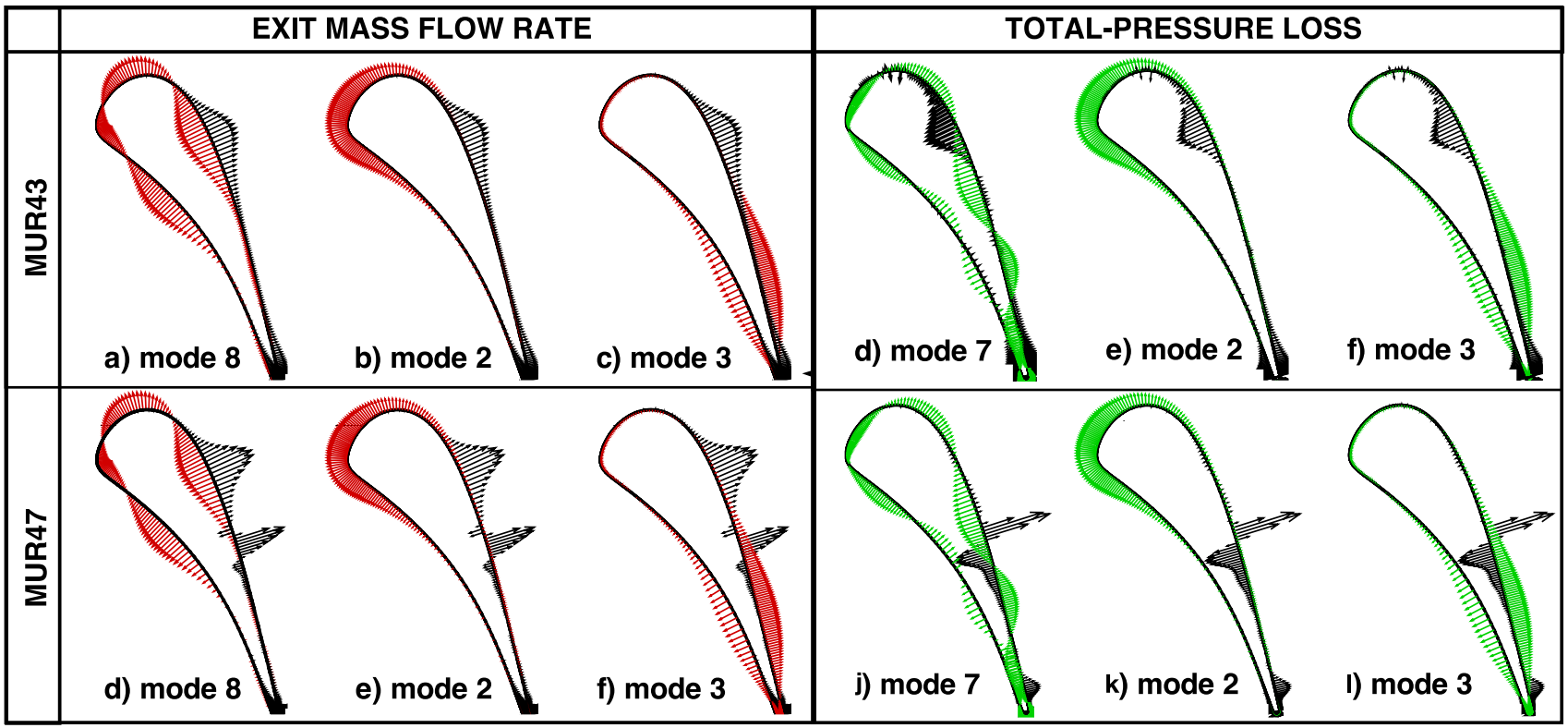
$$\bar{\lambda}_i = \frac{\eta_{PCA_i}}{\sum_{j=1}^{n} \eta_{PCA_j}} \qquad \bar{\Lambda}_i = \frac{\sum_{j=1}^{i} \eta_{PCA_i}}{\sum_{j=1}^{n} \eta_{PCA_j}}$$

Goal-based truncation of PCA modes



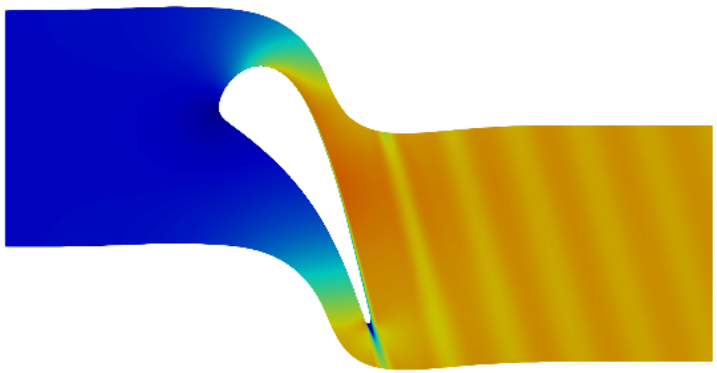
- Goal-based modes drastically different
- Difference for cost-function and flow condition
- MUR43 $M_{exit} = 0.84$ and MUR47 $M_{exit} = 1.02$

Goal-based truncation of PCA modes

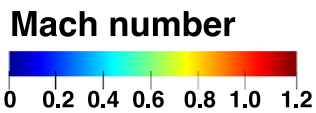


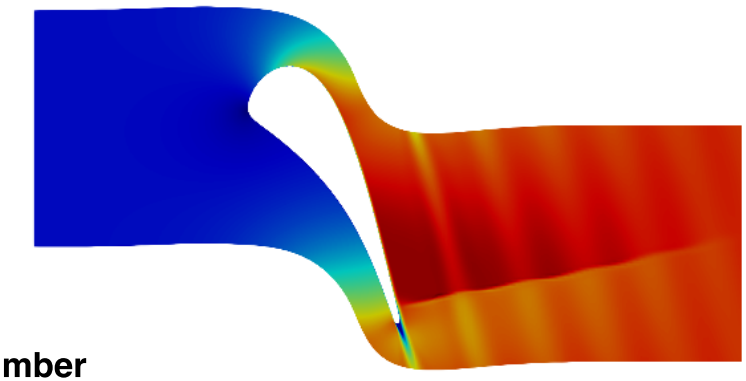
Constructive superposition of modes on sensitivity = high GPCA value

Results: Flow conditions

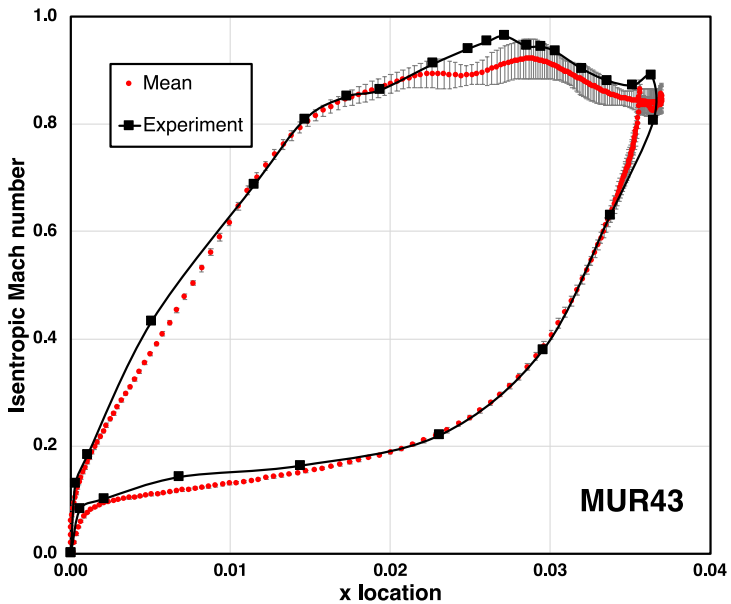


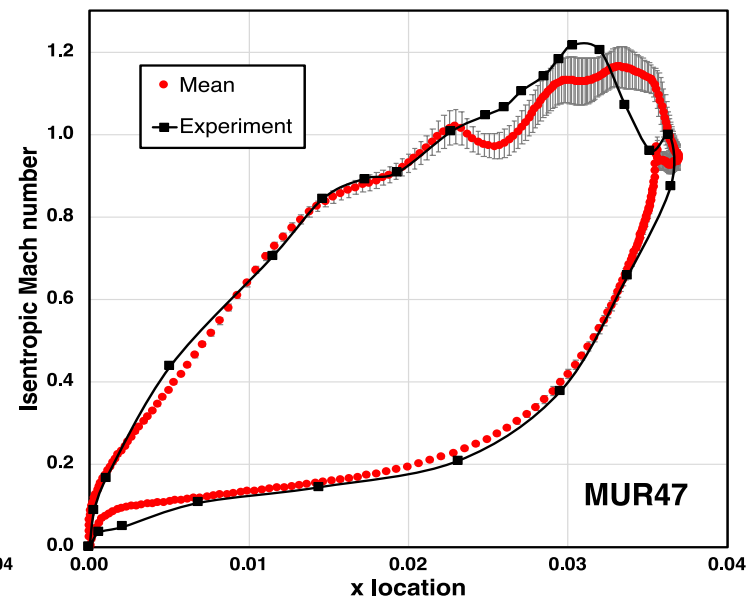
(a) MUR43





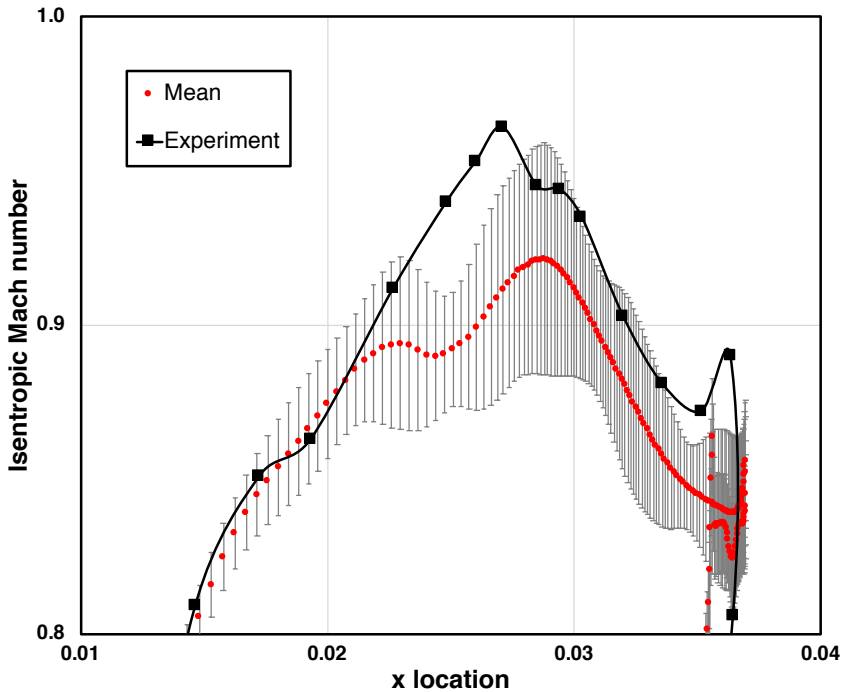
(b) MUR47



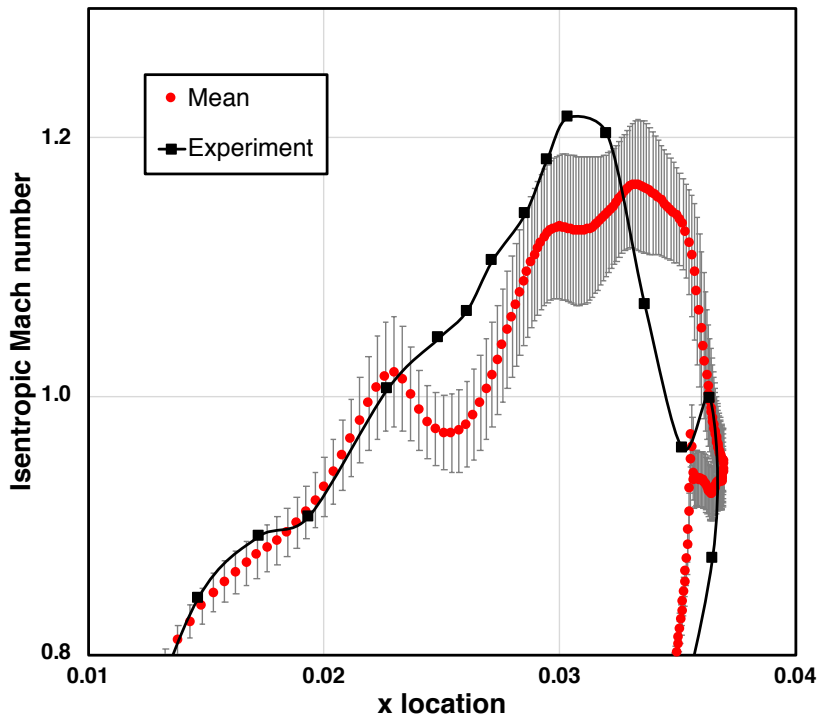


"Error bars are $\pm\sigma$ (std. dev.)"

(a) MUR43



(b) MUR47



Results: How many G-PCA modes?

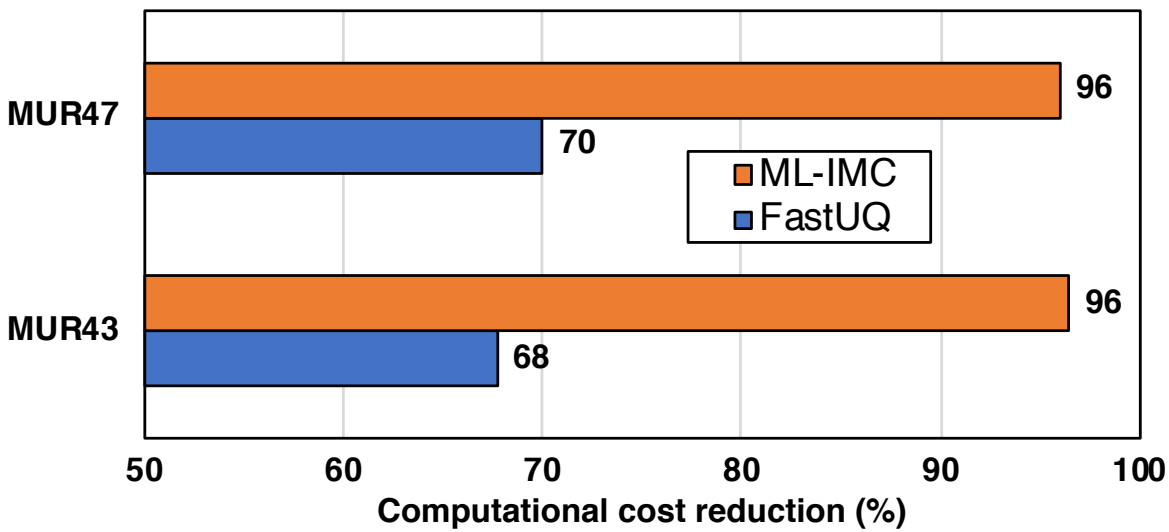
# G-PCA modes	Exit mass flow	Total-pressure loss
7 modes	$\mathbf{4.62}604\ (\pm\mathbf{2}.07639 \times \mathbf{10}^{-2})$	$\mathbf{1.8}5356\ (\pm\mathbf{1}.07648 \times \mathbf{10}^{-1})$
25 modes	$\mathbf{4.62}286\ (\pm\mathbf{2}.64565 \times \mathbf{10}^{-2})$	$\mathbf{1.8}9269\ (\pm\mathbf{1}.48167 \times \mathbf{10}^{-1})$

- 7 modes capture 50% of the G-PCA spectrum (both MUR43/47 + QoIs)
- Need 25 modes to capture 99% of the G-PCA spectrum
- Expect this to grow for 3-D and additional QoI/conditions
- Lange used 60 PCA (truncated from 130) modes for a 3-D turbine blade (efficiency and loss QoIs using Monte Carlo)

Lange et al. “Principal component analysis on 3D scanned compressor blades for probabilistic CFD simulation”, 53^{rd} AIAA conf., 2012

MLMF vs. Multilevel IMC: Computational cost reduction

% reduction in computational cost in comparison to SMLMC (for MSE $= 0.01$)



- MLMF as accurate as plain MLMC
- Cost comparable to ML-IMC

Conclusion

- Proposed a new adjoint-assisted MLMF for UQ
- Can handle large number of input uncertainties
- Gives 70% reduction in cost over MLMC for good correlation ($\rho \approx 0.8 - 0.95$) b/w IMC and HF
- Combining the IMC in a multifidelity control variate increases its range of application and accuracy (at an increased cost)

Type	**Eqv. HF (avg)**	% decrease
SMLMC	182	-
IMC 1	9	96%
FastUQ	56	69%

Outline

The adjoint method

Introduction to Algorithmic Differentiation

STAMPS - Source-Transformation Adjoint Multi-Purpose Solver

CAD-based shape optimisation

Uncertainty Quantification

Summary

Conclusions on AD

- Automatic differentiation is an effective and maintainable way to compute derivatives, provided source code is available.
- Forward mode is *straight*-forward, reverse mode is more complex and can be memory intensive.
- Initial investment in code transformation can be substantial, in particular if efficient code is needed. But from then on maintenance can be automated.
- The derivative computation is exact and typically robust.
- Reverse mode allows to tackle a wide variety of optimisation problems with very, very many of degrees of freedom.
- The challenge then may becomes convergence of the optimiser requiring good preconditioning of the control problem with multi-level approaches, regularisation.

Conclusions on AD

- Automatic differentiation is an effective and maintainable way to compute derivatives, provided source code is available.
- Forward mode is *straight*-forward, reverse mode is more complex and can be memory intensive.
- Initial investment in code transformation can be substantial, in particular if efficient code is needed. But from then on maintenance can be automated.
- The derivative computation is exact and typically robust.
- Reverse mode allows to tackle a wide variety of optimisation problems with very, very many of degrees of freedom.
- The challenge then may becomes convergence of the optimiser requiring good preconditioning of the control problem with multi-level approaches, regularisation.

Conclusions on adjoint workflows

- Source code production for adjoint STAMPS solver is fully automated and highly efficient: runtime 0.7 against flow, memory 1.12 against flow
- Successful differentiation of open-source CAD kernel Open-CASCADE in forward and reverse mode.
- Demonstration of CAD-in-the-loop using explicit shape parametrisations using hierarchic feature trees.
- Automatic derivation of rich design spaces from the BRep, NSPCC framework to impose geometric constraints.
- Extension of NSPCC to handle intersecting, trimmed patches.
- Adaptive and re-parametrisation approaches to produce optimal hierarchic design spaces.
- Application to pylon-nacelle cases in progress.

Conclusions on adjoint workflows

- Source code production for adjoint STAMPS solver is fully automated and highly efficient: runtime 0.7 against flow, memory 1.12 against flow
- Successful differentiation of open-source CAD kernel Open-CASCADE in forward and reverse mode.
- Demonstration of CAD-in-the-loop using explicit shape parametrisations using hierarchic feature trees.
- Automatic derivation of rich design spaces from the BRep, NSPCC framework to impose geometric constraints.
- Extension of NSPCC to handle intersecting, trimmed patches.
- Adaptive and re-parametrisation approaches to produce optimal hierarchic design spaces.
- Application to pylon-nacelle cases in progress.

Conclusions on adjoint workflows

- Source code production for adjoint STAMPS solver is fully automated and highly efficient: runtime 0.7 against flow, memory 1.12 against flow
- Successful differentiation of open-source CAD kernel Open-CASCADE in forward and reverse mode.
- Demonstration of CAD-in-the-loop using explicit shape parametrisations using hierarchic feature trees.
- Automatic derivation of rich design spaces from the BRep, NSPCC framework to impose geometric constraints.
- Extension of NSPCC to handle intersecting, trimmed patches.
- Adaptive and re-parametrisation approaches to produce optimal hierarchic design spaces.
- Application to pylon-nacelle cases in progress.

Conclusions on adjoint workflows

- Source code production for adjoint STAMPS solver is fully automated and highly efficient: runtime 0.7 against flow, memory 1.12 against flow
- Successful differentiation of open-source CAD kernel Open-CASCADE in forward and reverse mode.
- Demonstration of CAD-in-the-loop using explicit shape parametrisations using hierarchic feature trees.
- Automatic derivation of rich design spaces from the BRep, NSPCC framework to impose geometric constraints.
- Extension of NSPCC to handle intersecting, trimmed patches.
- Adaptive and re-parametrisation approaches to produce optimal hierarchic design spaces.
- Application to pylon-nacelle cases in progress.

Conclusions on adjoint workflows

- Source code production for adjoint STAMPS solver is fully automated and highly efficient: runtime 0.7 against flow, memory 1.12 against flow
- Successful differentiation of open-source CAD kernel Open-CASCADE in forward and reverse mode.
- Demonstration of CAD-in-the-loop using explicit shape parametrisations using hierarchic feature trees.
- Automatic derivation of rich design spaces from the BRep, NSPCC framework to impose geometric constraints.
- Extension of NSPCC to handle intersecting, trimmed patches.
- Adaptive and re-parametrisation approaches to produce optimal hierarchic design spaces.
- Application to pylon-nacelle cases in progress.

Conclusions on adjoint workflows

- Source code production for adjoint STAMPS solver is fully automated and highly efficient: runtime 0.7 against flow, memory 1.12 against flow
- Successful differentiation of open-source CAD kernel Open-CASCADE in forward and reverse mode.
- Demonstration of CAD-in-the-loop using explicit shape parametrisations using hierarchic feature trees.
- Automatic derivation of rich design spaces from the BRep, NSPCC framework to impose geometric constraints.
- Extension of NSPCC to handle intersecting, trimmed patches.
- Adaptive and re-parametrisation approaches to produce optimal hierarchic design spaces.
- Application to pylon-nacelle cases in progress.

Conclusions on adjoint workflows

- Source code production for adjoint STAMPS solver is fully automated and highly efficient: runtime 0.7 against flow, memory 1.12 against flow
- Successful differentiation of open-source CAD kernel Open-CASCADE in forward and reverse mode.
- Demonstration of CAD-in-the-loop using explicit shape parametrisations using hierarchic feature trees.
- Automatic derivation of rich design spaces from the BRep, NSPCC framework to impose geometric constraints.
- Extension of NSPCC to handle intersecting, trimmed patches.
- Adaptive and re-parametrisation approaches to produce optimal hierarchic design spaces.
- Application to pylon-nacelle cases in progress.

Outlook

- CAD-based shape optimisation is ready for industrial application with full-adjoint design chains
- “Coarse-grain” unsteady adjoint calculations are feasible, accuracy of gradients for large-scale shape modes appears robust.
- Adjoints will be an important tool to accelerate uncertainty quantification, essential for robust design.
- AD will be essential to develop adjoints for complex multi-level and multi-physics systems.

Outlook

- CAD-based shape optimisation is ready for industrial application with full-adjoint design chains
- “Coarse-grain” unsteady adjoint calculations are feasible, accuracy of gradients for large-scale shape modes appears robust.
- Adjoints will be an important tool to accelerate uncertainty quantification, essential for robust design.
- AD will be essential to develop adjoints for complex multi-level and multi-physics systems.

Outlook

- CAD-based shape optimisation is ready for industrial application with full-adjoint design chains
- “Coarse-grain” unsteady adjoint calculations are feasible, accuracy of gradients for large-scale shape modes appears robust.
- Adjoints will be an important tool to accelerate uncertainty quantification, essential for robust design.
- AD will be essential to develop adjoints for complex multi-level and multi-physics systems.

Outlook

- CAD-based shape optimisation is ready for industrial application with full-adjoint design chains
- “Coarse-grain” unsteady adjoint calculations are feasible, accuracy of gradients for large-scale shape modes appears robust.
- Adjoints will be an important tool to accelerate uncertainty quantification, essential for robust design.
- AD will be essential to develop adjoints for complex multi-level and multi-physics systems.

Acknowledgements

This work has been conducted within the **About Flow** and **IODA** projects at Queen Mary University of London

`http://{aboutflow,ioda}.sems.qmul.ac.uk`

We have received funding from the European Union's 7th FP and H2020 programs under Grant Agreement Nos. 317006 and 642959.