Lessons from differentiating barely stable PDE solvers

J.-D. Müller, Siamak Akbarzadeh
Queen Mary University of London

Sami Bayyuk
ESI-Group, Huntsville, AL

Jan Hückelheim
Imperial College, London

22nd AD workshop
Imperial College, London, 1-2 July 2019
How to compute adjoint solutions for challenging pdes

- AD has established itself as THE method to produce derivative code: automatically, accurately.
- Widely used in PDE-based modelling, adopting the typical method of lines/semi-discrete approach: use AD for derivative code of the spatial discretisation.
- Very successful for discretisations with linear operators for stabilisation such as FE, algorithmic diff can be used for complete libraries, see e.g. Dolfin library for Fenics.
- Also very successful for non-linear stabilisations such as use in finite volume methods for CFD, typically application of an AD tool to the specific source code.
- AD can be verified using the dot product test, and is accurate to machine precision (unlike continuous adjoint formulations in general).
How to compute adjoint solutions for challenging PDEs

- AD has established itself as THE method to produce derivative code: automatically, accurately.
- Widely used in PDE-based modelling, adopting the typical method of lines/semi-discrete approach: use AD for derivative code of the spatial discretisation.
- Very successful for discretisations with linear operators for stabilisation such as FE, algorithmic diff can be used for complete libraries, see e.g. Dolfin library for Fenics.
- Also very successful for non-linear stabilisations such as use in finite volume methods for CFD, typically application of an AD tool to the specific source code.
- AD can be verified using the dot product test, and is accurate to machine precision (unlike continuous adjoint formulations in general).
How to compute adjoint solutions for challenging pdes

- AD has established itself as THE method to produce derivative code: automatically, accurately.
- Widely used in PDE-based modelling, adopting the typical method of lines/semi-discrete approach: use AD for derivative code of the spatial discretisation.
- Very successful for discretisations with linear operators for stabilisation such as FE, algorithmic diff can be used for complete libraries, see e.g. Dolfin library for Fenics.
- Also very successful for non-linear stabilisations such as use in finite volume methods for CFD, typically application of an AD tool to the specific source code.
- AD can be verified using the dot product test, and is accurate to machine precision (unlike continuous adjoint formulations in general).
Iterators

• Application cases will often involve implicit solves of inner, linear systems.

• Taping and adjoint reversal of the primal from initialisation to convergence is typically not feasible and not necessarily stable AD applied to Krylov linear solvers can behave poorly.¹

• For steady-state: typically linearisation around the converged steady-state, replace differentiating the linear solver with calling linear solver an adjoint system.

• But: this requires linear stability of the spatial discretisation as non-linear stabilisation is determined by the primal (and frozen).

• Contractivity: all eigenvalues of the update scheme are $\leq 1$.

¹Moré, Wild, "Do you trust derivatives or differences?", J. Comp. Phys. 2014
Iterators

- Application cases will often involve implicit solves of inner, linear systems.
- Taping and adjoint reversal of the primal from initialisation to convergence is typically not feasible and not necessarily stable AD applied to Krylov linear solvers can behave poorly.\(^1\)
- For steady-state: typically linearisation around the converged steady-state, replace differentiating the linear solver with calling linear solver an adjoint system.
  - But: this requires linear stability of the spatial discretisation as non-linear stabilisation is determined by the primal (and frozen).
  - Contractivity: all eigenvalues of the update scheme are \(<= 1\).

\(^1\)Moré, Wild, ”Do you trust derivatives or differences?”, J. Comp. Phys. 2014
Iterators

• Application cases will often involve implicit solves of inner, linear systems.
• Taping and adjoint reversal of the primal from initialisation to convergence is typically not feasible and not necessarily stable AD applied to Krylov linear solvers can behave poorly.\(^1\)
• For steady-state: typically linearisation around the converged steady-state, replace differentiating the linear solver with calling linear solver an adjoint system.
• **But:** this requires linear stability of the spatial discretisation as non-linear stabilisation is determined by the primal (and frozen).
• Contractivity: all eigenvalues of the update scheme are \(\leq 1\).

\(^1\) Moré, Wild, ”Do you trust derivatives or differences?”, J. Comp. Phys. 2014
Convergence of the flow solver to limit cycles

CFD discretisations of large problems are typically not contractive:

Convergence of an industrial CFD code for turbomachinery case in off-design condition: Block-Jacobi multi-grid acceleration, (left) to limit cycles, divergence of the discrete adjoint solver (right).
Obtaining a stable adjoint

**Direct solver**  Sledgehammer method, only feasible for smaller-size objects

**Krylov solvers**  Match each unstable mode with a pair of Krylov vectors [Giles, Campobasso]. Expensive.

**Recursive Projection**  Filter out unstable modes, treat them with a direct solver or Krylov [Giles, Campobasso]. Expensive.

**Improve stability of primal**  Improve primal stability with strong preconditioners and implicit solvers [Xu, Müller], can work well for fully coupled discretisations. Convergence not guaranteed.
Stable adjoints using contractive primals

QMUL has developed the JT-KIRK iterative scheme combining a range of stabilisation techniques: GMRES with multi-stage Runge-Kutta as a smoother within a multigrid method

- runtime benefits for the primal,
- can often achieve full convergence for flow and hence adjoint,
- objective functions converge much more rapidly.

---

\(^2\)S. Xu et al, “Stabilisation of discrete steady adjoint solvers”, JCP 2015
Stable adjoints: essential for industrial optimisation

With a contractive primal, we can guarantee that its discrete adjoint converges.

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

We now know how to achieve this for the fully coupled compressible flow discretisations near design conditions. What about segregated algorithms, e.g. incompressible flow?
Stable adjoints: essential for industrial optimisation

With a contractive primal, we can guarantee that its discrete adjoint converges

![Residual vs Iteration](chart.png)

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

We now know how to achieve this for the fully coupled compressible flow discretisations near design conditions.

What about segregated algorithms, e.g. incompressible flow?
Stable adjoints: essential for industrial optimisation

With a contractive primal, we can guarantee that its discrete adjoint converges.

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

We now know how to achieve this for the fully coupled compressible flow discretisations near design conditions. What about segregated algorithms, e.g. incompressible flow?
SIMPLE Adjoint?

- The most widely used scheme for incompressible flow is the segregated SIMPLE scheme, which typically does not converge.
- In the absence of convergence, no guarantees for adjoint convergence.
- Currently no prospect to derive a linearly stable scheme for the saddle point problem of incompressible flow.
- What works?
Outline

Introduction

The SIMPLE scheme for incompressible flow

Differentiation of a commercial CFD code

Stability of SIMPLE adjoints

Conclusions
Outline

Introduction

The SIMPLE scheme for incompressible flow

Differentiation of a commercial CFD code

Stability of SIMPLE adjoints

Conclusions
Outline

Introduction

The SIMPLE scheme for incompressible flow

Differentiation of a commercial CFD code

Stability of SIMPLE adjoints

Conclusions
Introduction

The SIMPLE scheme for incompressible flow

Differentiation of a commercial CFD code

Stability of SIMPLE adjoints

Conclusions
Outline

Introduction

The SIMPLE scheme for incompressible flow

Differentiation of a commercial CFD code

Stability of SIMPLE adjoints

Conclusions
Introduction

The SIMPLE scheme for incompressible flow

Differentiation of a commercial CFD code

Stability of SIMPLE adjoints

Conclusions
Segregated SIMPLE scheme for incompressible flow

1. Solve the discrete momentum equations with the latest pressure field gradient to get an update to the velocity field.

2. Interpolate the velocity field onto cell faces using Momentum Interpolation. The resulting mass fluxes will generally not satisfy the continuity equation.

3. Solve the discrete pressure Poisson equation to get an update to the pressure field that satisfies the continuity equation.

4. Correct the velocity field with the pressure correction from Step 3. The resulting velocity field will satisfy the discrete continuity equation exactly, but will no longer satisfy the discrete momentum equations.

5. Solve in segregated fashion any turbulence model equations and any other transport equations, then update all properties.

6. Reapply the boundary conditions and repeat steps 1-5 until the desired convergence criteria are attained.
Segregated SIMPLE scheme for incompressible flow

1. Solve the discrete momentum equations with the latest pressure field gradient to get an update to the velocity field.
2. Interpolate the velocity field onto cell faces using Momentum Interpolation. The resulting mass fluxes will generally not satisfy the continuity equation.
3. Solve the discrete pressure Poisson equation to get an update to the pressure field that satisfies the continuity equation.
4. Correct the velocity field with the pressure correction from Step 3. The resulting velocity field will satisfy the discrete continuity equation exactly, but will no longer satisfy the discrete momentum equations.
5. Solve in segregated fashion any turbulence model equations and any other transport equations, then update all properties.
6. Reapply the boundary conditions and repeat steps 1-5 until the desired convergence criteria are attained.
Segregated SIMPLE scheme for incompressible flow

1. Solve the discrete momentum equations with the latest pressure field gradient to get an update to the velocity field.
2. Interpolate the velocity field onto cell faces using Momentum Interpolation. The resulting mass fluxes will generally not satisfy the continuity equation.
3. Solve the discrete pressure Poisson equation to get an update to the pressure field that satisfies the continuity equation.
4. Correct the velocity field with the pressure correction from Step 3. The resulting velocity field will satisfy the discrete continuity equation exactly, but will no longer satisfy the discrete momentum equations.
5. Solve in segregated fashion any turbulence model equations and any other transport equations, then update all properties.
6. Reapply the boundary conditions and repeat steps 1-5 until the desired convergence criteria are attained.
Segregated SIMPLE scheme for incompressible flow

1. Solve the discrete momentum equations with the latest pressure field gradient to get an update to the velocity field.
2. Interpolate the velocity field onto cell faces using Momentum Interpolation. The resulting mass fluxes will generally not satisfy the continuity equation.
3. Solve the discrete pressure Poisson equation to get an update to the pressure field that satisfies the continuity equation.
4. Correct the velocity field with the pressure correction from Step 3. The resulting velocity field will satisfy the discrete continuity equation exactly, but will no longer satisfy the discrete momentum equations.
5. Solve in segregated fashion any turbulence model equations and any other transport equations, then update all properties.
6. Reapply the boundary conditions and repeat steps 1-5 until the desired convergence criteria are attained.
Segregated SIMPLE scheme for incompressible flow

1. Solve the discrete momentum equations with the latest pressure field gradient to get an update to the velocity field.

2. Interpolate the velocity field onto cell faces using Momentum Interpolation. The resulting mass fluxes will generally not satisfy the continuity equation.

3. Solve the discrete pressure Poisson equation to get an update to the pressure field that satisfies the continuity equation.

4. Correct the velocity field with the pressure correction from Step 3. The resulting velocity field will satisfy the discrete continuity equation exactly, but will no longer satisfy the discrete momentum equations.

5. Solve in segregated fashion any turbulence model equations and any other transport equations, then update all properties.

6. Reapply the boundary conditions and repeat steps 1-5 until the desired convergence criteria are attained.
Segregated SIMPLE scheme for incompressible flow

1. Solve the discrete momentum equations with the latest pressure field gradient to get an update to the velocity field.
2. Interpolate the velocity field onto cell faces using Momentum Interpolation. The resulting mass fluxes will generally not satisfy the continuity equation.
3. Solve the discrete pressure Poisson equation to get an update to the pressure field that satisfies the continuity equation.
4. Correct the velocity field with the pressure correction from Step 3. The resulting velocity field will satisfy the discrete continuity equation exactly, but will no longer satisfy the discrete momentum equations.
5. Solve in segregated fashion any turbulence model equations and any other transport equations, then update all properties.
6. Reapply the boundary conditions and repeat steps 1-5 until the desired convergence criteria are attained.
Segregated SIMPLE scheme for incompressible flow

- Implementation of the segregated SIMPLE scheme is facilitated by a simplified matrix structure for each segregated equation.
- Convergence is hampered by neglecting the coupling between neighbouring velocities in the velocity correction equation.
- Convergence depth and rate are strongly dependent on the types and settings of linear solvers, the inertial relaxation values for the corrections, and the closeness of the coupling between the segregated equations.
- Convergence rate and depth both deteriorate with increasing non-orthogonality corrections (from poor cell quality) and increasing cell count.
- Convergence often stalls after 3-5 orders of drop in the residual for typical large-scale industrial problems with $10^7 - 10^9$ cells, even with a small ratio of poor-quality cells.
Segregated SIMPLE scheme for incompressible flow

- Implementation of the segregated SIMPLE scheme is facilitated by a simplified matrix structure for each segregated equation.
- Convergence is hampered by neglecting the coupling between neighbouring velocities in the velocity correction equation.
- Convergence depth and rate are strongly dependent on the types and settings of linear solvers, the inertial relaxation values for the corrections, and the closeness of the coupling between the segregated equations.
- Convergence rate and depth both deteriorate with increasing non-orthogonality corrections (from poor cell quality) and increasing cell count.
- Convergence often stalls after 3-5 orders of drop in the residual for typical large-scale industrial problems with $10^7 - 10^9$ cells, even with a small ratio of poor-quality cells.
Segregated SIMPLE scheme for incompressible flow

• Implementation of the segregated SIMPLE scheme is facilitated by a simplified matrix structure for each segregated equation.

• Convergence is hampered by neglecting the coupling between neighbouring velocities in the velocity correction equation.

• Convergence depth and rate are strongly dependent on the types and settings of linear solvers, the inertial relaxation values for the corrections, and the closeness of the coupling between the segregated equations.

• Convergence rate and depth both deteriorate with increasing non-orthogonality corrections (from poor cell quality) and increasing cell count.

• Convergence often stalls after 3-5 orders of drop in the residual for typical large-scale industrial problems with $10^7 - 10^9$ cells, even with a small ratio of poor-quality cells.
Segregated SIMPLE scheme for incompressible flow

- Implementation of the segregated SIMPLE scheme is facilitated by a simplified matrix structure for each segregated equation.
- Convergence is hampered by neglecting the coupling between neighbouring velocities in the velocity correction equation.
- Convergence depth and rate are strongly dependent on the types and settings of linear solvers, the inertial relaxation values for the corrections, and the closeness of the coupling between the segregated equations.
- Convergence rate and depth both deteriorate with increasing non-orthogonality corrections (from poor cell quality) and increasing cell count.
- Convergence often stalls after 3-5 orders of drop in the residual for typical large-scale industrial problems with $10^7 - 10^9$ cells, even with a small ratio of poor-quality cells.
Segregated SIMPLE scheme for incompressible flow

Typical “good” scenario: near-linear convergence to machine zero, obtained on a good-quality $\sim 5$M-cell grid with coupled flow, turbulence, heat, and radiation.
Segregated SIMPLE scheme for incompressible flow

Typical “poor” scenario: low convergence rates followed by stalling and limit-cycle behaviour, obtained on a poor-quality $\sim 25M$-cell grid with coupled flow, turbulence, heat, and chemical reaction (with the chemical reaction started at 1000 iterations, and the chemical species convergence lines omitted for image clarity).
Introduction

The SIMPLE scheme for incompressible flow

Differentiation of a commercial CFD code

Stability of SIMPLE adjoints

Conclusions
Differentiation using Tapenade. Phase I: code reduction

ACE+ is a multi-physics simulation code with a separate module for the Navier-Stokes Equations. Source is simplified using scripts, then submitted to Tapenade.

- Eliminate the coupling between the flow module and the other physics modules, such as the electromagnetics, chemistry, multi-phase, and cavitation modules. Leave coupling only to the heat-transfer, turbulence, and scalar transport modules.
- Eliminate various internal sub-models within the flow module, including sub-models for, e.g. porous media, etc. and some less relevant BC types, such as moving-boundary.
Differentiation using Tapenade. Phase I: code reduction

ACE+ is a multi-physics simulation code with a separate module for the Navier-Stokes Equations. Source is simplified using scripts, then submitted to Tapenade.

- Eliminate the coupling between the flow module and the other physics modules, such as the electromagnetics, chemistry, multi-phase, and cavitation modules. Leave coupling only to the heat-transfer, turbulence, and scalar transport modules.

- Eliminate various internal sub-models within the flow module, including sub-models for, e.g. porous media, etc. and some less relevant BC types, such as moving-boundary.
Differentiation using Tapenade. Phase I: code reduction

ACE+ is a multi-physics simulation code with a separate module for the Navier-Stokes Equations. Source is simplified using scripts, then submitted to Tapenade.

- Eliminate the coupling between the flow module and the other physics modules, such as the electromagnetics, chemistry, multi-phase, and cavitation modules. Leave coupling only to the heat-transfer, turbulence, and scalar transport modules.
- Eliminate various internal sub-models within the flow module, including sub-models for, e.g. porous media, etc. and some less relevant BC types, such as moving-boundary.
Differentiation using Tapenade. Phase II: relocation or hiding of non-algorithmic source code

Use scripts to accomplish the following:

- Hide the central memory management of ACE+ behind a simple pointer-based access system to reduce checkpoint size.
- Relocate all dynamic memory allocations and deallocations within the flow solution algorithm code to outside this code.
- Relocate any input or output operations, including reading of run-time inputs to outside the solution algorithm source code.

Insert pre-processing pragmas to hide from differentiation:

- Higher-order interpolations and schemes.
- External libraries and interfaces, including those for user-subroutines and parallel processing exchanges. “External” must be used for all Fortran subunits that are “not visible”.


Differentiation using Tapenade. Phase II: relocation or hiding of non-algorithmic source code

Use scripts to accomplish the following:

- Hide the central memory management of ACE+ behind a simple pointer-based access system to reduce checkpoint size.
- Relocate all dynamic memory allocations and deallocations within the flow solution algorithm code to outside this code.
- Relocate any input or output operations, including reading of run-time inputs to outside the solution algorithm source code.

Insert pre-processing pragmas to hide from differentiation:

- Higher-order interpolations and schemes.
- External libraries and interfaces, including those for user-subroutines and parallel processing exchanges. “External” must be used for all Fortran subunits that are “not visible”.

17 / 29
Differentiation using Tapenade. Phase II: relocation or hiding of non-algorithmic source code

Use scripts to accomplish the following:

- Hide the central memory management of ACE+ behind a simple pointer-based access system to reduce checkpoint size.
- Relocate all dynamic memory allocations and deallocations within the flow solution algorithm code to outside this code.
- Relocate any input or output operations, including reading of run-time inputs to outside the solution algorithm source code.

Insert pre-processing pragmas to hide from differentiation:

- Higher-order interpolations and schemes.
- External libraries and interfaces, including those for user-subroutines and parallel processing exchanges. “External” must be used for all Fortran subunits that are “not visible”.
Differentiation using Tapenade. Phase II: relocation or hiding of non-algorithmic source code

Use scripts to accomplish the following:

- Hide the central memory management of ACE+ behind a simple pointer-based access system to reduce checkpoint size.
- Relocate all dynamic memory allocations and deallocations within the flow solution algorithm code to outside this code.
- Relocate any input or output operations, including reading of run-time inputs to outside the solution algorithm source code.

Insert pre-processing pragmas to hide from differentiation:

- Higher-order interpolations and schemes.
- External libraries and interfaces, including those for user-subroutines and parallel processing exchanges. “External” must be used for all Fortran subunits that are “not visible”.


Differentiation using Tapenade. Phase II: relocation or hiding of non-algorithmic source code

Use scripts to accomplish the following:

- Hide the central memory management of ACE+ behind a simple pointer-based access system to reduce checkpoint size.
- Relocate all dynamic memory allocations and deallocations within the flow solution algorithm code to outside this code.
- Relocate any input or output operations, including reading of run-time inputs to outside the solution algorithm source code.

Insert pre-processing pragmas to hide from differentiation:

- Higher-order interpolations and schemes.
- External libraries and interfaces, including those for user-subroutines and parallel processing exchanges. “External” must be used for all Fortran subunits that are “not visible”.
Differentiation using Tapenade. Phase II: relocation or hiding of non-algorithmic source code

Use scripts to accomplish the following:

- Hide the central memory management of ACE+ behind a simple pointer-based access system to reduce checkpoint size.
- Relocate all dynamic memory allocations and deallocations within the flow solution algorithm code to outside this code.
- Relocate any input or output operations, including reading of run-time inputs to outside the solution algorithm source code.

Insert pre-processing pragmas to hide from differentiation:

- Higher-order interpolations and schemes.
- External libraries and interfaces, including those for user-subroutines and parallel processing exchanges. “External” must be used for all Fortran subunits that are “not visible”.

Differentiation using Tapenade. Phase III: Post-differentiation modifications

Use scripts and manual editing to modify the differentiated source code generated by Tapenade as follows:

- Apply bug fixes / corrections to any known errors in the differentiated code. The number of needed corrections has been steadily decreasing as Tapenade evolves and improves.
- Replace the reverse-differentiated linear solver source code with a stand-alone linear solver for the adjoint system.
- Insert informational messages or any needed input-output functionalities into the primal and the differentiated code.
- Introduce code to implement the dot-product test if desired.
From Tapenade 3.6, 2010 to Tapenade 3.14, 2019

The pre-processing steps outlined above were complicated by the limitations of Tapenade 3.6. These limitations required:

- Manual checking of differentiated statements involving pointers to ensure their correctness and to determine the need for any push and pop calls added by Tapenade.
- The need to locate and explicitly replace generated derivative type definitions with the original type definitions.
- The need to replace primal-equivalent generated source code with the original source code.
- The need to correct defects in the differentiated code generally and to record the needed changes so they can be re-introduced after every differentiation of the same code.

Tapenade has been steadily getting better and more reliable. With Release 3.14, no compilation errors are encountered with the differentiated source code and almost no changes or corrections to the differentiated source code are needed.
Verification I

- Comparison of differentiated code at various depths to finite differences
  - Scalar product test tangent-linear vs. adjoint
  - Validation of derivatives of objective wrt control for constructed analytic cases.
Verification I

- Comparison of differentiated code at various depths to finite differences
- Scalar product test tangent-linear vs. adjoint
  - Validation of derivatives of objective wrt control for constructed analytic cases.
Verification I

- Comparison of differentiated code at various depths to finite differences
- Scalar product test tangent-linear vs. adjoint
- Validation of derivatives of objective wrt control for constructed analytic cases.
Verification II

Elementary verification tests were carried out to confirm the correctness of the differentiated code and solution algorithm. These included the following:

1. For a channel with a fixed inflow mass flow rate, it was confirmed that \( \bar{u} = \frac{\partial \dot{m}}{\partial u} \), the derivative of the outflow mass flow rate with respect to the cell-centered velocity components within the channel, was zero.

2. For a channel with symmetry walls, it was confirmed that \( \bar{u} = \frac{\partial (P_{0,\text{inlet}} - P_{0,\text{outlet}})}{\partial u_{\text{inlet}}} \), the drop in the total pressure with respect to the inflow velocity at the inlet, was zero.

3. For a cylinder in two-dimensional uniform flow, it was confirmed that \( \bar{\alpha} = \frac{\partial L}{\partial \alpha} \), the derivative of the lift force \( L \) with respect to the angle of attack \( \alpha \), is zero.
Outline

Introduction

The SIMPLE scheme for incompressible flow

Differentiation of a commercial CFD code

Stability of SIMPLE adjoints

Conclusions
Stability of the primal, I

- Full conv can’t be achieved with SIMPLE except for near-trivial cases
- Conv depends on mesh size: gets worse the finer the mesh
- Practioners of SIMPLE often ’tune’ their overall iterative scheme for specific problem types and even for specific test cases of the same type.
- The tuning is applied for each segregated equation being solved, and the main turning parameters are the inertial (or implicit) relaxation, the linear relaxation, and the linear solver settings, including the choice of solver and convergence tolerance and the maximum number of iterations.
Stability of the primal, II

Example: pipe flow, arbitrary polyhedral mesh. As the mesh size increases beyond $\sim 50M$ cells, the convergence of the primal begins to degrade, both with respect to the rate and the depth of convergence.

Adjoint convergence:
Stability of the adjoint

Returning to the flow in a cylindrical pipe with an arbitrary polyhedral mesh, the convergence of the adjoint begins to degrade well before that of the primal, at around $\sim 5-10M$ cells. Similar behavior is observed for a U-bend test case: the adjoint convergence begins to degrade earlier than the primal convergence with increasing mesh size or decreasing mesh quality.

$$d(\text{Total Pressure Loss})/d(Z\_displacement)$$
Outline

Introduction

The SIMPLE scheme for incompressible flow

Differentiation of a commercial CFD code

Stability of SIMPLE adjoints

Conclusions
Conclusions

- There is currently no method to achieve robust convergence for the workhorse incompressible CFD method: SIMPLE.
- Solutions only partly converge to limit cycles, absence of linear stability affects iterators for discrete adjoints.
- In our so-far limited experience, discrete adjoints around partially converged “fixed-points” seem to show a linear blow-up similar to that found with compressible flow simulations.
- Strong iterative schemes (e.g. memory-expensive Krylov methods with many vectors) can help, but achievable mesh sizes are about an order of magnitude smaller than for the primal.
Conclusions

- There is currently no method to achieve robust convergence for the workhorse incompressible CFD method: SIMPLE.
- Solutions only partly converge to limit cycles, absence of linear stability affects iterators for discrete adjoints.
- In our so-far limited experience, discrete adjoints around partially converged “fixed-points” seem to show a linear blow-up similar to that found with compressible flow simulations.
- Strong iterative schemes (e.g. memory-expensive Krylov methods with many vectors) can help, but achievable mesh sizes are about an order of magnitude smaller than for the primal.
Conclusions

• There is currently no method to achieve robust convergence for the workhorse incompressible CFD method: SIMPLE.
• Solutions only partly converge to limit cycles, absence of linear stability affects iterators for discrete adjoints.
• In our so-far limited experience, discrete adjoints around partially converged “fixed-points” seem to show a linear blow-up similar to that found with compressible flow simulations.
• Strong iterative schemes (e.g. memory-expensive Krylov methods with many vectors) can help, but achievable mesh sizes are about an order of magnitude smaller than for the primal.
Conclusions

- There is currently no method to achieve robust convergence for the workhorse incompressible CFD method: SIMPLE.
- Solutions only partly converge to limit cycles, absence of linear stability affects iterators for discrete adjoints.
- In our so-far limited experience, discrete adjoints around partially converged “fixed-points” seem to show a linear blow-up similar to that found with compressible flow simulations.
- Strong iterative schemes (e.g. memory-expensive Krylov methods with many vectors) can help, but achievable mesh sizes are about an order of magnitude smaller than for the primal.
Alternatives?

- Use a different, more stable discretisation. But try to convince the user community.
  - Borrow from continuous adjoint methods: add a non-linear stabilisation that focuses on adjoint instabilities.
  - Need to understand the stability of these systems, which is already challenging for the primal.
  - But then gradients are no longer exact, loss of consistency.
  - Add a term to the primal that impairs primal accuracy, but restores consistency?
Alternatives?

- Use a different, more stable discretisation. But try to convince the user community.
- Borrow from continuous adjoint methods: add a non-linear stabilisation that focuses on adjoint instabilities.
  - Need to understand the stability of these systems, which is already challenging for the primal.
  - But then gradients are no longer exact, loss of consistency.
  - Add a term to the primal that impairs primal accuracy, but restores consistency?
Alternatives?

- Use a different, more stable discretisation. But try to convince the user community.
- Borrow from continuous adjoint methods: add a non-linear stabilisation that focuses on adjoint instabilities.
- Need to understand the stability of these systems, which is already challenging for the primal.
  - But then gradients are no longer exact, loss of consistency.
  - Add a term to the primal that impairs primal accuracy, but restores consistency?
Alternatives?

• Use a different, more stable discretisation. But try to convince the user community.
• Borrow from continuous adjoint methods: add a non-linear stabilisation that focuses on adjoint instabilities.
• Need to understand the stability of these systems, which is already challenging for the primal.
• But then gradients are no longer exact, loss of consistency.
• Add a term to the primal that impairs primal accuracy, but restores consistency?
Alternatives?

- Use a different, more stable discretisation. But try to convince the user community.
- Borrow from continuous adjoint methods: add a non-linear stabilisation that focuses on adjoint instabilities.
- Need to understand the stability of these systems, which is already challenging for the primal.
- But then gradients are no longer exact, loss of consistency.
- Add a term to the primal that impairs primal accuracy, but restores consistency?
Acknowledgements

This work builds on the first work on differentiating ACE+ done by Dominic Jones.

Parts of this work have been conducted within the FlowHead and About Flow and projects at Queen Mary University of London

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

We have received funding from the European Union’s Seventh Framework Programme and Horizon 2020 under Grant Agreement Nos. 218626 and 317006.