Checkpointing with time gaps for unsteady adjoint CFD

Jan C. Hueckelheim*
Email: j.c.hueckelheim@qmul.ac.uk

Mateusz Gugala
Email: m.gugala@qmul.ac.uk

Jens-Dominik Mueller
Email:j.mueller@qmul.ac.uk

School of Engineering and Materials Science (SEMS)
Queen Mary University of London, Mile End Road, London, E1 4NS, UK

Summary

Gradient-based optimisation using adjoints is an increasingly common approach for industrial flow applications. For cases where the flow is largely unsteady however, the adjoint method is still not widely used, in particular because of its prohibitive computational cost and memory footprint. Several methods have been proposed to reduce the peak memory usage, such as checkpointing schemes or checkpoint compression, at the price of increasing the computational cost even further. We investigate incomplete checkpointing as an alternative, which reduces memory usage at almost no extra computational cost, but instead offers a trade-off between memory footprint and the fidelity of the model. The method works by storing only selected physical time steps and using interpolation to reconstruct time steps that have not been stored. We show that this is enough to compute sufficiently accurate adjoint sensitivities for many relevant cases, and does not add significantly to the computational cost. The method works for general cases and does not require to identify periodic cycles in the flow.

Keywords: *Unsteady adjoint, checkpointing, checkpoint compression, gradient-based optimisation*

1 Introduction

The adjoint method is commonly used in academia and industry to compute the derivative of a cost function with respect to its design variables. Its greatest appeal lies in the fact that the computational cost is constant in the number of design variables, in contrast to simpler approaches such as finite differences or tangent-linear derivatives. This makes the method feasible for industrial applications with rich design space.[1]

Many real-world problems however still present a challenge for the adjoint method. A particular problem is severe unsteadiness, as can be found in turbines, including wind turbines, aircraft wings in high-lift configuration, car engines and many more.[2] The adjoint method has been formulated for this kind of problem in frequency[3] and temporal space,[4] but requires the storage of the full flow history, resulting in prohibitive memory requirements in most cases.

A well-known way to mitigate this problem is the REVOLVE checkpointing algorithm.[5] It stores checkpoints only at carefully chosen time steps, and recomputes each time step when it is needed, starting from the time steps that have been stored. Another approach that has recently been proposed[6] is the compression of checkpoints. Both ideas have one thing in common: the memory requirements are relaxed at the cost of increased computational expense. Furthermore, lossless data compression does not offer large savings in storage space,[7] and so the method becomes more useful if lossy compression is used, resulting in errors in the reconstruction of the primal flow field.

We investigate incomplete checkpointing as an alternative, which reduces memory usage at no extra computational cost, but instead offers a direct trade-off between memory footprint and the fidelity of the model. We use a dual timestepping scheme in which the inner iterations are fully converged, so that only physical time steps need to be stored and the adjoint field can be reconstructed based on the fully converged checkpoint, which preserves the accuracy of the result if the innter iteration was fully converged.[8] In addition, we store only selected physical time steps and use interpolation to reconstruct time steps that have not been stored.

The scheme comes at negligible cost for linear interpolation, and at no cost at all in case of constant interpolation between checkpoints. In particular, the reconstruction from data available in memory is significantly faster than reading the checkpoint from disk, which would be another possible (but slow) way of addressing the memory limitations. Finally, our method does not require any assumptions about the flow such as periodicity.

2 Background

We use an unsteady RANS solver with Spalart-Allmaras turbulence model, BDF2 dual time stepping and an implicit solver to converge the inner iterations which was presented in.[9] The adjoint solver is generated using the automatic differentiation tool Tapenade[10] with some hand-coded optimisations for improved speed.[11]

2.1 Solving the flow equations

The unsteady RANS flow equations can be written as

$$\frac{\partial U}{\partial t} + R(U) = 0 \tag{1}$$

and can be discretised using a third-order accurate BDF2 time marching scheme as

$$\frac{\partial U}{\partial t} + R(U_t) = \frac{U_{t-2} - 4 \cdot U_{t-1} + 3 \cdot U_t}{2\Delta t} + R(U_t) \tag{2}$$

$$:= \hat{R}(U_{t-2}, U_{t-1}, U_t) \tag{3}$$

The above system can be evolved in time by solving the linearised system for U_k and successively updating the converged flow solution U_t at time t

$$\left[\frac{\partial \hat{R}(U_{t-2}, U_{t-1}, U_k)}{\partial U_k}\right] \delta U_k = -\hat{R}(U_{t-2}, U_{t-1}, U_k) \tag{4}$$

$$U_t = U_{t-1} + \delta U_k \tag{5}$$

2.2 Solving the adjoint equations

The unsteady adjoint system can be written as

$$-\frac{\partial v}{\partial t} + \underbrace{\left(\frac{\partial R}{\partial U}\right)^T v - \left(\frac{\partial J}{\partial U}\right)^T}_{:=R_v} = 0 \tag{6}$$

and can, like the primal equation, be discretised using BDF2 as

$$-\frac{v^{t-2} - 4 \cdot v^{t-1} + 3 \cdot v^t}{2\Delta t} + R_v(v^t) \tag{7}$$

$$:= \hat{R}_v(U_{t-2}, U_{t-1}, U_t) \tag{8}$$

and solved using the same method as the primal equation.

The solution of the adjoint equation requires the history of the flow solution U_t at each time step for the calculation of R_v and the preconditioning matrix P^T.

This flow field can be stored during the flow solution and loaded during the adjoint solution, recomputed by running the flow solver again (e.g. following the REVOLVE algorithm), restored approximately from a compressed data, or reconstructed using interpolation, following our new approach.

2.3 Physical checkpointing

We use an approach in which only the physical time steps are stored during the primal computation and restored during the adjoint computation, as presented in.[12] The memory requirements are orders of magnitude smaller compared to the brute-force method of storing every iteration. The method is illustrated in the below pseudo-code.

```
n ← 0;
U_0, V_0 ← initial guess;
while t < t_final do // primal loop
    n ← 0;
    while R(U_{t,n}) > cutoff do // primal loop
        U_{t,n+1} ← flow_pseudostep(U_{t,n});
        n ← n + 1
    end
    t ← t + Δt;
    U_{t+1,0} ← U_{t,n}; // init for next step
    store(U_t);
end
while t > t_init do // adjoint loop
    load(U_t);
    V_{t,0} ← V_{t+1,n}; // init for next step
    t ← t − Δt;
    while R(V_t) > cutoff do // primal loop
        V_{t,n+1} ← adjoint_pseudostep(V_{t,n}, U_t);
        n ← n + 1
    end
end
```

Function Dual timestepping with physical checkpointing

The challenge in this approach lies in the need to fully converge the inner loop so that an efficient adjoint method for fixed-point loops can be used.[8] To address this, we use an implicit solver with geometric multigrid and ILU preconditioning to converge in an acceptable time.[9]

The calls to `store()` and `load()` in this algorithm are replaced by calls to augmented routines `gappyStore()` and `gappyLoad()` in order to implement the method proposed in this paper.

3 Checkpointing with gaps

3.1 Storing

The routine to store the current flow state is augmented by a logic that selects certain snapshots worth storing, which are denoted by the set of stored time steps T_s which are a subset

of all time steps T.

if $t \in T_s$ **then** `// primal loop`
  store(U_t);
end

Function gappyStore(t)

In the simplest incarnation of this method, T_s would contain only every n-th time step. In some special cases it might be beneficial to vary the checkpoint density over time, e.g. to capture a particular phenomenon with a higher accuracy. This was not investigated in this work.

If our method is regarded as a very simple form of data compression, then the data compression ratio is $\|T\|/\|T_s\|$. Obviously we get better compression ratios if we store fewer time steps. For evenly spaced snapshots as suggested above, we obtain a compression ratio $\|T\|/\|T_s\| = n$.

3.2 *Reconstruction*

Checkpoints that have not been stored need to be reconstructed. Let t denote the time for which a checkpoint needs to be reconstructed. Also, let t^+ and t^- denote the unique time steps for which all of the following conditions hold:

$$\begin{aligned} & t^+, t^- \in T_s \\ & \nexists t^* : t < t^* < t^+ \\ & \nexists t^* : t^- < t^* < t \end{aligned}$$

In other words, t^+ and t^- are the closest stored time steps just after and before t, respectively. Furthermore, let t_n be defined by

$$t_n = \begin{cases} t^- & \text{if } |t-t^-| < |t-t^+| \\ t^+ & \text{else} \end{cases}$$

i.e. the time step denoted by t_n is either t^+ or t^-, whichever is closer to t.

We can then implement the `gappyLoad()` routine for a chosen interpolation order O as follows:

if $t \in T_s$ **then** `// t was stored`
  return ***load***(U_t);
else
  if $O = 1$ **then** `// linear`
    $U^- \leftarrow$ ***load***(U_{t^-});
    $U^+ \leftarrow$ ***load***(U_{t^+});
    return $U^- + \frac{t-t^-}{t^+-t^-} \cdot (U^+ - U^-)$
  else
    return ***load***(U_{t_n});
  end
end

Function gappyLoad(t)

It is worth noting that if gaps between two stored time steps t^- and t^+ are large and therefore there are several calls to `gappyLoad()` where t^- and t^+ do not change between calls, this method can be implemented much more efficiently by storing most of the intermediate results.

The routine could be implemented for higher orders of interpolation, taking into account more of the surrounding stored time steps. Second order interpolation could be used to match the order of the BDF2 time marching scheme used in the flow computation.

4 Test case

To test our method, we use a RAE2822 aerofoil . The trailing edge has been truncated at 5% cord length to provoke vortex shedding. The freestream velocity is 0.2 Ma, at 0° angle of attack. We use the Spalart Allmaras turbulence model. The mesh has around $300k$ cells, the solver is node-centred and uses 4 levels of geometric multigrid for faster convergence. The overall geometry and the vortex shedding can be seen in Figs. 1 and 2.

We use a time step size of 0.2 ms, which corresponds to approximately 30 checkpoints for every flow period. We run several tests, in which we store checkpoints at every 1 (reference result), 2, 4, 8, 16 and 32 time steps. Note that while this flow does exhibit clear periodic cycles, our solver does not make use of this periodicity in any way and our findings should therefore apply to non-periodic flow as well.

The objective function is defined as the total drag of the aerofoil . The aerofoil surface nodes' displacement in surface normal direction is used as design vector. The spring analogy model is used to project volume sensitivities onto the surface. No actual optimisation step is performed, as we only consider the surface sensitivities in the assessment of the result quality.

All sensitivity results will be compared time-averaged over a window of 10 flow periods. We try both linear as well as constant interpolation for the checkpoint reconstruction.

As a reference, we will also show results of a simulation where both primal and adjoint solutions have been carried out with a time step size that is larger than the initial run by a factor of 2, 4, 8, 16 and 32. We expect that there is a benefit of running the primal solution time-accurately with fine time steps and storing selected snapshots of this high-fidelity primal solution, compared to just running both primal and adjoint with coarse time steps that are too large to be time-accurate.

5 Results

For the truncated aerofoil case we perform two parameter studies. The flow parameters are the same in both cases. In the first set, which will be called *gappy checkpointing*, we use our proposed method of running the primal flow solver with a time step size that is sufficient to be time-accurate. The adjoint solver will use interpolation to reconstruct the checkpoints that were not stored.

In a reference test set, which will be called coarse set, we will run both the primal and adjoint solver with larger time steps. The unsteady flow will not be fully resolved. The time step sizes in the coarse set will be chosen to match the size of gaps between stored checkpoints in

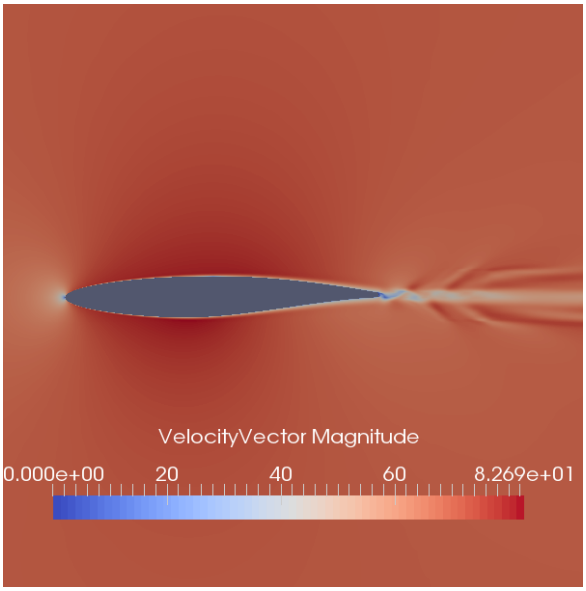

Figure 1: Vortex shedding behind the trailing edge of the truncated aerofoil . Note that this flow is poorly converged, the final version of the paper will have converged results.

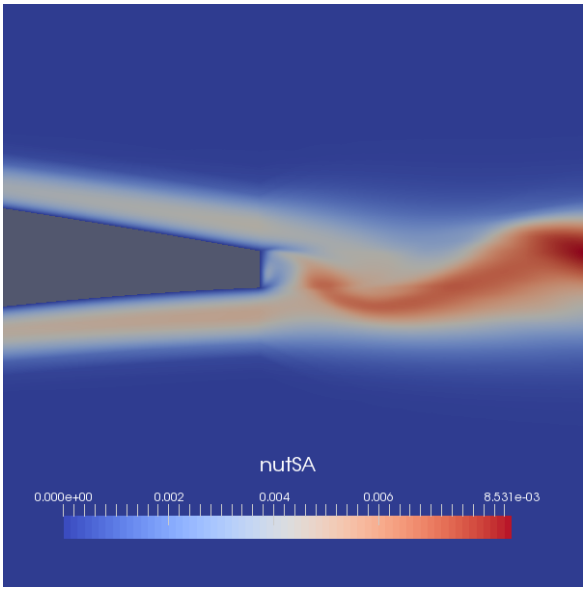

Figure 2: Zoom into the trailing edge area. We can see the flow detaching at the corners of the truncated trailing edge and producing vortices.

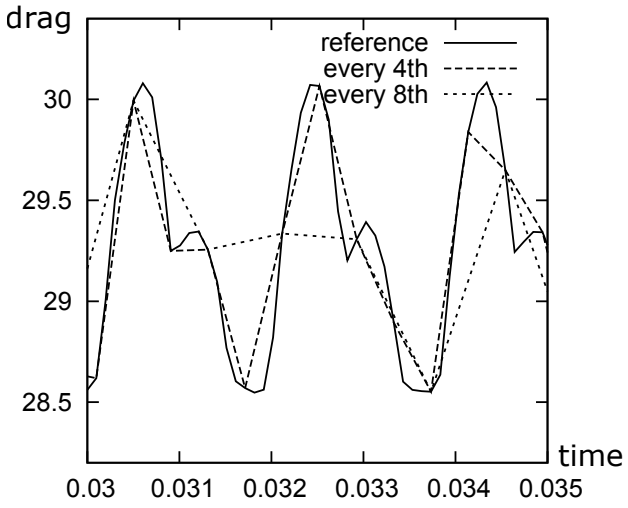

Figure 3: Drag history over three shedding periods, with the reference result, and samplings at every 4th or every 8th timestep. While the sampling at every 4th step follows the general trend reasonably well, the sampling at every 8th step is drastically different. The adjoint solution is based on a sampling that is just as coarse. Note that for this simulation, we used only 10 time steps per shedding period to speed up the simulation. In the final version of the paper, the simulation will be carried out with finer time steps to resolve the shedding accurately in time.

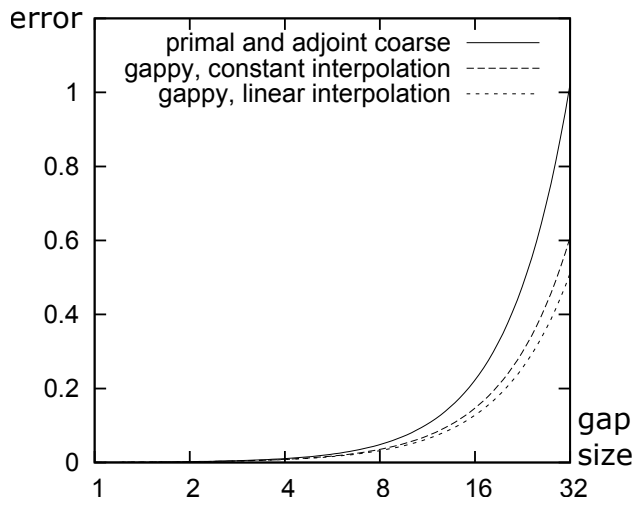

Figure 4: Illustration of the error trend that we expect to see. The error (y-axis) will grow as the time resolution is decreased by a factor (x axis). At some point, the solver will not be able to converge any more. Using the full time resolution and then using a coarser sampling for the adjoint solver is expected to be significantly better than just using a coarser time resolution for the primal and adjoint solver. Both would result in the same memory footprint. Note that this plot does not yet show any actual measurement results, this will be included in the final paper.

our gappy set. This will allow us to quantify how much is gained by running the primal flow with full temporal solution and then discarding some of the checkpoints due to memory constraints, as compared to using a lower temporal resolution in the first place for both primal and adjoint.

6 Conclusion, future work

We show that gaps in the stored time trajectory are an easy and effective way of reducing the memory footprint of unsteady adjoint calculations. The effect on the sensitivity accuracy is acceptable for many industrial cases, making our approach worth considering as an alternative to lossy checkpoint compression, with a significantly smaller implementation effort and computational cost.

In the future, we plan to investigate higher-order interpolation in time, non-uniform checkpoint storing intervals, and checkpointing that is incomplete in space, e.g. by storing only a coarse grid result from our geometric multigrid solver as a checkpoint.

7 Acknowledgement

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no [317006].

This research utilised Queen Mary's MidPlus computational facilities, supported by QMUL Research-IT and funded by EPSRC grant EP/K000128/1.

References

[1] Giles, M., Pierce, N., Giles, M., and Pierce, N. *Adjoint equations in CFD - Duality, boundary conditions and solution behaviour*. American Institute of Aeronautics and Astronautics 2015/05/03 (1997).

[2] Lee, B. J. and Liou, M.-S. Unsteady adjoint approach for design optimization of flapping airfoils. *AIAA Journal* **50**(11), 2460–2475 2015/05/03 (2012).

[3] Nadarajah, S. and Jameson, A. Optimum shape design for unsteady three-dimensional viscous flows using a nonlinear frequency-domain method. *Journal of Aircraft* **44**(5), 1513–1527 2015/05/03 (2007).

[4] Rumpfkeil, M. and Zingg, D. *A General Framework for the Optimal Control of Unsteady Flows with Applications*. American Institute of Aeronautics and Astronautics 2015/05/03 (2007).

[5] Griewank, A. and Walther, A. Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Softw.* **26**(1), 19–45 March (2000).

[6] Tim Wildey, E. C. C. and Shadid, J. Adjoint based a posteriori error estimates using data compression. In VI International Conference on Adaptive Modeling and Simulation, J. P. Moitinho de Almeida, P. D ıez, C. T. and Parés, N., editors, (2013).

[7] Ratanaworabhan, P., Ke, J., and Burtscher, M. Fast lossless compression of scientific floating-point data. In *Proceedings of the Data Compression Conference*, DCC '06, 133–142 (IEEE Computer Society, Washington, DC, USA, 2006).

[8] Christianson, B. Reverse aumulation and imploicit functions. *Optimization Methods and Software* **9**(4), 307–322 (1998).

[9] Xu, S., Radford, D., Meyer, M., and Müller, J. D. Stabilisation of discrete adjoint solvers. *Journal of Computational Physics* **submitted** (2014).

[10] Hascoët, L. and Pascual, V. The Tapenade Automatic Differentiation tool: principles, model, and specification. Research Report RR-7957, May (2012).

[11] Christakopoulos, F., Jones, D., and Müller, J.-D. Pseudo-timestepping and verification for automatic differentiation derived {CFD} codes. *Computers Fluids* **46**(1), 174 – 179 (2011). 10th {ICFD} Conference Series on Numerical Methods for Fluid Dynamics (ICFD 2010).

[12] Hueckelheim, J., Xu, S., Gugala, M., and Müller, J.-D. *Time-averaged steady vs. unsteady adjoint: a comparison for cases with mild unsteadiness*. American Institute of Aeronautics and Astronautics 2015/05/03 (2015).