Discrete Adjoint Approaches for CHT Applications in OpenFOAM

Markus Towara*
Software and Tools for Computational Engineering (STCE)
RWTH Aachen University, 52062 Aachen, Germany
Email: towara@stce.rwth-aachen.de

Johannes Lotz
Software and Tools for Computational Engineering (STCE)
RWTH Aachen University, 52062 Aachen, Germany
Email: lotz@stce.rwth-aachen.de

Uwe Naumann
Software and Tools for Computational Engineering (STCE)
RWTH Aachen University, 52062 Aachen, Germany
Email: naumann@stce.rwth-aachen.de

Summary

Conjugate Heat Transfer (CHT) simulations allow the prediction of complex interactions between fluid and solid mediums. Our application is the optimization of heat transfer between heat sinks and a cooling fluid, used to extract the heat from server infrastructure. Adjoint methods allow the optimization of high dimensional parameter settings, using sensitivity information. Compared to classical approaches to sensitivity generation, e.g. finite differences, a significant improvement in run time can be achieved, as the complexity of deriving the sensitivity scales with the output dimension, instead of the input (parameter) dimension. As an initial prove of concept, our discrete adjoint OpenFOAM framework has been extended to facilitate the differentiation of the `chtMultiRegionSimpleFoam` solver. To combat prohibitive memory loads a checkpointing approach is used. We will present results of the heat transfer of a copper heat sink immersed in glycol.

Keywords: *CHT, CFD, Algorithmic Differentiaton, OpenFOAM*

1 Introduction

Conjugate heat transfer (CHT) simulations allow the prediction of complex interactions between solids and fluids. A discussion on the history of CHT methods can e.g. be found in [1]. Previous studies with heat transfer using the continuous or adjoint method include [2–4].

The paper builds on our previous works [5–7] to introduce Algorithmic Differentiation (AD) into OpenFOAM. The outline of this paper is as follows. In Section 2 we briefly introduce the CHT problem formulation, as utilized by OpenFOAM. Further, in Section 3 we introduce the basic approaches of AD. In Sections 4 and 5 we then focus on the implementation of checkpointing techniques for the CHT problem and how they differ from our existing implementations for singe domain solvers. Methods for identifying issues in checkpointing implementations are discussed. In Section 6 further details, required for obtaining accurate shape sensitivities, are discussed. Section 7 will introduce a CHT case of a copper heat sink, immersed in glycol. Preliminary sensitivity results are shown. A more involved analysis of the results as well as run time analysis and parallel scaling results will be shown in the full paper.

2 CHT Foundations

The CHT problem is characterized by the discretization and solution of multiple PDEs on different domains. In the fluid domain, the incompressible Navier-Stokes equations, including the momentum (1), mass (2), and energy conservation (3) equations are solved. In OpenFOAMs `multiRegionSimpleFoam` solver this is achieved by discretizing the equations using the finite volume method (FVM) and applying the SIMPLE algorithm, implicitly coupling the pressure to the velocities. [8].

On the solid domain, the energy equation simplifies to the less complex Poisson equation (4), that can be solved to predict the temperature distribution within the solid. The

governing equations are outlined below, for details and how they are implemented and discretized within OpenFOAM see [9].
Fluid domain:

$$(\mathbf{u}\otimes\nabla)\mathbf{u} = \nu\nabla^2\mathbf{u} - \frac{1}{\rho}\nabla p + \mathbf{b} \tag{1}$$

$$\nabla\cdot\mathbf{u} = 0 \tag{2}$$

$$\nabla\cdot(\rho c_p \mathbf{u} T) = \nabla\cdot(k\nabla T) + \dot{q}_F\,. \tag{3}$$

Solid domain:

$$k\nabla^2 T = -\dot{q}_S \tag{4}$$

Here $\mathbf{u}$ denotes velocity, p pressure, ρ fluid density, ν kinmatic viscosity, T temperature, c_P specific heat capacity, k heat conductivity, $\dot{q}_F$ external heat fluxes into the fluid domain, and $\dot{q}_S$ external heat fluxes into the solid domain.

Assuming a positive temperature gradient between solid and fluid, the fluid convects heat energy away from the solid surface, thus effectively cooling the boundary and interior of the solid domain. The solution of the Navier-Stokes and Poisson equation are only loosely coupled, that is both equations are discretized and solved for independently and are only coupled by the shared temperature boundary conditions. This helps with stability and complexity of individual simulation steps, but it can lower the overall convergence rate. In our experience, the under-relaxation factor for the temperature in the solid domain can be chosen close or equal to one, greatly improving convergence of the solid temperature field.

The interface between solid and fluid phases can either be a conforming mesh, where both phases share the same patch with identical boundary faces (with flipped normals) or a non-conforming mesh with incompatible boundary faces. In this case the values can be interpolated from the fluid to solid patch and vice-versa. Both cases can be differentiated by AD without modifications, however the interpolation adds a non-trivial amount of computational work.

3 Algorithmic Differentiation

We consider the optimization problem $J(\mathbf{x})$ for $J : \mathbb{R}^n \to \mathbb{R}$, where each function evaluation $J(\mathbf{x})$ comprises the solution of the discrete Navier-Stokes equations and the coupled heat equations, forming a very large system of parameterized nonlinear equations. First-order AD assumes J to be at least once continuously differentiable at all points of interest. For a given implementation of the primal objective $y = J(\mathbf{x})$, a corresponding (first-order) adjoint code computes

$$\bar{\mathbf{x}} = \bar{J}(\mathbf{x},\bar{y}) \equiv \nabla J^T\cdot\bar{y},$$

where $\bar{\mathbf{x}} \in \mathbb{R}^n$ and $\bar{y} \in \mathbb{R}$ are the adjoints of $\mathbf{x}$ and y respectively. Using the adjoint mode of AD, the gradient can be obtained at a computational cost of $O(1)\cdot Cost(J)$, where $Cost(J)$ denotes the computational cost of a single evaluation of J. The actual run time factor depends on various parameters, including the mode of differentiation (continuous vs. discrete adjoint), the expertise of the adjoint code developer, and the quality of the AD software tool, if one is used.

For reference, the computational cost to compute the same gradient using finite differences or the tangent mode of AD is $O(n)\cdot Cost(J)$. For our CHT applications we use the adjoint model, as typically a very large number of inputs are mapped onto a single output.

Conceptually, AD is based on the fact that the given implementation of the primal objective as a computer program can be decomposed at run time into a *single assignment code*.

$$\begin{aligned}&\text{for } j = n,\ldots,n+p\\ &\quad v_j = \varphi_j(v_i)_{i\prec j}\quad ,\end{aligned}$$

where $i \prec j$ denotes a direct dependence of the variable v_j on v_i. The result of each *elemental function* φ_j is assigned to a unique auxiliary variable v_j. The *n independent inputs* $x_i = v_i$, for $i = 0,\ldots,n-1$, are mapped onto the *dependent output* $y = v_{n+p}$. The values of p *intermediate variables* v_k are computed for $k = n,\ldots,n+p-1$.

The primal code is augmented with instructions for storing data which is required for the reversal of the data flow and for the computation of the local partial derivatives $\frac{\partial\varphi_j}{\partial v_i}$, for $j = n,\ldots,n+p$ and $i \prec j$. A data structure commonly referred to as *tape* is used for this purpose. This *(augmented) forward section* of the adjoint code is succeeded by the *reverse section* propagating adjoints $\bar{v}_i$ for all v_i in reverse order, that is, for $i = n+p-1,\ldots,0$:

$$\left.\begin{aligned}&\text{for } j = n,\ldots,n+p+m-1\\ &\quad v_j = \varphi_j(v_i)_{i\prec j}\end{aligned}\right\}\text{forward section}$$

$$\left.\begin{aligned}&\text{for } i = n+p-1,\ldots,0\\ &\quad \bar{v}_i = \sum_{j:i\prec j}\frac{\partial\varphi_j}{\partial v_i}\cdot\bar{v}_j\end{aligned}\right\}\text{reverse section} \tag{5}$$

Note that the v_j computed in the forward section are potentially required as arguments of local partial derivatives within the reverse section. They are read in reverse with respect to the original order of their evaluation. The additional persistent memory requirement of the adjoint code becomes $O(n+p)$. The efficient reversal of the data flow is among the main challenges in adjoint AD. It is responsible for black-box adjoint AD typically not being applicable to large-scale numerical simulations. The available persistent memory may simply not be large enough [10].

For our implementation we use the tape based AD tool `dco/c++` [11], which implements an operator overloading approach of AD, as opposed to source code transformation.

4 Checkpointing Considerations

Checkpointing is an important technique to reduce the memory demands of the adjoint mode of AD by trading memory against run time [12]. Only parts of the program are adjoined at a time, a previous state is then restored from a checkpoint and a different part of the program is taped and adjoined. Let $\mathbf{x}^i$ be the state at an iteration step i. E.g. for the incompressible laminar Navier-Stokes equations the state is the combination of velocity, pressure and face flux fields $\mathbf{x} = (\mathbf{U}, \mathbf{p}, \phi)$. The general procedure to adjoin a single iteration step f^i, transforming state $\mathbf{x}^i$ into $\mathbf{x}^{i+1}$ can be formalized as follows. We assume at least one checkpoint at $\mathbf{x}^0$ is available. We further assume that the adjoints $\mathbf{x}^{i+1}$ are already known from previous applications of the procedure.

- Restore state $\mathbf{x}^j$ where $j \leq i$ and $\min_{c_j \in C}(i-j)$
- If $j < i$ passively recalculate $\mathbf{x}^i$
- Register state $\mathbf{x}^i$ as inputs. If no other statements are executed in this step, this has the added benefit, that the adjoints $\bar{\mathbf{x}}^i$ will be located in memory contiguously, once they are calculated.
- Calculate and tape iteration step i: $\mathbf{x}^{i+1} = f^i(\mathbf{x}^i)$.
- Register state $\mathbf{x}^{i+1}$ as outputs. If no other statements are executed in this step, this has the added benefit, that the adjoints $\bar{\mathbf{x}}^{i+1}$ can be written to memory contiguously.
- Restore previously calculated adjoints $\bar{\mathbf{x}}^{i+1}$ into the tape.
- Interpret tape, calculating $\bar{\mathbf{x}}^i = \left(\frac{\partial f(\mathbf{x}^i)}{\partial \mathbf{x}^i}\right)^T \cdot \bar{\mathbf{x}}^{i+1}$ and $\bar{\alpha} = \bar{\alpha} + \left(\frac{\partial f(\mathbf{x}^i)}{\partial \alpha}\right)^T \cdot \bar{\mathbf{x}}^{i+1}$.
- Extract calculated adjoints $\bar{\mathbf{x}}^i$ from tape
- Reset tape.

This procedure can be repeated until all iteration steps have been adjoined and all desired adjoints $\bar{\alpha}$ have been accumulated.

Compared to the checkpointing procedure already outlined in [5], the complexity is increased for CHT applications in OpenFOAM by the following: Firstly the mesh is decomposed into multiple regions, corresponding to solid and fluid phases. Secondly, the CHT implementation and case setup uses boundary conditions not previously studied in the context of our discrete adjoint implementation. Two of these offending boundary conditions are outlined below. The `fixedFluxPressure` condition for `p_rgh` inherits from the `fixedGradient` boundary condition. Thus the boundary field on patches declared with the `fixedFluxPressure` are of type `fixedGradientFvPatchField`. The `fixedGradientFvPatchField` class declares a private data member `Field<Type> gradient_`, storing the surface normal (pressure) gradient. This is easily overlooked, as the gradient is private to the specific implementation of the boundary condition and is not part of the general `fvPatchField` boundary condition it inherits from. The `fixedFluxPressure` boundary condition iteratively updates the gradient, making the gradient part of the state. Thus, it needs to be checkpointed. The same principle applies to the `mixedFvPatchField` class, that is utilized by the `inletOutlet` boundary condition. The `inletOutlet` condition locally switches between the fixed value and fixed gradient boundary condition, depending on flow direction. It is commonly used to prohibit backflow. Similarly, the `mixedFvPatchField` class stores a private `scalarField volumeFraction_`, which in the context of `inletOutlet` switches between a fixed gradient and fixed value. If this field is not checkpointed, wrong primals are calculated during the repeated passive evaluations.

Table 1 lists the quantities that were identified as being part of the state and need to be checkpointed for the `chtMultiRegionSimpleFoam` solver using the `kOmegaSST` turbulence model. This is basically a complete list of OpenFOAMs `IOobject` registry with some additional quantities.

Our checkpointing interface needs the possibility to advance the iteration state one step at a time (from a given state). Previously this was achieved by holding references to all fields (locally) created in the OpenFOAM solvers `main()` routine in a separate class structure. This involves a lot of code duplication and additional work to adopt the checkpointing procedure to different solvers. Therefore, we recently switched to an implementation where the iteration step is captured in an C++11 lambda expression, which allows to explicitly or implicitly capture the variables local to the main routine. By wrapping the created lambda function into a `std::function<T>` structure it can be passed to the checkpointing interface. Thus, the simulation state can be advanced whenever necessary by calling the created function. As checkpointing schemes we support Revolve [13] and a simple equidistant scheme.

As stated earlier, the interpolation between different meshes can add a significant overhead to the required tape memory. For a static (non-moving) mesh the interpolation coefficients are constant. However, the interpolation is currently recorded in the tape during each iteration step. The calculation of the adjoints of the interpolation can potentially be handled more efficiently using automatic or manual local pre-accumulation [12].

5 Verifying the Checkpointing Implementation

A robust checkpointing implementation is important, as it also builds the foundation for our more advanced reverse accumulation [14] and piggybacking [12] solvers. For the verification of the correctness of the checkpointing implementation and easier interpretation of issues, we

Table 1: Quantities that need to be checkpointed for the solid and fluid phases. All quantities are either volScalarFields, volVectorFields or surfaceScalarFields, with the exception of cumulativeContErr. The checkpoint for cumulativeContErr is not strictly required, as it is not connected to the parameters, however it will trip the debugging safeguards of the AD tool.

region	type	name
global	scalar	cumulativeContErr
fluid	volScalarField	gh
fluid	volScalarField	thermo:mu
fluid	volScalarField	alphat
fluid	volScalarField	thermo:psi
fluid	volScalarField	nut
fluid	volScalarField	yWall
fluid	volScalarField	p
fluid	volScalarField	T
fluid	volScalarField	e
fluid	volScalarField	rho
fluid	volScalarField	k
fluid	volScalarField	omega
fluid	volScalarField	p_rgh
fluid	volScalarField	thermo:rho
fluid	volScalarField	thermo:alpha
fluid	volVectorField	U
fluid	surfaceScalarField	phi
fluid	surfaceScalarField	ghf
solid	volScalarField	thermo:mu
solid	volScalarField	betavSolid
solid	volScalarField	thermo:psi
solid	volScalarField	thermo:rho
solid	volScalarField	p
solid	volScalarField	T
solid	volScalarField	thermo:alpha
solid	volScalarField	h

implemented three different debug modes for our AD tool dco/c++. Besides allowing to find issues in the current cases, these modes can also help to prevent future problems. They can identify assignments which not yet actively influence the numerics, but might become relevant for different activity paths. The modes are described below and illustrated with brief examples.

Figure 1 shows the conceptual tape layout for a simple example code with two assignments. For each assignment an entry in the stack is created, storing the partial derivatives w.r.t. the variables on the right hand side of the assignment, as well as pointers to the location in the adjoint vectors which need to be incremented by the product between the partial derivative and the incoming adjoints during the tape interpretation. Note how edges in the tape always point upwards, propagating adjoint information backwards through the tape. For a more complete discussion of the tape implementation see [7, 15].

The debug modes can e.g. be applied to the iteration loop, to check if any part of the current step depends on any variables outside of the state. If this is the case, there is dependence on data which might not be correct once the chronological order of iterations is broken for the recomputation of states. The challenge in essence is to capture the full state necessary to accurately recompute future states, without storing unneeded intermediate values. For a complex code, as OpenFOAM, this is not a trivial task, exemplified by the amount of fields listed in Table 1.

Overwrite barrier: After each assignment into a floating point variable already known to the tape, its associated tape index increases in order to handle the name aliasing of the variable. After an *overwrite* barrier is introduced, variables that were defined before the barrier are not allowed to be overwritten. An exception is raised if such a variable is modified. Whenever a variable with global scope is modified within the iteration, it has to become part of the checkpointed state, else it will not be restored to its correct value when a previous checkpoint is loaded. By placing the barrier in front of the iteration it can be used to determine if variables not part of the current iteration or state are overwritten.

Forward barrier: A window in the adjoint vector is declared, to which no partial edges are allowed to point. This means that the primal variables corresponding to these tape positions are not allowed to occur on the right hand side of an assignment. To allow the adjoint accumulation of global parameters, the corresponding tape entries can be moved to before the window. These parameters must not be overwritten during the iteration phase. The barrier is enforced during the (augmented) *forward* evaluation of the code.

Reverse barrier: As forward barrier, but takes the actual

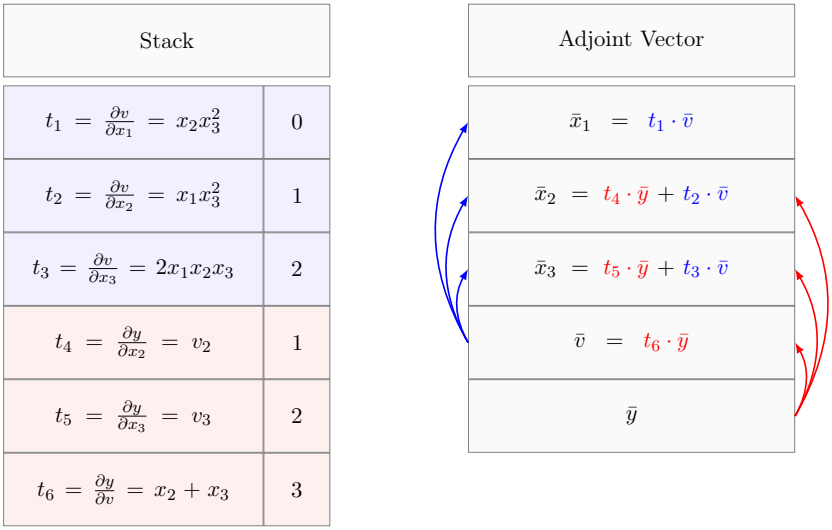

Figure 1: Conceptual tape layout of the stack and adjoint vector for the program $v = x_1 \cdot x_2 \cdot x_3$; $y = (x_2 + x_3) \cdot v$ [7].

dependencies of the outputs into account, by enforcing the barrier during the *reverse* interpretation phase. This helps to avoid false positives, where the desired cost functional does not actually depend on the quantity on the left hand side of the assignment. Thus, no adjoints will ever be propagated along the offending edges, producing a false positive in forward barrier mode.

The forward and reverse barriers are especially useful to debug issues with the propagation of adjoints, when the correct recomputation of primals has already been established. On the other hand, the overwrite barrier can be used when the primals of the recomputed state do not match the expected values. In order to not negatively influence the efficiency of the AD tool, the debugging capability has been implemented in a separate adjoint data type. The introduction of AD types into the OpenFOAM code base has already been discussed in [5] and in detail in [7]. Note, that to enforce the barriers no actual calculation and propagation of adjoints has to take place, only dependency information is needed. Thus, this functionality is removed for the debugging type, significantly lowering the memory footprint of this type.

In addition to the mentioned barriers, another check is implemented in `dco/c++`, which prevents edges pointing to positions further in the tape. During normal operation such edges should never exist and in the context of checkpointing are an indication that states from a previous iteration have not correctly been identified and checkpointed.

6 Additional Considerations for Shape Optimization

Conceptually, the application of checkpointing remains unchanged from the case of topology optimization [5]. Compared to topology optimization, the active path through the pre-processor stage is much more complex. During mesh construction the parameters, that is the location of the individual points of the mesh (contained in the OpenFOAM primitive mesh), are used at various locations in the code to construct the CFD mesh representation. This mesh construction phase is only executed once and can not be restored from a checkpoint easily, therefore it is permanently included in the tape. Following the pre-processing phase, the tape is switched off and the usual checkpointed iteration phase begins. After all iteration steps have been adjoined, the remaining tape of the pre-processor is adjoined, yielding the adjoints of the parameters.

A naive implementation yields results that are not consistent with black-box adjoints, indicating that some dependencies are missed. Those missing dependencies have been first identified as the non-orthogonal correction vectors by manually comparing the tapes of black-box and checkpointed adjoint [7]. With the newly introduced debugging facilities the issues can be easily identified using the forward or reverse barrier technique. The reason the dependencies are missed is the presence of on demand functions in OpenFOAM. Several data fields in the mesh object are stored in dynamic memory, and are only constructed once they are first requested by their access routine.

The following access functions in the `fvMesh` class create their fields on demand:

- `C()`: Constructs the cell center vector,
- `Cf()`: Constructs the face center vector,
- `V()`: Constructs the cell volume vector,
- `Sf()`: Constructs the face area vectors,
- `magSf()`: Constructs the magnitude of face area vectors,
- `deltaCoeffs()`: Constructs delta coefficients,
- `nonOrthDeltaCoeffs()`: Constructs the non orthogonal delta coefficients,

- `nonOrthCorrectionVectors()`: Constructs the non orthogonal correction vectors.

Most of these functions are first accessed during the pre-processor phase, and thus the construction of the fields is captured by the tape. However, the non-orthogonal correction vectors are first constructed when discretizing the gradient operator in the momentum equations, using the corrected surface-normal gradient scheme. The first occurrence of this discretization is in the first SIMPLE iteration, at which point the tape has already been switched off by the checkpointing procedure, to advance the state in passive mode to the first active section. When the `nonOrthCorrectionVectors()` access function is subsequently called while the tape is active, only a reference to the field created earlier is returned. Therefore the dependence of the correction vectors on the parameters is lost.

To fix this problem, we explicitly call all on demand generator functions of the `fvMesh` instance, after the pre-processing is finished but before the tape is switched off. This might be redundant for some functions, if the field has already been initialized. However, as in that case only a reference is returned, which is subsequently ignored, the run time and memory cost of those additional calls is negligible. The actual constructors generating the data are private to the `fvMesh` class, and would require modifications inside the OpenFOAM code base in order to be accessible from our solvers. Therefore we simply trigger dummy calls to the accessor routines, which have the side effect of creating the required data fields. The changes required in order to obtain a consistent checkpointed shape adjoint are presented in Listing 1.

An example for the calculation of shape adjoints using checkpointing is presented in the following section. The same fixes apply when using the checkpointing interface to implement reverse accumulation or piggybacking.

Listing 1: Forcing the early on demand construction of the `fvMesh` fields by calling their access routines.

```
void init_mesh(Foam::fvMesh& mesh){
  mesh.Sf();     mesh.magSf();
  mesh.C();      mesh.deltaCoeffs();
  mesh.Cf();     mesh.nonOrthDeltaCoeffs();
  mesh.V();      mesh.nonOrthCorrectionVectors();
}

int main(int argc, char *argv[])
{
  #include "createTime.H"
  #include "createMeshes.H"
  #include "createFields.H"

  for(fvMesh& solidMesh : solidRegions)
  {
    init_mesh(solidMesh);
  }
  forAll(fvMesh& fluidMesh : fluidRegions)
  {
    init_mesh(fluidMesh);
  }

  ADmode::global_tape->switch_to_passive();
  [...] // Continue w. checkpointed CHT algorithm
}
```

7 CHT Sensitivity Results

Figures 2-4 show preliminary results for the calculation of heat transfer between a heat sink with seven fins at an draft angle of approximately 1.7 degree. Both domains are meshed with a `blockmesh` with conforming interface. The solid domain consists of 129 360 hex cells, the fluid domain of 321 552 cells.

The bottom patch of the solid domain (with material properties of copper) is held at a constant temperature of 375 K. The fluid (with material properties of glycol) enters the domain with a constant velocity of 0.05 m/s and temperature of 300 K. All exterior walls are assumed to be adiabatic. The heat transfer between solid and fluid domain is modeled with OpenFOAMs `turbulentTemperatureCoupledBaffleMixed` model

The flow fields and the temperature on the solid is initialized by running 400 (passive) iteration steps of `chtMultiRegionSimpleFoam`. At this point the simulation has mostly converged. We then run 20 additional iteration steps of our `adjointChtMultiRegionCheckpointingSimpleFoam` solver to obtain sensitivities.

Taping one iteration step of the shape sensitivity problem (using efficient symbolic differentiation of linear solvers) takes roughly 52 GB of tape memory, while one step of the temperature sensitivity problem takes only 23 GB. The higher demand of the shape sensitivity is due to the additional complexity caused by the differentiation of the OpenFOAM mesh representation.

Figure 2 shows the temperature distribution on the surface of the solid, as well as a slice through the fluid domain, showing the velocity distribution. As the cost function we choose the average temperature on the outlet patch, as calculated by `gAverage(T)`. The dependence of the average temperature on the temperature distribution at the heated wall is depicted in Figure 3. Red regions therefore indicate where the heat energy is best transported from the bottom plate to the fluid.

For the same case, Figure 4 shows the shape sensitivities of the average temperature at the outlet w.r.t. movement of the surface mesh points in surface normal direction. Red regions indicate where the cross section of the fins should shrink, making them narrower, blue regions (with negative sign) indicate where the cross section should be expanded.

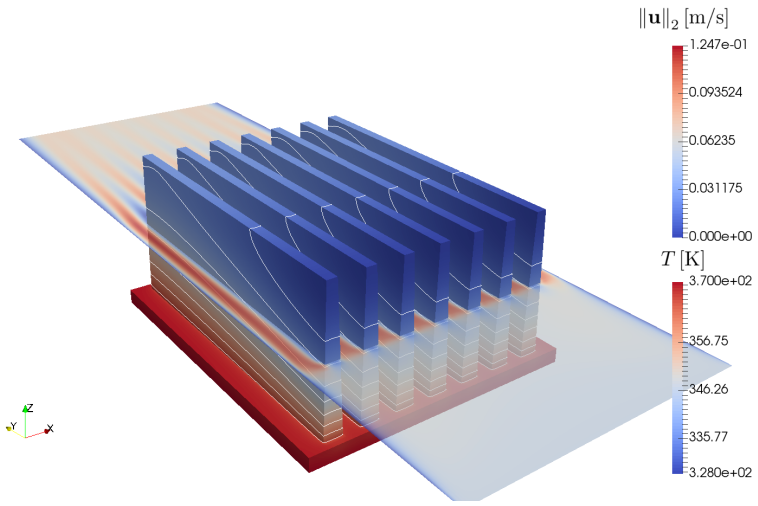

Figure 2: Temperature distribution T on the solid surface with temperature isolines (white). Velocity magnitude distribution in the fluid domain on a z-Normal slice.

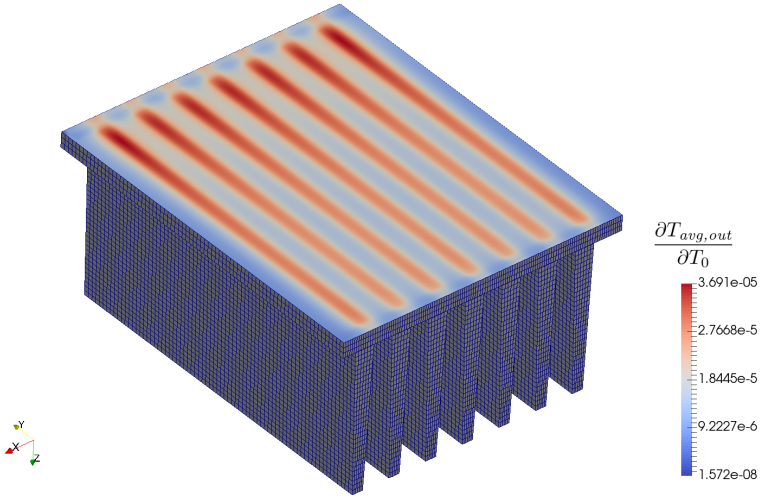

Figure 3: Sensitivity of the average outlet temperature w.r.t. the temperature on the heated bottom wall of the solid domain.

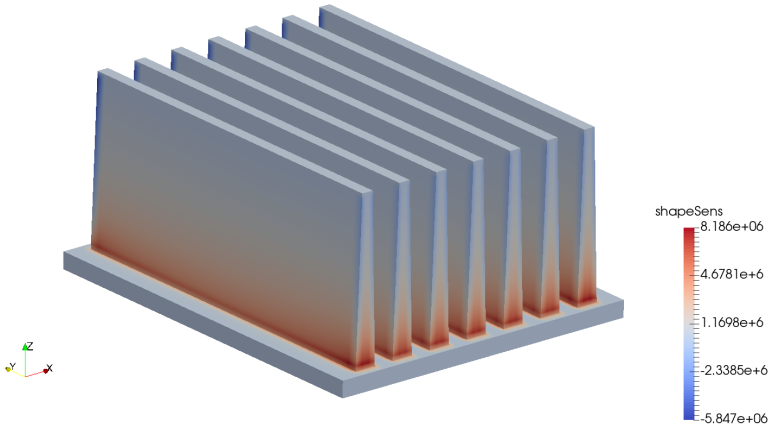

Figure 4: Surface sensitivity on the solid to fluid interface.

8 Outlook

We demonstrated the applicability of our existing discrete adjoint framework for OpenFOAM to complex CHT cases. Emphasis was placed on the correct checkpointing treatment of the states in the solid and fluid domains. While, utilizing the presented debugging tools, the manual treatment of the conflicting boundary conditions is possible, it is desirable to obtain a more robust implementation that relies on the already existing primal copy constructors of OpenFOAM. Such an implementation is currently in development and will potentially be presented at the workshop. As part of our ongoin research, we plan to apply our discrete adjoint CHT framework to a variety of different heat sink geometries.

Acknowledgment

This work is partially supported by the BMWi ZIM project *Entwicklung optimierter Kühlgeometrien mittels adjungierter Simulationsmethoden für die Direkt-Heißwasserkühlung von Rechenzentren* (Developmtent of optimized cooling geometries for hot watercooling of date centers using adjoint simulation methods).

References

[1] Dorfman, A. S. *Conjugate problems in convective heat transfer*. CRC Press, (2009).

[2] Zeinalpour, M., Mazaheri, K., and Kiani, K. A coupled adjoint formulation for non-cooled and internally cooled turbine blade optimization. *Applied Thermal Engineering* **105**, 327 – 335 (2016).

[3] Kontoleontos, E. A., Papoutsis-Kiachagias, E. M., Zymaris, A. S., Papadimitriou, D. I., and Giannakoglou, K. C. Adjoint-based constrained topology optimization for viscous flows, including heat transfer. *Engineering Optimization* **45**(8), 941–961 (2013).

[4] Burghardt, O., Gauger, N. R., and Economon, T. D. Coupled adjoints for conjugate heat transfer in variable density incompressible flows. In *AIAA Aviation 2019 Forum*, 3668, (2019).

[5] Towara, M. and Naumann, U. A discrete adjoint model for OpenFOAM. *Procedia Computer Science* **18**(0), 429 – 438 (2013). 2013 International Conference on Computational Science.

[6] Towara, M., Schanen, M., and Naumann, U. MPI-parallel discrete adjoint OpenFOAM. *Procedia Computer Science* **51**, 19 – 28 (2015). 2015 International Conference on Computational Science.

[7] Towara, M. *Discrete Adjoint Optimization with OpenFOAM*. Dissertation, RWTH Aachen University, (2019).

[8] Patankar, S. V. and Spalding, D. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *Int. J. of Heat and Mass Transfer* **15**(10), 1787–1806 (1972).

[9] Moukalled, F., Mangani, L., Darwish, M., et al. *The Finite Volume Method in Computational Fluid Dynamics*. Springer, (2016).

[10] Naumann, U. DAG reversal is NP-complete. *Journal of Discrete Algorithms* **7**, 402–410 (2009).

[11] Leppkes, K., Lotz, J., and Naumann, U. Derivative Code by Overloading in C++ (`dco/c++`): Introduction and Summary of Features. Technical Report AIB-2016-08, RWTH Aachen University, September (2016).

[12] Griewank, A. and Walther, A. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, (2008).

[13] Griewank, A. and Walther, A. Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software* **26**(1) March (2000).

[14] Christianson, B. Reverse accumulation and attractive fixed points. *Optimization Methods and Software* **3**(4), 311–326 (1994).

[15] Lotz, J. *Hybrid Approaches to Adjoint Code Generation with dco/c++*. Dissertation, RWTH Aachen University, (2016).