11th World Congress on Computational Mechanics (WCCM XI)
5th European Conference on Computational Mechanics (ECCM V)
6th European Conference on Computational Fluid Dynamics (ECFD VI)
E. Oñate, J. Oliver and A. Huerta (Eds)

# ADJOINTS OF FIXED-POINT ITERATIONS

**A.TAFTAF*,V.PASCUAL* AND L. HASCOËT***

* INRIA, Sophia-Antipolis, France, {Elaa.Teftef, Valerie.Pascual, Laurent.Hascoet}@inria.fr

**Key words:** Automatic Differentiation, Adjoint, Fixed-Point algorithms

**Abstract.** Adjoint algorithms, and in particular those obtained through the adjoint mode of Automatic Differentiation (AD), are probably the most efficient way to obtain the gradient of a numerical simulation. This however needs to use the flow of data of the original simulation in reverse order, at a cost that increases with the length of the simulation. AD research looks for strategies to reduce this cost, taking advantage of the structure of the given program. One such frequent structure is fixed-point iterations, which occur e.g. in steady-state simulations, but not only. It is common wisdom that the first iterations of a fixed-point search operate on a meaningless state vector, and that reversing the corresponding data-flow may be suboptimal. An adapted adjoint strategy for this iterative process should consider only the last or the few last iterations. At least two authors, B. Christianson and A. Griewank, have studied mathematically fixed-point iterations with the goal of defining an efficient adjoint. In this paper, we describe and contrast these two strategies with the objective of implementing the best suited one into the AD tool that we are developing. We select a representative application to test the chosen strategy, to propose a set of user directives to trigger it, and to discuss the implementation implications in our tool.

## 1 INTRODUCTION

The adjoint mode of Automatic Differentiation (AD) is widely used in science and engineering : assuming that the simulation has a scalar output (objective function), the adjoint algorithm can return its gradient at a cost independent of the number of inputs. The key is that adjoints propagate partial gradients backwards from the result of the simulation.

The main difficulty of adjoint AD lies in the management of intermediate values. The computation of the partial gradients involves the partial derivatives of each run-time elementary computation of the original simulation. As these partial derivatives are needed in reverse execution order, and they use values from the original computation, strategies must be designed to retrieve the original values in reverse order. For instance,

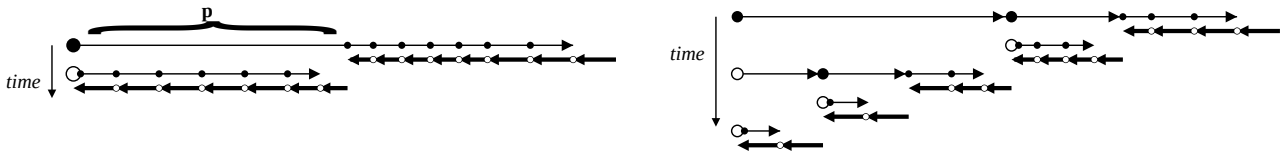


**Figure 1**: left : single check point P, right : nested checkpoints

- the Recompute-All approach recomputes the intermediate values whenever needed, by restarting the program from the stored initial state,
- the Store-All approach stores the original intermediate values into a stack or at least those that will be needed, during a preliminary execution of the original program known as the forward sweep (FW). Then follows the so-called backward sweep (BW) which computes the partial derivatives using these stored values. The needed `push` and `pop` primitives are provided by a separate library.

In the rest of this paper, we assume a Store-All approach. Extension to the Recompute-All approach is possible, but is beyond the scope of this paper. On large real applications both RA and SA approaches turn to be impracticable due to their cost in time or memory space respectively. The answer to this problem is called checkpointing: for instance in the SA setting, checkpointing consists in selecting some part of the program, and in *not* storing its intermediate values, but rather storing the minimum amount of data to run this part again later ("a snapshot"). After taking the snapshot, P is run with no storage of intermediate values. Then P is run again when it is time to propagate derivatives backwards through P.

The result of checkpointing part P is shown on the left of figure 1. The snapshot (big black dot) memorizes whatever is needed to run P again. The peak stack size is reduced, at a small additional cost which is the duplicate execution of P. The right part of the figure shows the execution of the program when there are many nested checkpoints. The peak stack size is reduced again at the cost of some snapshots and some duplicates executions. In our AD tool, we use the SA approach and by default checkpointing is applied at each subroutine call. To summarize, the structure of an adjoint code is a FW sweep followed by a BW sweep, and recursively for each checkpoint P a copy of P (with no `push`) in the enclosing FW sweep and the sequence of FW and BW of P in the enclosing BW sweep. This strategy to orchestrate reversal of the data-flow is general, and as such are unable to take advantage of algorithmic knowledge of the specific simulation. Exploiting knowledge of the algorithm and of the structure of the given simulation code can yield a huge performance improvement in the adjoint code. In our tool, special strategies are already available for parallel loops, long unsteady iterative loops, linear solvers... We focus here on the case of fixed-point loops, i.e. loops that iteratively refine a value until it becomes stationary.

As Fixed-Point algorithms often start from a "random" guess initial state, one intu-

ition is that at least the first iterations are in a sense meaningless and should not be "remembered" for the adjoint computation. Furthermore, there is a discrete component of an iterative algorithm, namely the number of iterations, and this makes differentiability questionable : a small change of the inputs may add or remove one iteration of the loop, which is a discontinuous change. For these reasons we are looking for a specific strategy for the adjoint that reverses only the necessary data-flow, and that restores confidence in the validity of the derivative.

At least two authors have studied mathematically fixed-point iterations with the goal of defining an efficient adjoint. Griewank's "Delayed Piggyback"(AG) [3, 4] ultimately targets computation of the adjoint derivatives together with the tangent derivatives in the same order. Both will be the ingredients of a so-called "reduced approximation estimate" that exhibits improved convergence properties. Christianson's "Two Phases" (BC) method [1, 2] focuses exclusively on building adjoints. His method is applicable to arbitrary fixed-point loops.

In the following sections, we will see in more detail the two methods, their common points and differences. We will explain our choice and we will demonstrate on a representative example the benefits of this strategy.

## 2 STRATEGIES FOR FIXED-POINT ADJOINTS

Consider a piece of code that solves a fixed-point equation, i.e. finds $z_*$ such that

$$z_* = \phi(z_*, x)$$

where $x$ is some fixed parameter and therefore $z_*$ is a function $z_*(x)$ of $x$. This code initializes $z$ with some initial guess $z_0$, then iteratively calls $z_{k+1} = \phi(z_k, x)$ until meeting some stopping criterion that express stationarity of $z$. Finally $z_*$ is used by the sequel code to compute some final result $y = f(z_*, x)$.

### 2.1 Standard adjoint strategy

The standard adjoint, sketched on the right of figure 2, propagates the partial gradients in the reverse order across each iteration, reusing the intermediate $z_k$ in reverse order. To this end, the successive $z_k$ must be stored as they are computed during the FW sweep. This also implies that the number of iterations in the BW sweep is the same as in the FW sweep. In other words, whereas the forward loop is a fixed-point loop, the backward loop is not, as it does not test for stationarity of $\overline{z}$. Notations: $\phi_z$ and $\phi_x$ respectively represent $\frac{\partial}{\partial z}\phi$ and $\frac{\partial}{\partial x}\phi$.

### 2.2 Griewank's "delayed piggyback" method

Griewank's method (AG) observes first that the adjoint algorithm must wait for the original fixed-point to be "sufficiently" converged before starting derivative computations. More importantly, the method aims at computing the adjoint iterations in the same order

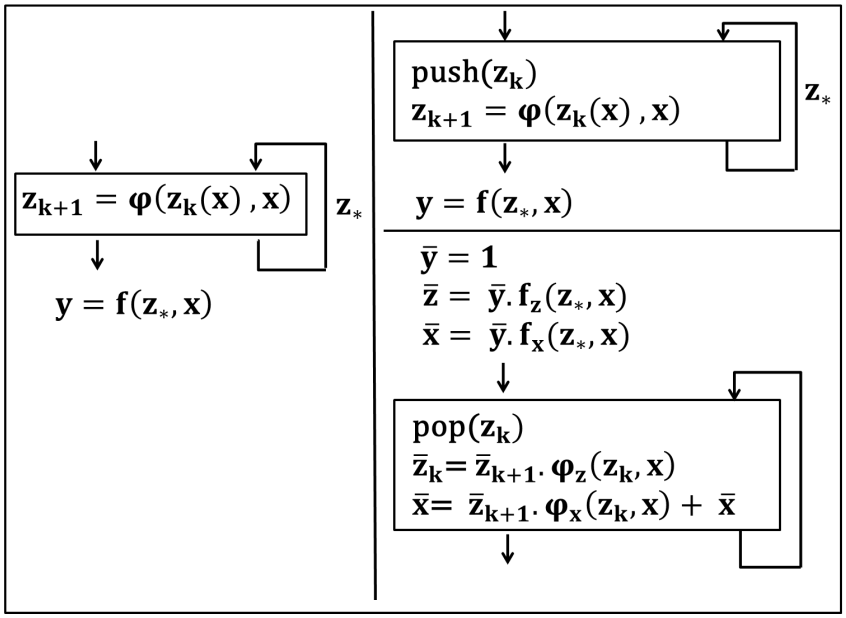


**Figure 2**: left : example of code containing a fixed-point loop, right: standard adjoint strategy applied to this code

as the original. This unconventional adjoint structure comes from the targeted application, which is to compute a so-called "reduced approximation estimate" that involves both adjoint and tangent derivatives. This has also the consequence that the adjoint of the downstream computation $f$ must be repeated inside the adjoint fixed-point loop. As a last restriction, the method assumes that the fixed-point function $\phi$ is in fact split into computing some $w = F(z, x)$ followed by updating $z$ with $w$ times some preconditioning matrix $P$. The method is sketched in figure 3 : once the original fixed-point loop is sufficiently converged (not shown in figure), the remaining iterations are augmented with derivative computations, and there is no need for a backward sweep. These iterations continue until some stopping criterion is met, which combines stationarity of the original $z$ and of the adjoint $\overline{w}$. The adjoint of the fixed-point computation terminates by computing the required $\overline{z}$ using the final $z_*$, $\overline{w_*}$, and the adjoint derivatives of $F$ and $f$ with respect to $x$.

### 2.3 Christianson's "two-phases" method

Christianson's method (BC) observes that only the converged value $z_*$ and the converged intermediate values occurring in $z_* = \phi(z_*, x)$ must be used in the derivative computation. All intermediate values of previous iterations are inexact and should not be used. Furthermore, the computation of the derivative of $\phi$ with respect to $x$ needs

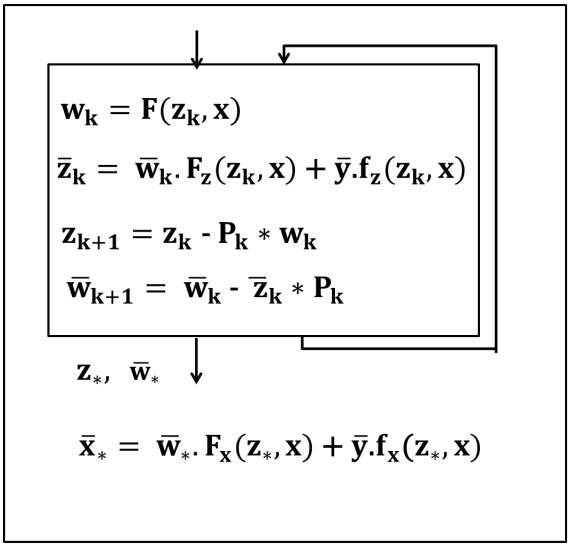


**Figure 3**: Delayed piggyback method (AG)

to be computed only once and does not need to be included in the fixed-point iteration that solves for $\overline{w}$. This results in the algorithm sketched in figure 4 : globally this algorithm keeps the standard structure of adjoint codes for everything before and after the fixed-point loop. It also keeps the structure of the forward sweep of the fixed-point loop, except that only the last iteration's intermediate values are stored. On the backward sweep however the method introduces a new variable $\overline{w}$ of the same shape as $z$, which must be computed as the solution $\overline{w_*}$ of a new fixed-point equation:

$$\overline{w} = \overline{w}.\frac{\partial}{\partial z}\phi(z_*, x) + \overline{z}$$

where $\overline{z}$ results from the adjoint of the downstream computation $f$. The adjoint of the fixed-point computation terminates by computing the required $\overline{x}$, using $\overline{w_*}$ and the adjoint derivatives of $\phi$ with respect to $x$. The above adjoint derivative computations repeatedly use the intermediate values stored by the last forward iteration, which are $z_*$ plus whatever was used to compute it during the last iteration.

## 3 Selecting the method to be implemented

Both BC and AG methods yield an adjoint convergence rate similar to original fixed-point loop. Derivatives convergence may lag behind by a few iterations, but will eventually converge at the same rate. Both methods achieve to differentiate only the last or the few last iterations i.e. those who operate on physically meaningful values. Both manage also to avoid naïve inversion of the original sequence of iterations, therefore saving the cost

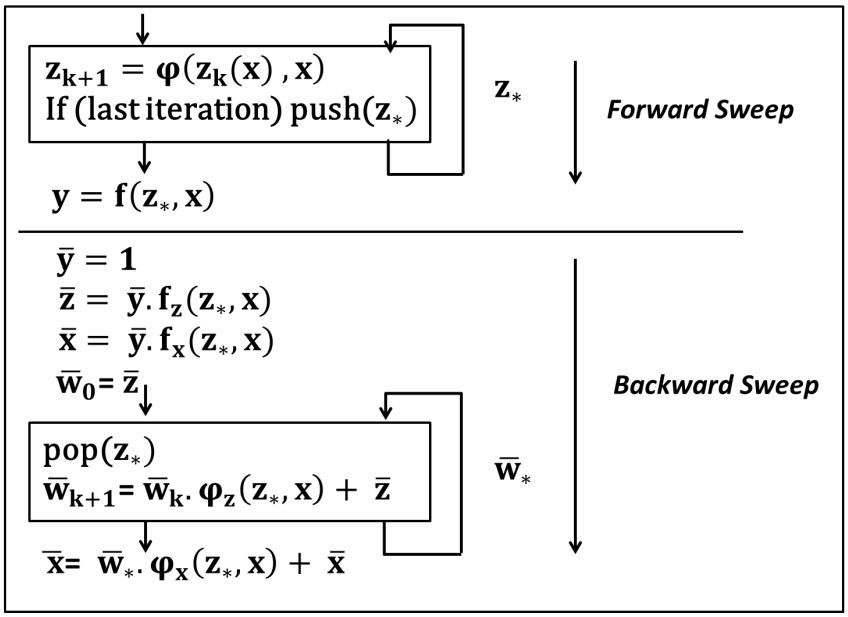


**Figure 4**: Two-phases method (BC)

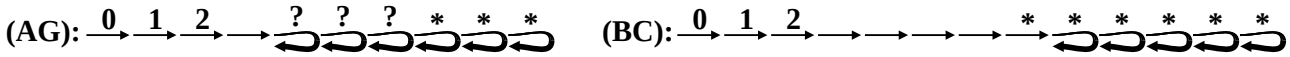


**Figure 5**: comparison of executing adjoint code with each strategy

of data-flow reversal. Consequently the adjoint, which is itself a fixed-point, must have a distinct, specific stopping criterion.

Because of its setting, AG method makes some additional assumptions on the shape of the iteration step and on the structure of the surrounding program whereas BC remains general. Another difference, visible on figure 5, is that BC starts adjoining the iteration step, actually the last one, only when the original iteration has converged "fully" (indicated by *), whereas AG triggers the adjoint iterations earlier, together with the remaining original ones, when those are converged only "sufficiently" (indicated by ?). This may be hard to determine automatically. Since AG adjoint computation starts with slightly approximate values, it may require a few more iterations than BC. A last difference is that AG requires adjoining the sequel of the program i.e. the part $f$ after the fixed-point iteration, repeatedly inside the adjoint iteration step. This is fine in the chosen setting where the sequel is assumed short, but it has a significant cost in general when the sequel is complex or when fixed-point loops are nested.

Both methods provide some correctness proofs and our choice is mostly based on im-

plementation reasons. We want to implement in our AD tool the strategy that covers more cases, so we prefer not to have assumptions in the iteration shape. We are also worried by the cost of the sequel $f_z$ in AG. We also consider that BC has the advantage of preserving the two-sweep structure. For these reasons, we currently select Christianson's strategy.

## 4 USER DIRECTIVES

One question is how to trigger the special-purpose strategy for fixed-point loops. Obviously, it is very hard to detect automatically every instance of a fixed-point loop computation in a given code. It is even impossible in general because of undecidability of general static data-flow analysis. Therefore we must rely on the end-user to provide this information, for instance by means of the following two pairs of directives :

- One pair named (`$AD_START_FP_LOOP`, `$AD_END_FP_LOOP`) designates the starting and ending of the fixed-point loop. The deliminated code fragment contains in particular the stopping test. These directives must also specify the fixed-point variables $z$ and the stopping criterion to be used in the adjoint loop.

- Another pair named (`$AD_START_FP_ITERATION`, `$AD_END_FP_ITERATION`), designates the body of the fixed-point loop, i.e. the piece of code that implements $\phi$.

The data-flow analysis already present in our tool help us to identify the input and output variables of the fixed-point loop body, so that the user-given specification of $z$ is enough to determine $x$.

## 5 EXTENSION OF STACK MECHANISM

We mentioned in section 2.3 that the intermediate values are stored only during the last forward iteration. Then they are repeatedly used in each of the backward iterations. Our standard stack mechanism does not support this behavior. We need to define an extension to specify that some zone in the stack (a "repeated access zone") will be read repeatedly. Our choice is to add three new primitives to our stack, supposed to be called in the middle of a sequence of stack `pop`'s.

- `start_repeat_stack()` states that the current stack position is the top of a repeated access zone.

- `reset_repeat_stack()` states that the stack pointer must return to the top of the repeated access zone.

- `end_repeat_stack()` states that there will be no other read of the repeated access zone.

In the adjoint generated code, these procedures must be called :

- `start_repeat_stack()` at the start of the adjoint backward fixed-point loop.
- `reset_repeat_stack()` before each call to $\phi_z$ and before the call to $\phi_x$.
- `end_repeat_stack()` at the end of the adjoint backward fixed-point loop.

However this set of primitives doesn't handle the case of checkpointing occurring inside the adjoint iterations. Checkpointing implies that the stack may grow again (with `push`'es) and the danger is to overwrite the contents of the repeated access zone. Our solution to keep this zone safe is to store the new values at the real top of the stack, i.e. above the repeated access zone. This requires two additional primitives.

- `freeze_repeat_stack()` saves the current stack pointer and says that all coming `push`'es must go above the top of the current repeated access zone.
- `unfreeze_repeat_stack()` states that previous `pop`'s have returned the stack pointer to the top of the current repeated access zone, and therefore resets the stack pointer to its saved location so that next `pop`'s will read in the repeated access zone.

This is illustrated by figure 6. Notice that `unfreeze_repeat_stack()` is in principle unnecessary, since every `pop` could check if the stack pointer is at the top of a repeated access zone and react accordingly. However this would slow down each call to `pop`, which are frequent. On the other hand, `unfreeze_repeat_stack` may be called only once, at a location that can be statically determined by the AD tool. Therefore, in the adjoint generated code, we will call :

- `freeze_repeat_stack()` before each checkpointed adjoint subroutine call or code fragment during the adjoint backward iteration.
- `unfreeze_repeat_stack()` after the corresponding adjoint subroutine call or code fragment.

This also allows us to handle nested fixed-point loops, as demonstrated in the following experiment.

## 6 EXPERIMENTAL RESULTS

Before actually implementing the fixed-point strategy in our AD tool, we have chosen to validate it on a representative example (sketched in figure 7), applying the strategy by hand but as mechanically as possible. The representative example must be complex enough to capture the following features of real codes : non-trivial calculations whose adjoint need many values stored on the stack, fixed-point loops with possibly non-quadratic convergence rates, and possibly nested. We chose an algorithm that solves for $u$ in an equation similar to a heat equation, with the form :

$$-\Delta u + u^3 = F \tag{1}$$

with $F$ given. The solving algorithm uses two nested fixed-point resolutions.

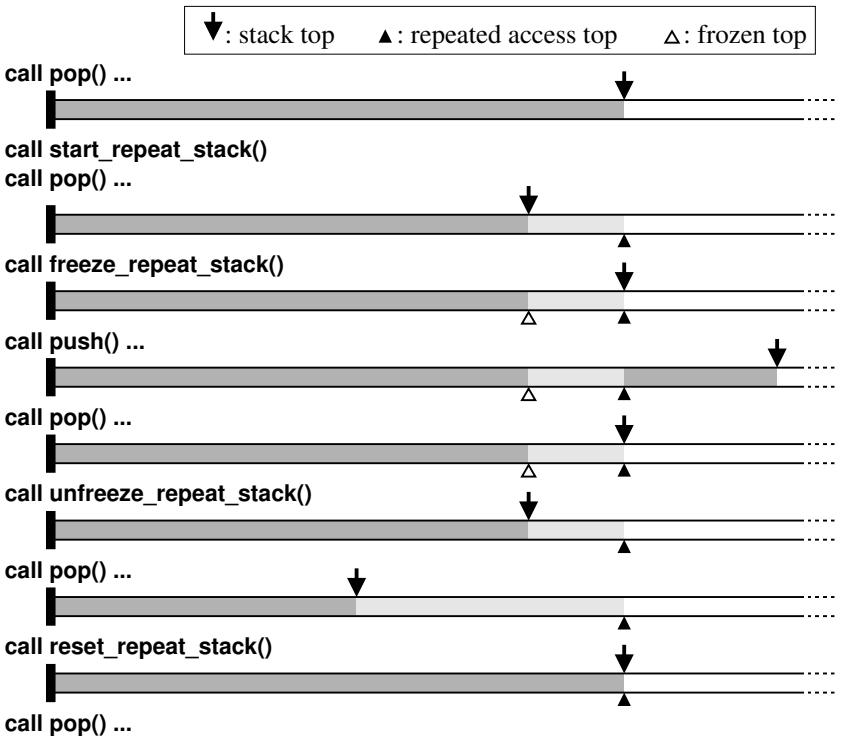


**Figure 6**: extension of stack mechanism to manage a repeated access zone

- On the outside is a (pseudo)-time integration, considering that $u$ evolves with time from an arbitrary $u(0)$ towards the stationary solution $u(\infty)$, following the equivalent equation

$$\frac{u(t+1) - u(t)}{\Delta t} - \Delta u(t+1) + u^3(t) = F \tag{2}$$

- On the inside is the resolution of the implicit equation for $u(t+1)$ as a function of $u(t)$ and $F$ with a Jacobi iterative method, which is in turn a fixed-point algorithm.

We have differentiated the complete algorithm using our AD tool that applies the standard adjoint algorithm described in section 2.1. Then we have manually modified the adjoint code to apply BC strategy on both fixed-point loops. We arbitrary set the stopping criterion of the adjoint fixed-point loop so the stationarity of $\overline{w}$ is up to the same level of accuracy as the original value. We compare performance of the modified code with the standard adjoint.

Performance comparison is made difficult by the fact that the two algorithms do not produce the same result: only the BC approach has a stopping criterion that takes into account actual stationarity of the adjoint. As a result, the BC approach iterates slightly fewer times than the standard:

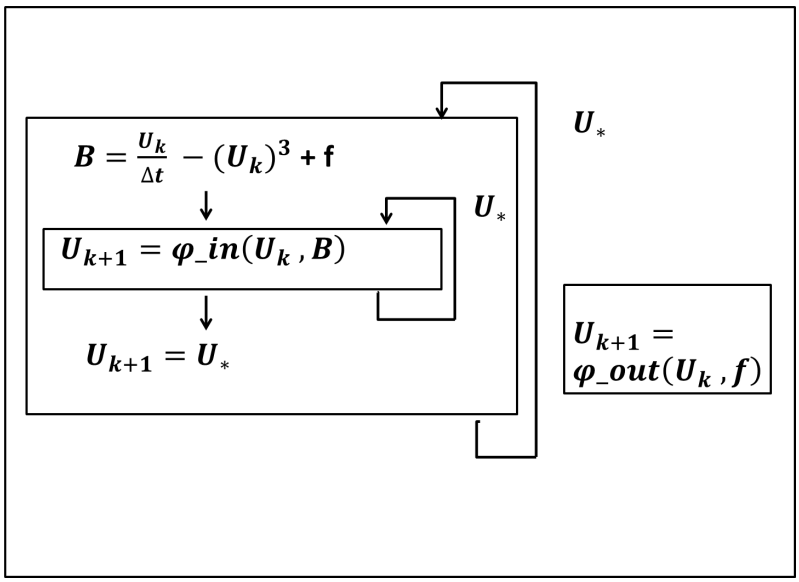


**Figure 7**: representative example containing two nested Fixed-Point loops : outside loop $\phi\_out()$ and inside loop $\phi\_in()$

- number of iterations in the outside fixed-point loop is 289 instead of 337.
- number of iterations in the inside fixed-point loops is 34 instead of an average of 44.

However the result is less accurate than with the standard AD tool, although inside the prescribed accuracy of $10^{-15}$. Accuracy is estimated by comparison to a result obtained with a much smaller stationarity criterion ($10^{-40}$). We then choose to take an alternate viewpoint, forcing the BC approach to iterate as much as the standard approach and examining the accuracy of the result. The result of the standard adjoint deviates from the reference result by $2.1 * 10^{-5}$ %. The result of BC adjoint deviates by $1.1 * 10^{-5}$ %. This a small improvement, due to the fact that BC adjoint is computed using only the fully converged values. Notice however that the principal benefit of the BC method is not about accuracy nor run time but about reduction of the memory consumption, since the intermediate values are stored only during the last forward iteration. The peak stack space used by the standard adjoint is about 86 Kbytes, whereas the BC adjoint uses only a peak stack size of 268 bytes. These measurements are coherent with our understanding of the BC adjoint method.

## 7 CONCLUSION

We are seeking to improve performance of adjoint codes produced by the adjoint mode of Automatic Differentiation, by taking advantage of common structures occurring in scientific codes. We considered here the frequent case of fixed-point loops, for which several

authors have proposed adapted adjoint strategies. We explained why we consider the strategy initially proposed by Christianson as the best suited for our needs and implementation context. We experimented this strategy on a representative code and quantified its benefits, which are marginal in terms of runtime, and significant in terms of memory consumption. Implementation of this strategy inside our AD tool is under way, and we discussed some of the implementation implications.

Theoretical numerical analysis papers discuss the question of the best stopping criterion for the adjoint fixed point loop. However these criteria seem far too theoretical for an automated implementation. We believe that the initial implementation will have to ask the end user to specify this adjoint criterion, but hope remains to derive it mechanically from the original loop's stopping criterion, perhaps using software analysis rather than numerical analysis.

An interesting further research is the question of the initial guess. A fixed point algorithm starts from a "random" initial guess, which must have no influence on the final converged value, but may vastly affect the number of iterations needed to attain convergence. In fact a very good initial guess, that leads to the converged value in few iterations, may be adverse to adjoint convergence in the general setting where the adjoint loop mimics the number of iterations of the original loop. This remark is at the source of several theoretical papers on the limitations of adjoint AD. Christianson's strategy solves this problem since the adjoint iterations are unrelated to the original iterations. Going further, we feel that this strategy could be improved by looking, when possible, for a better initial guess for the adjoint fixed point loop.

## ACKNOWLEDGEMENT

This research is supported by the project "About Flow", funded by the European Commission under FP7-PEOPLE-2012-ITN-317006. See "http://aboutflow.sems.qmul.ac.uk".

## REFERENCES

[1] B. Christianson. Reverse accumulation and attractive fixed points. *Optimization Methods and Software*, 3:311–326, 1994.

[2] B. Christianson. Reverse accumulation and implicit functions. *Optimization Methods and Software*, 9(4):307–322, 1998.

[3] A. Griewank and C. Faure. Piggyback differentiation and optimization. In Biegler et al., editor, *Large-scale PDE-constrained optimization*, pages 148–164. Springer, LNCSE #30, 2003.

[4] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Other Titles in Applied Mathematics, #105. SIAM, 2008.