Combinatorial Problems in OpenAD

Jean Utke[1] and Uwe Naumann[2]

[1] Argonne National Laboratory, MCS
9700 S. Cass Ave., Argonne, IL 60439, USA
utke@mcs.anl.gov
[2] RWTH Aachen University, LuFG Informatik 12
52056 Aachen, Germany
naumann@stce.rwth-aachen.de

Abstract. Computing derivatives using automatic differentiation methods entails a variety of combinatorial problems. The OpenAD tool implements automatic differentiation as source transformation of a program that represents a numerical model. We select three combinatorial problems and discuss the solutions implemented in OpenAD. Our intention is to explain the specific parts of the implementation so that readers can easily use OpenAD to investigate and develop their own solutions to these problems.

Keywords. automatic differentiation, combinatorial problems, OpenAD

1 Introduction

Computing derivatives of numerical models $\boldsymbol{f}(\boldsymbol{x}) \mapsto \boldsymbol{y} : \mathbb{R}^n \mapsto \mathbb{R}^m$ given as a computer program P is an important but also compute-intensive task. Automatic differentiation (AD) [1] provides the means to obtain such derivatives. OpenAD [2] implements AD as a source transformation applied to Fortran programs for both the forward and reverse modes. In this paper we describe the solutions to three combinatorial aspects of AD as implemented in OpenAD. For information regarding the general use of OpenAD please see [3]. Because OpenAD is a source transformation tool, the combinatorial problems are solved at compile time; hence, our approach **can afford costly heuristics to approximate a good solution**. Using such heuristics even with small improvements is justified when, for instance, these improvements benefit the numerical kernel of a loop and therefore the improvement accumulates over the runtime, realizing considerable savings. Our intention is to explain the representation of the combinatorial problems in OpenAD such that the reader will find it easy to implement and investigate other solutions to these problems. The remainder of this section provides a brief overview of the concepts of AD that are relevant to the problems in question. General information on AD can be found in recent conference proceedings [4,5], the AD community website [6], and the survey paper in this proceedings [7].

Automatic differentiation relies on the chain rule applied to a sequence of elemental operations. These operations are the function calls and intrinsics built into the given programming language, such as sin, cos, or the operators + and `*`. Typically, right-hand-side expressions of assignment statements form single-expression-use graphs [8], and

Dagstuhl Seminar Proceedings 09061
Combinatorial Scientific Computing
http://drops.dagstuhl.de/opus/volltexte/2009/2095

under certain conditions a sequence of single-expression-use graphs corresponding to a sequence of assignment statements in a given program can be *flattened* into a directed acyclic graph [9]. This graph is called the *computational graph* G. An example is shown in Fig. 1. The nonminimal vertices of G represent values computed as a result of a sin-

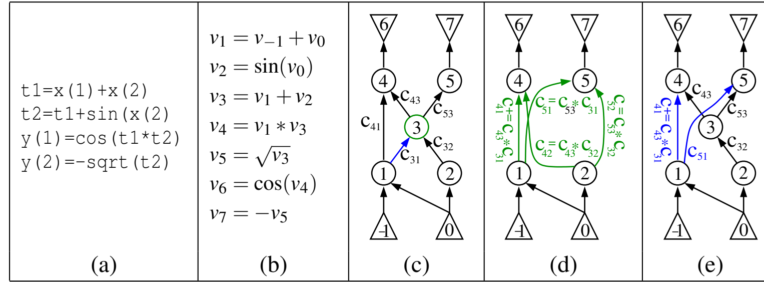

Fig. 1. Sequence of statements (a), corresponding sequence of elemental operations enumerating the computed values (b), the corresponding G using the value enumeration (c), eliminaton of vertex 3 from G (d), and front elimination of edge $(1,3)$ from G (e).

gle intrinsic function or operator call. For each such computed value v_i we can provide the local partial derivative $c_{ij} = \partial v_i / \partial v_j$ with respect to its arguments v_j. For instance, in Fig. 1(b) we have $v_4 = v_1 * v_3$ and therefore $c_{41} = v_3$ and $c_{43} = v_1$. These partial derivatives are given as edge labels in G; see Fig. 1(c). Following Baur's formula [10], we can compute Jacobian entries by multiplying edge labels along paths from minimal to maximal edges in G and adding parallel paths. Because of distributivity of the operations, the edge labels can be combined in many different orders yielding the same result but **differing in the number of multiplication (and addition) operations performed**. The multiplications and additions can be expressed as graph manipulations in G. Figure 1(d) shows a vertex elimination where the number of multiplications is equivalent to the Markowitz degree, and Figure 1(e) shows a front-edge elimination where the number of multiplications is equal to the number of out-edges of the target vertex; the modified elements are shown in color. For details and additional operations see [7,11] and [3], Sec. 3.2. Each elimination step modifies G to some $G^{(k)}$. One can proceed with eliminations until the modified G has reached bipartite form G^b and the edge labels are the entries of the Jacobian $\boldsymbol{J}$ of $\boldsymbol{f}$ evaluated at some point $\boldsymbol{x}_0$. This description illustrates the basis of the following NP-complete [12] problem.

1. What is the minimal number of multiplications to reduce G to bipartite form?

The OpenAD representation[1] of the problem is discussed in Sec. 2.1 followed by example heuristics in Sec. 2.2.

[1] OpenAD is actively being developed. All names of files, classes, methods, and variables mentioned here refer to versions xaifBooster: b88d9f62ae82+ , angel: ff7faed78ea6 of the respective Mercurial source code repositories (see [2]). Changes since then can be traced back by using the Mercurial web interface at http://mercurial.mcs.anl.gov/ad .

In many practical applications one does not require the full $\boldsymbol{J}$ but only projections $\boldsymbol{JS}$ or $\boldsymbol{W}^T\boldsymbol{J}$ with $\boldsymbol{S} \in \mathbb{R}^{n\times p}, \boldsymbol{W} \in \mathbb{R}^{m\times p}$, where p is the number of desired directions. During the eliminations described above, the number of edges (with nonunit/nonconstant labels) at any intermediate stage $G^{(k)}$ often is smaller than that of the final G^b. Consequently, the **projection operations executed with $G^{(k)}$ are cheaper** than with G^b. The general concept is known as *scarcity* [13] and has been investigated for practical use in [14]. Figure 2 shows an example for a rank-1 update. The initial G has only $3n$ edges, of which $2n$ are constant. If we eliminate the intermediate vertex z (see also Fig. 1(d)), we have n^2 nonconstant edges. Consequently, the projections $\boldsymbol{JS}$ or $\boldsymbol{W}^T\boldsymbol{J}$ require only $3np$ operations compared to n^2p. This scenario leads to the second combinatorial problem.

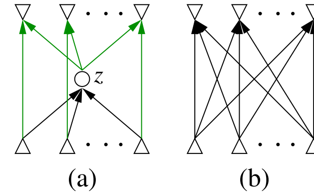

Fig. 2. Computational graph (edges with constant labels are green) for $\boldsymbol{f}(\boldsymbol{x}) = (\boldsymbol{D} + \boldsymbol{a}\boldsymbol{x}^T)\boldsymbol{x}$ with an intermediate variable $z = \boldsymbol{x}^T\boldsymbol{x}$ in (a) and state after eliminating z in (b).

2. What is a minimal representation $\boldsymbol{J}^*$ for $\boldsymbol{J}$?

In Sec. 2.3 we will explain the implementation of a greedy heuristic to approximate the minimal representation, with particular consideration given to the additional savings possible when one considers some edge labels to have unit value.

Because in OpenAD we have to build G at compile time, the limitations of an automatic code analysis[2] to identify computed values (on the left-hand side of an assignment) to their subsequent uses (in later right-hand-side-expressions) also limit the scope of any single G typically to the contents of a basic block. Thus, for an entire program P with control flow we have a sequence of s graphs G_i and, corresponding to that, a sequence of s local Jacobians $\boldsymbol{J}_i$. The overall Jacobian $\boldsymbol{J} \in \mathbb{R}^{m\times n}$ for P is therefore a chained matrix product over the $\boldsymbol{J}_i$. Again, instead of the full Jacobian $\boldsymbol{J}$ we typically require the projections $(\boldsymbol{J}_s \circ \ldots \circ (\boldsymbol{J}_1 \circ \boldsymbol{S})\ldots)$, called the *forward mode*, or $(\ldots(\boldsymbol{W}^T \circ \boldsymbol{J}_s) \circ \ldots \circ \boldsymbol{J}_1)$, called the *reverse mode*. Here we can also use the $\boldsymbol{J}_i^*$ in place of the less efficient $\boldsymbol{J}_i$. The bracketing for the reverse mode indicates that the $\boldsymbol{J}_i$ are required in an order inverse to the execution of the original program P. Typically, because of memory limitations, one will not be able to compute and store *all* the $\boldsymbol{J}_i$ at once to then use them for the reverse sweep. Instead one will have to trade off some recomputation of the $\boldsymbol{J}_i$ from checkpoints. Determining the checkpoints and orchestrating the computation of the $\boldsymbol{J}_i$ and their use in the reverse sweep is called a *reversal scheme*. This leads to the third combinatorial problem, also shown to be NP-complete [15].

3. Which reversal scheme achieves the fewest recomputations, given constraints on the memory available for checkpoints and storing the $\boldsymbol{J}_i$?

Section 3.1 highlights split and joint modes as two simple cases at the respective ends of the spectrum controlled at the level of subroutine calls. Section 3.3 discusses the optimal solution for the special case of uniform iterations.

[2] This stems from *aliasing*, for example the possibility that two array elements `a(i)` and `a(j)` or two pointers `p` and `q` point to the same address.

2 Computational Graphs

The modular design of OpenAD separates different tasks of the source transformation. The core AD transformation engine is a module called `xaifBooster` separate from the modules for parsing, code analysis, and unparsing. An obvious candidate for the separation within the AD transformation engine is the computation of the local Jacobians $\boldsymbol{J}_i$ or $\boldsymbol{J}_i^*$, pertaining to problems 1 and 2. We rely on the object-oriented language features of C++ to facilitate the module separation via interface classes.

2.1 Problem Representation

To experiment with elimination heuristics, we do not need to understand how G is constructed from code or how exactly the elimination steps are translated back into executable code. We need only an interface that describes the structure of G, the elimination steps, and (for problem 2) G^*, or *remainder graph*. The main interface is defined[3] in the following file.

```
xaifBooster/algorithms/CrossCountryInterface/inc/Elimination.hpp
```

The structural representation of G is a class called `LinearizedComputationalGraph`, or `LCG`. It uses vertex and edge classes `LCGVertex` and `LCGEdge`; their definitions can be found in the same directory in header files with the respective names. This graph class (like all other graph classes in `xaifBooster`) is based on the Boost Graph Library [16]. An edge elimination step such as the front elimination of edge $(1,3)$ shown in Fig. 1(e) yields the following two (because vertex 3 has two outedges) fused multiply-add operations: c_{41}+=c_{43}*c_{31} and c_{51}=c_{53}*c_{31}. They are represented as instances of the class `JacobianAccumulationExpression`, or `JAE`. Because we need only structural information, the nonminimal vertices of a `JAE` represent the `*` or `+` operations, and the minimal vertices have references either to edges of G or to maximal vertices of other `JAE` that are earlier elimination results, such as the new edge (1,5) in Fig. 1(e). An instance of `JAEList` and the remainder graph G^* (which, like the input graph G, is an instance of `LCG`) are the results of the top-level routine `Elimination::eliminate` called from within `xaifBooster`.

Aside from the purely structural representation of G, the instances of `LCGEdge` contain a discriminator to identify whether an edge label is ± 1, constant, or generally variable. This information is needed to solve the second problem. In the following we will discuss elimination heuristics implemented by *Angel* [17,18].

2.2 Elimination Heuristics

The top-level driver routine for the heuristics is `Elimination::eliminate()` implemented in `angel/src/xaif_interface.cpp`. An `Elimination` instance is instantiated from `xaifBooster`, and its attributes are set via `oadDriver` command line switches `a`,

[3] We follow the convention used in the OpenAD manual [3], Sec. 2.2, by referencing all files relative to the OpenAD install directory set as the `$OPENADROOT` environment variable. We provide generated documentation of the source of all OpenAD components under the website's [2] "Documentation" link.

`A`, `m`, `M`, and `R`; see [3], Sec. 4.1.3.4. The attributes determine which specific elimination routine is called by `eliminate`, for instance, `compute_elimination_sequence()`.[4]

Markowitz-Based Heuristics Using Markowitz (triggered by setting `-M 0`) as an example, we illustrate the implementation of a heuristic and **identify in red the minimal set of elements to be changed for a new heuristic**. Angel internally uses plain boost graphs. The first step in `compute_elimination_sequence()` is to convert the `LCG` given as input via a call to `read_graph_xaif_booster`. Then we declare a stack $\mathcal{F}$ of heuristics that filters the elimination target vertices down to a single vertex. For example, the first of three such $\mathcal{F}$ (for vertex elimination) is declared in `xaif_interface.cpp`, line 1151, as

```
typedef heuristic_pair_t<lowest_markowitz_vertex_t>, reverse_mode...> lm_rm_t;
```

and later, on line 1154, is defined as follows.

```
lm_rm_t lm_rm_v (lowest_markowitz_vertex, reverse_mode_vertex);
```

This filter stack internally first passes vertices with the lowest Markowitz degree and then uses the reverse mode order as a tie breaker. The Markowitz filter and the re-

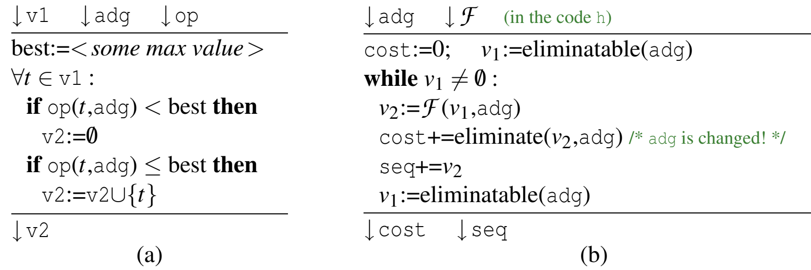

Fig. 3. Pseudo code for `standard_heuristic_op` (a) and for `use_heuristic` (b).

verse mode tie breaker are defined by using `standard_heuristic_op` and a function object [5] called `lmv_op_t`, defined in `angel/src/heuristics.cpp`. Figure 3(a) shows the pseudo code implemented in `standard_heuristic_op` taking in a vector of elimination targets `v1`, a graph `adg`, and the aforementioned function object as a formal parameter `op`. The core of the Markowitz heuristic is encapsulated in the function object `lmv_op_t`'s `operator()`, which returns the expected product of in and out degree.

```
in_degree(v,cg) * out_degree(v,cg)
```

With this framework, the implementation of a new heuristic requires comparatively little effort, and one has the remainder of OpenAD readily available to evaluate its efficacy.

Evaluating the Cost Now that we have described how a filter stack is implemented, we return to `compute_elimination_sequence()` to look at the computation and comparison of the resulting cost. The above-mentioned three $\mathcal{F}$ together with simple forward and reverse order are passed in a call to `best_heuristic` (line 1169), which com-

[4] We encourage using the "search" provided in the Angel and `xaifBooster` generated source code documentation under "Documentation" at [2].

[5] C++ classes with an `operator()`.

putes the elimination sequences and their respective cost individually and returns the cheapest elimination sequence. The definition is found in `heuristics_impl.hpp`. To determine the cost of each of the five $\mathcal{F}$ individually, it calls `use_heuristic`, defined in `heuristics.cpp`; see also Fig. 3(b). It takes as input the graph G as `adg` and the filter and returns the `cost` and the elimination sequence `seq`. The filter $\mathcal{F}$ determines the next elimination target, and the cost is accumulated as incurred by the actual elimination. For the latter, one can follow the calls from `eliminate` to `vertex_elimination` to `back_edge_elimination` to see that the cost is the number of edge label multiplications. Consequently a new heuristic for this cost model does not necessitate any further changes.

Edge Eliminations Returning once more to `compute_elimination_sequence()`, we also find a variety of *edge elimination* heuristic filters declared. The logic employed for these is analogous to that applied to vertex eliminations. The cost, following the same cost model used for vertex elimination, for the best edge elimination is compared to that of the best vertex elimination. The winner is converted into an equivalent face elimination [11] sequence. Because a face elimination can be considered the elemental building block for all elimination operations, there is a common method to populate the caller-provided instance of `JAEList` which expects a sequence of face eliminations as input. An actual face elimination sequence and accompanying heuristics can be found in `compute_elimination_sequence_lsa_face`.

2.3 Scarcity-Preserving Heuristics

To approximate a minimal representation $\boldsymbol{J}^*$ of the Jacobian $\boldsymbol{J}$ (problem 2), OpenAD has various scarcity-preserving heuristics [14]. We describe a simple example of such a heuristic that is implemented in `compute_partial_elimination_sequence`. Its execution is triggered by the `oadDriver` command line settings `-M 1 [ -m ]`. While the heuristics discussed in Sec. 2.2 minimize the count of elimination operations, the cost here is simply the number of nonunit edge labels in the remainder graph G^*. Minimizing the elimination-induced operations count is a secondary concern. Given the distinction of unit, constant, and variable edge labels in the `LCGEdge` class, we can disregard the multiplication of constant edge labels from the elimination operations cost because it is a compile time effort. All counters pertaining to the heuristic are kept in an instance of `elimSeq_cost_t` that is defined in `angel/include/angel_types.hpp`. The pseudo code in Fig. 4 illustrates the core logic. Here, for simplicity, we consider only edge elimination operations. Again we use a stack of filters to narrow the eligible targets down to a single edge. However, here we do so repeatedly because in each elimination sequence we may detect a refill[6] that in a subsequent elimination sequence we attempt to avoid (line 09), thereby modifying the set v_3 and arriving at different result. After each elimination is complete, we compare it to the current best result, where `elimSeq_cost_t` holds all relevant counters, such as the minimal edge count reached along the way, and the operations incurred. If no new refill dependence is detected, we are done. The easiest entry point to **change the behavior for a new heuristic** is to

[6] An edge (i,j) is *refilled* if it is eliminated but subsequently recreated as a consequence of eliminating edges in an alternative path from i to j.

```
↓ourLCG   (the graph G)
01 best:=< default instance of elimSeq_cost_t >
02 do
03     adg:=ourLCG;    curr:=< default instance of elimSeq_cost_t >
04     do
05         v1:=eliminatable(adg)    /* find eliminatable edges */
06         if v1 ≡ ∅ then break     /* this elimination is complete */
07 ||      v2:=reducing_edge_eliminations(v1,adg)    /* find edge count reductions */
08 ||      if v2 ≡ ∅ then v2 := v1    /* if such target edges don't exist use the previous target set */
09 ||      v3:=refill_avoiding_edge_eliminations(v2,adg)
10 ||      if v3 ≡ ∅ then v3 := v2
11 ||      v4:=lowestMarkowitzEdgeElim(v3,adg)
12 ||      v5:=reverseModeEdgeElim(v4,adg)
13         curr.elims+=v5
14         curr.cost+=eliminate(v5,adg) /* modifies adg */
15     if (curr < best) then best:=curr
16     if (! curr.revealedNewDependence) then break /* checking for new refill dependencies*/
17 /* code to extract partial elimination from 'best' until J* and populate jae_list and remainderLCG */
↓jae_list   ↓remainderLCG
```

Fig. 4. Pseudo code for `compute_partial_elimination_sequence`

modify the filter stack at any of the red-marked lines 07–12. To finish, we need to create G^* by replaying the best elimination sequence to the first point when the minimal edge count is reached. This also populates the instance of `JAEList`. Finally we populate the caller-provided instance of `LCG` with the contents of the Angel internal G^*. The propagation through the remainder graph is encapsulated entirely in `xaifBooster` as a code generation step. Note that the heuristic logic inside `reducing_edge_eliminations` is no longer as simple as, for instance, the Markowitz criterion because we need to **precompute the effect an elimination would have** on the edge count considering different combinations of unit/constant and variable edge labels. This precomputation is implemented in `edge_elim_effect`. Other implementations of scarcity-preserving heuristics can be found in `compute_partial_transformation_sequence` which includes logic to produce pre- and postrouting steps, and in the `_random` variants of the above, which include randomized choices and backtracking in heuristics rather than the simple greedy algorithm explained here.

3 Reversal Schemes

Section 1 introduces reversal schemes as a means to obtain the $\boldsymbol{J}_i$ in reverse order, potentially involving recomputation as a tradeoff for storage. In OpenAD the granularity of choice for making this tradeoff is the subroutine call.

3.1 Simple Split and Joint Modes

For a given scope the two extreme ends of the tradeoff are called *split* mode and *joint* mode. The former stores (or "tapes") all $\boldsymbol{J}_i$ at once; the latter minimizes the storage for

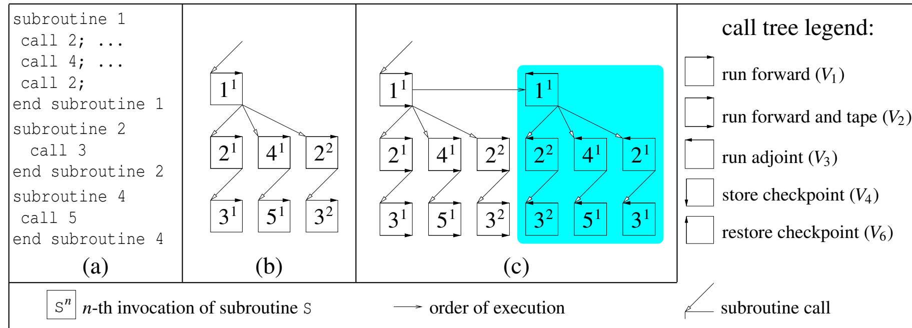

Fig. 5. Example code (a), corresponding call tree (b), split mode (c), and legend.

the J_i and the checkpoints. A split mode example is shown in Fig. 5. The adjoint phase, that is, the propagation $(\ldots(\boldsymbol{W}^T \circ \boldsymbol{J}_s) \circ \ldots \circ \boldsymbol{J}_1)$, is colored blue. The joint mode is characterized by the fact that we store the $\boldsymbol{J}_i$ for each subroutine only **immediately before** the corresponding adjoint sweep. A joint mode corresponding to Fig. 5(a) is shown in Fig. 6 where the pairs of storing the $\boldsymbol{J}_i$ followed by the adjoint are colored green and recomputations are framed in red. Both schemes exhibit a very regular structure in which each subroutine executes one of five specific variants (V_1-V_4,V_6) generated by OpenAD.[7] Rather than the extremes, in practice a hybrid reversal scheme is used, consisting in part of the split mode for as large a section of the model for which the corresponding $\boldsymbol{J}_i$ still fit into memory, while the higher-level parts use some checkpointing scheme and the joint mode. Because this is an application-dependent problem, we do not directly generate the entire reversal scheme in `xaifBooster` but rather **use a template and a postprocessing step to orchestrate the reversal**.

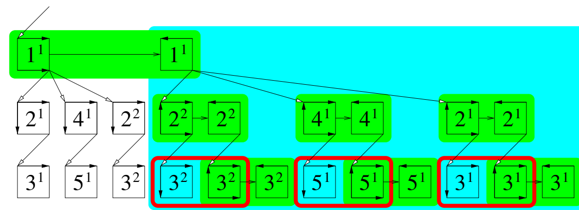

Fig. 6. Joint mode for Fig. 5(a).

3.2 Template Mechanism

An OpenAD template is best understood as a sample subroutine with some control flow into which at predefined spots the postprocessor inserts the variants V_i. While the xaifBooster transformation generates the V_i, the control structure together with some static state information determines *which* version V_i at any invocation point in the call tree is executed. As examples, the split and joint mode templates are shown in Fig. 7(a) and (b), respectively; the control structure is shown in blue. The numbering of the subroutine variants (V_1-V_4,V_6) directly reflects the `id` number referenced in the

[7] Other generated variants V_5, V_7 etc. are not of relevant for this paper.

```
1  subroutine template()
2    use OAD_tape
3    use OAD_rev
4  !$TEMPLATE_PRAGMA_DECLARATIONS
5    integer iaddr
6    external iaddr
7    if (our_rev_mode%plain) then
8    ! original function
9  !$PLACEHOLDER_PRAGMA$ id=1
10   end if
11   if (our_rev_mode%tape) then
12   ! taping
13 !$PLACEHOLDER_PRAGMA$ id=2
14   end if
15   if (our_rev_mode%adjoint) then
16   ! adjoint
17 !$PLACEHOLDER_PRAGMA$ id=3
18   end if
19 end subroutine template
```

(a)

```
1  subroutine template()
2    use OAD_tape
3    use OAD_rev
4    use OAD_cp
5    type(modeType) :: our_orig_mode
6    if (our_rev_mode%arg_store) then
7  !$PLACEHOLDER_PRAGMA$ id=4
8    end if
9    if (our_rev_mode%arg_restore) then
10 !$PLACEHOLDER_PRAGMA$ id=6
11   end if
12   if (our_rev_mode%plain) then
13     our_orig_mode=our_rev_mode
14     our_rev_mode%arg_store=.FALSE.
15 !$PLACEHOLDER_PRAGMA$ id=1
16     our_rev_mode=our_orig_mode
17   end if
19     call OAD_revStorePlain
20 !$PLACEHOLDER_PRAGMA$ id=2
21     call OAD_revAdjoint
22   end if
23   if (our_rev_mode%adjoint) then
24     call OAD_revRestoreTape
25 !$PLACEHOLDER_PRAGMA$ id=3
26     call OAD_revRestoreTape
27   end if
28 end subroutine template
```

(b)

Fig. 7. Split mode template (a); joint mode template (b). The definitions of the `OAD_rev*` routines can be found in `$OPENADROOT/runTimeSupport/simple/OAD_rev.f90`.

`!$PLACEHOLDER_PRAGMA$` (shown in red). The logic for the split mode is self-evident: the driver for the program sets the global `our_rev_mode` and invokes the top-level model routine once with `our_rev_mode%tape` and once with `our_rev_mode%adjoint` set to true. The joint mode logic in the template is more complicated, but one can easily see that the execution of each subroutine variant implies which variant should be executed for its callees. The callee variant is set through calls to the respective `OAD_rev*` routines (shown in Fig. 7(b) in green); and by consulting the joint scheme figure, one can easily verify the correctness of the template logic. In particular, we see that after storing the $\boldsymbol{J}_i$ for the given subroutine (line 20) the mode is set to adjoint (line 21), which then immediately follows (lines 23–27) as expected in the joint mode.

3.3 Reversal Scheme Using Revolve

In uniform time-stepping schemes one can assume that all checkpoints are of the same size, as are the storage requirements to produce the $\boldsymbol{J}_i$ of a single timestep. This can be used to derive an optimal reversal scheme [19] that minimizes the number of recomputations for a given total s of timesteps and permitted number p of checkpoints. Rather than a strict split or joint mode it indicates for each step, to the top-level time step routine and all its callees, which of the respective variants V_i should be executed. A Fortran version of optimal algorithm is available.[8] Figure 8(a) shows an example loop

[8] See `http://mercurial.mcs.anl.gov/ad/RevolveF9X`.

```
subroutine loopBody(x)
  double precision :: x
  x=sin(x)
end subroutine
```

```
!$openad XXX Template ad_revTempl.f
subroutine loopWrapper(x,s)
  double precision :: x
  integer :: s
!$openad INDEPENDENT(x)
  do i=1,s
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

```
program driver
  use OAD_active
  use OAD_tape
  implicit none
  external head
  type(active) :: x
  integer :: s
  call oad_tape_init()
  x%v=.5D0
  x%d=1.0D0
  write (*,fmt='(A)',advance='no') &
    'number of iterations = '
  read (*,*) s
  call loopWrapper(x,s)
  print *, 'driver running for x =',x%v
  print *, ' yields dy/dx =',x%d
end program driver
```

(a)

```
subroutine template()
  use OAD_tape
  use OAD_rev
  use OAD_cp
  use revolve
  LOGICAL :: ini=.FALSE.
  TYPE(rvAction) :: rvAct
  CHARACTER , DIMENSION(80) :: errorMsg
  integer :: p, curr=0
  write (*,fmt='(a)',advance='no') &
    'number of checkpoints = '
  read (*,*) p
  ini=rvInit(s,p,errorMsg)
  IF (.NOT.ini) WRITE(*,'(A,A)') &
    'Error: ', errorMsg
  do while (rvAct%actionFlag/=rvDone)
    rvAct=rvNextAction()
    select case (rvAct%actionFlag)
    case (rvStore)
      call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
      call cp_close
    case (rvRestore)
      call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
      curr=rvAct%iteration
      call cp_close
    case (rvForward)
      call OAD_revPlain
      do curr=curr,rvAct%iteration−1
        call loopBody(x)
      end do
    case (rvFirstUTurn,rvUTurn)
      call OAD_revTape
      call loopBody(x)
      call OAD_revAdjoint
      call loopBody(x)
     end select
  end do
end subroutine template
```

(b)

Fig. 8. Time-stepping example loop in `loopWrapper` with the loop body encapsulated in `loopBody` and a driver (a); `ad_revTempl.f` to be applied to `loopWrapper` (b).

code with adjustments needed to apply OpenAD shown in brown; see also [3], Sec. 1.3. Figure 8(b) shows the template to be applied to `loopWrapper` and used in conjunction with revolve. The key ingredients of the template are the loop (line 16) replacing the original time-stepping loop (line 6) in `loopWrapper`. All actions are determined by calling `rvNextAction` (line 17). We distinguish storing and restoring checkpoints to file by injecting the subroutine variants V_4 (line 21) and V_6 (line 25), respectively, computing forward (lines 30–32) up to a step determined by revolve, and doing split adjoint computation (lines 34–37) for `rvFirstUTurn` and `rvUTurn`. In the latter the `loopBody` is directly injected (lines 35,37) because in the template in Fig. 8(b) we explicitly replace the entire loop construct of `loopWrapper` (lines 6–8). Consequently, for the template mechanism it is important to have the time-stepping loop separated in a wrapper, as done in our example. While the above represents the solution to a very regular setup, one can use the same idea to **apply new heuristics in cases where the problem is**

combinatorial, for example, when the timesteps (and therefore the associated storage requirements) are not homogeneous. We note that the template mechanism as the entry point to the overall reversal is well insulated from the remainder of the OpenAD tool chain and provides an easy access to experiment with other reversal schemes.

Acknowledgements This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy under Contract DE-AC02-06CH11357.

References

1. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. 2nd edn. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA (2008)
2. OpenAD website: downloads, manual, links. http://www.mcs.anl.gov/openad
3. Utke, J., Naumann, U.: OpenAD/F: User Manual. Technical report, Argonne National Laboratory (2009) latest version available online at http://www.mcs.anl.gov/OpenAD/openad.pdf.
4. Bücker, H.M., Corliss, G.F., Hovland, P.D., Naumann, U., Norris, B., eds.: Automatic Differentiation: Applications, Theory, and Implementations. Volume 50 of Lecture Notes in Computational Science and Engineering. Springer, New York (2005)
5. Bischof, C.H., Bücker, H.M., Hovland, P.D., Naumann, U., Utke, J., eds.: Advances in Automatic Differentiation. Volume 64 of Lecture Notes in Computational Science and Engineering. Springer, Berlin (2008)
6. AD community website: news, tools collection, bibliography. http://www.autodiff.org/
7. Hovland, P., Naumann, U., Walther, A.: Combinatorial problems in automatic differentiation. In Naumann, U., Schenk, O., Simon, H., Toledo, S., eds.: Combinatorial Scientific Computing. Dagstuhl Seminar Proceedings, Dagstuhl, Germany, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2009)
8. Naumann, U., Hu, Y.: Optimal vertex elimination in single-expression-use graphs. ACM Transactions on Mathematical Software **35** (2008)
9. Utke, J.: Flattening basic blocks. [4] 121–133
10. Baur, W., Strassen, V.: The complexity of partial derivatives. Theoretical Computer Science **22** (1983) 317–330
11. Naumann, U.: Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. Mathematical Programming, Ser. A **99** (2004) 399–421
12. Naumann, U.: Optimal Jacobian accumulation is NP-complete. Math. Prog. **112** (2006) 427–441
13. Griewank, A.: A mathematical view of automatic differentiation. In: Acta Numerica. Volume 12. Cambridge University Press (2003) 321–398
14. Lyons, A., Utke, J.: On the practical exploitation of scarsity. [5] 103–114
15. Naumann, U.: Call tree reversal is NP-complete. [5] 13–22
16. Boost C++ Libraries website: downloads, documentation, news. http://www.autodiff.org/
17. AD nested graph elimination library (angel) website: downloads, overview. http://angellib.sourceforge.net

18. Naumann, U., Gottschling, P.: Simulated annealing for optimal pivot selection in Jacobian accumulation. In Albrecht, A., Steinhöfel, K., eds.: Stochastic Algorithms: Foundations and Applications. Volume 2827 of Lecture Notes in Computer Science., Springer (2003) 83–97
19. Griewank, A., Walther, A.: Algorithm 799: Revolve: An implementation of checkpoint for the reverse or adjoint mode of computational differentiation. ACM Transactions on Mathematical Software **26** (2000) 19–45 Also appeared as Technical University of Dresden, Technical Report IOKOMO-04-1997.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.