A Mixed Approach to Adjoint Computation with Algorithmic Differentiation

Kshitij Kulshreshtha, Sri Narayanan, Tim Albring

A mixed approach to adjoint computation with algorithmic differentiation

Kshitij Kulshreshtha[1], Sri Hari Krishna Narayanan[2], and Tim Albring[3]

[1] Universität Paderborn, Paderborn, Germany
[2] Argonne National Laboratory, Argonne, IL, USA
[3] Technische Universität Kaiserslautern, Kaiserslautern, Germany

Abstract. Various algorithmic differentiation tools have been developed and applied to large-scale simulation software for physical phenomena. Until now, two strictly disconnected approaches have been used to implement algorithmic differentiation (AD), namely, source transformation and operator overloading. This separation was motivated by different features of the programming languages such as Fortran and C++. In this work we have for the first time combined the two approaches to implement AD for C++ codes. Source transformation is used for core routines that are repetitive, where the transformed source can be optimized much better by modern compilers, and operator overloading is used to interconnect at the upper level, where source transformation is not possible because of complex language constructs of C++. We have also devised a method to apply the mixed approach in the same application semi-automatically. We demonstrate the benefit of this approach using some real-world applications.

Keywords: algorithmic differentiation, adjoint computation

1 Introduction

Solution techniques for optimal control and optimal design problems rely on the correct and efficient computation of the adjoint state. Various analytical and numerical techniques have been devised to compute these derivatives in the past. One of the emerging techniques for the computation of derivatives on modern computers is algorithmic differentiation (AD) [**?**]. Despite differentiation being a badly conditioned operation in general, research has shown [**?**] that the process of algorithmic differentiation is well behaved and the derivatives obtained are accurate to within round-off errors. This situation is in contrast to numerical derivatives computed by using finite-differencing techniques, where the difference step size is of critical importance.

Algorithmic differentiation assumes that functions are evaluated by using a finitely terminating evaluation procedure consisting of simple arithmetic operations $\{+, -, /, *\}$ and elementary function evaluations $\{\sqrt{}, \sin, \cos, \exp, \log, \ldots\}$. Since the analytical derivatives of such arithmetic operations and elementary functions are well known, these can be introduced in the evaluation procedure

almost mechanically, and the chain rule of differentiation can be applied to propagate the derivatives from one variable to another in the evaluation. In [**?**] various modes of propagation of derivatives as well as methods to implement tools are discussed in great detail. Here we present two techniques of AD, namely, source transformation and operator overloading.

Source transformation: Source transformation AD tools generate a new source code that computes the derivatives of an input source code. The output code must be compiled and executed in order to compute the derivatives. Tools such as ADIFOR [**?**], Tapenade [**?**], and OpenAD [**?**] can be used to generate derivative code for functions written in Fortran. Tapenade and ADIC [**?**] are examples of source transformation AD tools for C. In this work, we use ADIC to differentiate input source code portions written in C. In the output code, active variables are declared as objects of `DERIV_TYPE`, and runtime functions are used to propagate derivatives between them. When the output code is compiled with an appropriate driver and runtime library provided by ADIC, the Jacobian matrix can be computed.

Because such tools perform source code analysis, they can identify algorithmically active and passive variables and portions of the code. Furthermore, compilers can optimize the output code, resulting in high performance. However, no tool can generate derivative code for complete C++ input. C++ contains features such as polymorphism, inheritance, and templates that cannot be resolved statically, precluding the generation of correct derivative code.

Operator overloading: In an object-oriented language such as C++ the concept of operator overloading is well known. Several tools have been developed in recent years for AD using C++ operator overloading. ADOL-C [**?**] is a well-known open source AD tool with many features and high flexibility and has been successfully used to compute derivatives in a large number of simulation codes. The most important manual change required for using ADOL-C in any simulation is to convert the datatype of the real values to the special datatype `adouble` defined in the ADOL-C library. All operations executed after a call to `trace_on()` and before a call to `trace_off()` are recorded in an internal representation called the *trace*. Before the actual computation takes place, the independent variables are marked by assigning them values using the special `<<=` operator. Similarly the final dependent variables are marked by extracting their values using the special `>>=` operator. The trace can then be used in any mode of AD (i.e., forward or reverse) in order to compute first or higher derivatives. Several easy-to-use drivers for computing the derivative information from the trace are available. The most-used drivers are `gradient()`, `jacobian()`, and `hessian()`. For further usage details see [**?**].

The creation of the trace is the most crucial part of the whole program; and depending on the complexity of the functions being traced, the trace can become large and thus has the most impact on the memory consumption of the program. Where the trace does not fit into a prescribed amount of memory (RAM), it spills over automatically to the disk as trace files, thereby reducing the performance of

the implementation severely. Past attempts at reducing the memory requirement for certain applications include using checkpointing strategies [**?**,**?**]. For a number of problems, however, checkpointing is not applicable.

We propose a mixed approach that uses both operator overloading and source transformation to differentiate an input code that is largely written in C++ but whose computationally intensive portions are written in a C-like manner. Our approach employs operator overloading for most of the application and source transformation for the C-like portions. Because the computationally intensive portions contribute most to the trace, using source transformation instead for these portions leads to a smaller trace and better performance. We have made changes to both ADIC and ADOL-C and written a preprocessor that enables the approach to be semi-automated. The rest of the paper is organized as follows. Section 2 presents the details of the mixed approach. Section 3 presents experimental results on two applications and, Section 4 discusses future work.

2 Mixed approach

The process of converting an ADOL-C instrumented application to use ADIC in certain parts is the following: (1) The user identifies a computationally intensive and C-like function from the input based on performance analysis or experience. (2) This function must be treated as an externally differentiated function (EDF) by ADOL-C. For this purpose, annotations are added to the input to support extraction of the EDF and its callees for differentiation by ADIC. Additional annotations are used to generate wrappers functions and files to copy data between ADOL-C data structures and the EDF. (3) ADIC is used to differentiate the EDF and provide forward- and reverse-mode differentiated code for it. (4) The EDF input, output, wrapper files, and original ADOL-C code are then built together. The rest of this section elaborates on the concepts of the EDF and the changes we made to ADOL-C and ADIC to support the mixed approach.

Externally differentiated functions in ADOL-C: The individual arithmetic operations and mathematical function evaluations of an EDF are not recorded on the ADOL-C trace. Instead the actual implementation of the differentiated EDF is provided via user defined function pointers that implement a certain predefined signature. As one can see in Fig. 1 the EDF replaces a large part of the trace by repeated calls to itself, which reduces the size of the trace. When ADOL-C processes the trace and arrives at a call to the EDF, ADOL-C calls the corresponding user-provided forward mode- or reverse mode- derivative code to obtain the derivatives.

ADOL-C previously maintained the EDF interface using a special structure `struct ext_diff_fct` that is registered to the ADOL-C core on a per function basis. Implementations for the forward- and reverse-mode first-order derivative computations are set up in this structure as function pointers that have a particular signature as defined in the header file `<adolc/externfcts.h>`. The limitation of this interface is that it expects all inputs as well as all outputs to the EDF to be passed as two contiguously allocated arrays.

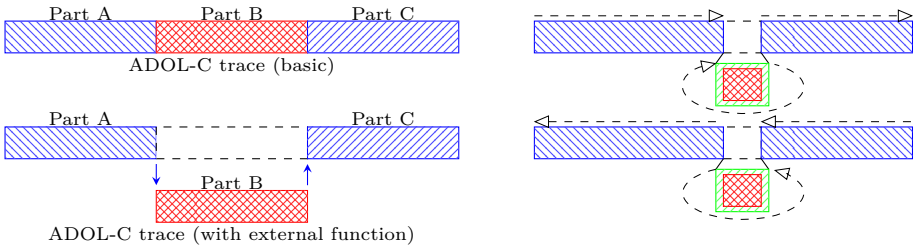
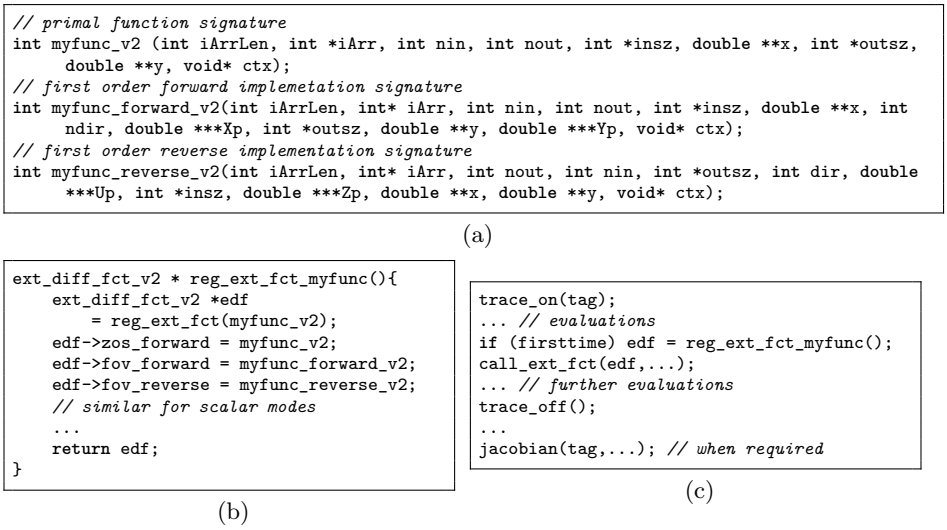


Fig. 1. ADOL-C trace of a simple and externally differentiated function (left). Repeated evaluation of an external function in forward and reverse mode (right)

The design of the `adouble` type in ADOL-C creates an internal representation of the executed code at runtime. In order to do so most efficiently, `adouble` objects are allocated in a memory pool whereever there is unused space. Unless the pool is exhausted, new memory is not allocated. This design makes the allocation of large contiguous arrays an expensive operation, because of the need for finding a suitable chunk of unused space in the memory pool. Several smaller contiguous arrays, on the other hand, can be allocated more easily. Therefore we designed a second version of the EDF interface structure `struct ext_diff_fct_v2` that supports providing several input arrays and several output arrays, each not necessarily of the same size. We also added extra integer-valued input parameters and an opaque object-valued input/output parameter that do not have an effect on the differentiation process outside the EDF. These changes required adjusting the signatures of the forward- and reverse-mode implementations for the EDF. The signatures now contain the number of input and output vectors, the sizes of each of these, their values, the corresponding tangents or adjoints, extra integer-valued input arrays, and an opaque context object if needed (see Fig. 2(a)). However, the process of registration and setup of the function pointers stays the same as in the original EDF interface and can even be encapsulated in a separate routine (see Fig. 2(b)), which is called once before the function is required to be evaluated. The ADOL-C evaluation of the complete structure would then look something like the code in Fig. 2(c). The user-provided functions `edf->fov_forward()` or `edf->fov_reverse()` are called during the evaluation of the `jacobian()` at the appropriate point.

Runtime support for ADIC generated code: To support the mixed approach's use of forward-and-reverse mode AD in a single execution instance, we recoded ADIC's runtime library in C++ and used namespaces to separate forward- and reverse-mode derivative manipulation routines. The namespace usage is inserted into the ADIC-generated code by using simple postprocessing scripts. The `DERIV_TYPE` structure was rewritten to be a class that supports both dynamic and static allocation of the `grad` array within `DERIV_TYPE`. Because dynamic allocation for every `DERIV_TYPE` object can be expensive, we created a memory manager that allocates a large amount of memory from the heap and then allocates the



```
// primal function signature
int myfunc_v2 (int iArrLen, int *iArr, int nin, int nout, int *insz, double **x, int *outsz,
     double **y, void* ctx);
// first order forward implemetation signature
int myfunc_forward_v2(int iArrLen, int* iArr, int nin, int nout, int *insz, double **x, int
     ndir, double ***Xp, int *outsz, double **y, double ***Yp, void* ctx);
// first order reverse implementation signature
int myfunc_reverse_v2(int iArrLen, int* iArr, int nout, int nin, int *outsz, int dir, double
     ***Up, int *insz, double ***Zp, double **x, double **y, void* ctx);
```

(a)

```
ext_diff_fct_v2 * reg_ext_fct_myfunc(){
    ext_diff_fct_v2 *edf
        = reg_ext_fct(myfunc_v2);
    edf->zos_forward = myfunc_v2;
    edf->fov_forward = myfunc_forward_v2;
    edf->fov_reverse = myfunc_reverse_v2;
    // similar for scalar modes
    ...
    return edf;
}
```

(b)

```
trace_on(tag);
... // evaluations
if (firsttime) edf = reg_ext_fct_myfunc();
call_ext_fct(edf,...);
... // further evaluations
trace_off();
...
jacobian(tag,...); // when required
```

(c)

Fig. 2. (a) Signatures of the forward- and reverse-mode wrapper routines; (b) per-routine registration of EDF; (c) calling an EDF in ADOL-C instrumented code

`grad` array of an object from this pool. We matched ADIC's layout of `grad` array to ADOL-C's layout of tangents and adjoints for the input and output vectors. Therefore only pointers are copied, and ADIC reuses memory already allocated in ADOL-C.

User annotations and preprocessing: User annotations have two purposes: First, they identify an EDF and its callees for extraction and subsequent differentiation by ADIC. The annotations surround the EDF and its callees, as shown in Fig. 3(a). The extraction of code is necessary because ADIC requires the C code to be isolated from the C++ code that it does not differentiate. Additional user editing may be required to obtain code that is appropriate for differentiation by ADIC. Second, annotations are used to generate the interface code for arguments of the EDF. The annotation identifies inputs, outputs, and their respective sizes or extra integers required for the computation, as well as the position of each formal parameter in the argument list of the EDF. This information helps generate wrapper code to transfer data between ADOL-C data structures and ADIC-generated code. These annotations are written directly as Python tuples, as seen in Fig. 3(b). Each tuple contains the name of the formal argument, followed by its size and the position in the formal argument list. The size itself is a list of length 0, 1, or 2, depending on whether the argument represents a scalar, a vector, or a matrix. Integer arguments are always scalars. The size may also contain references to values stored in the integers list. Several interface definitions, and thus multiple EDF structures, may also be used in any application.

```
/*@ declare doubletype=adouble @*/
 // since ADIC doesn't know adouble

/*@ begin adic_extract global @*/
... // global defines, variables etc.
    // required by ADIC routines
/*@ end adic_extract @*/

/*@ begin adic_extract @*/
... // lower level computational routines
    // differentiated by ADIC
/*@ end adic_extract @*/

/*@ begin adic_extract replace=rk_iter
    type=void @*/
... // top level interface routine
    // differentiated by ADIC
/*@ end adic_extract @*/
```

(a)

```
void rk_iter(double h, adouble *y, adouble
     **k, adouble *rhs, int n, adouble *u,
      int m, adouble *yt, adouble *ynew)
{
/*@ begin adic_export interface
name = 'rk_iter'
iarr = [ ('n',5), ('m',7) ]
input = [ ('h', [], 1),
         ('y', ['2*nDe+5'], 2),
         ('u', ['5'],6) ]
output = [ ('k', ['stage','2*nDe+5'], 3),
           ('rhs', ['2*nDe+5'], 4),
           ('yt', ['2*nDe+5'], 8),
           ('ynew',['2*nDe+5'], 9)]
@*/
... // original ADOL-C computation code
/*@ end adic_export @*/
}
```

(b)

Fig. 3. (a) Annotations for extracting code for ADIC processing; (b) annotations describing the interface routine to generate wrapper code

Table 1. Sizes of trace files created on disk in a purely ADOL-C implementation of periodic adsorbtion process that are not present in a mixed approach

N_{space} \ N_{time}	2000	3000	5000
20	576 MB	863 MB	2511 MB
30	856 MB	2240 MB	3734 MB
50	2472 MB	3707 MB	7146 MB

3 Applications

We have tested the mixed approach on two applications. The following describes each application briefly and provides the results obtained by using the mixed approach.

Periodic adsorption process: The periodic adsorption process was studied from an optimization point of view in [**?**,**?**]. A system of PDAEs in time and space with periodic boundary conditions models the cyclic steady state of a process, where a fluid is preferentially absorbed on the surface of a sorbent bed. This leads to dense Jacobians that dominate the computation time (see [**?**]). Therefore, previous works have used inexact Jacobians (for example, [**?**]). Using AD, however, we compute the equality and inequality constraint Jacobians as well as the objective gradient exactly up to machine precision.

The PDAE system is discretized in space by using a finite-volume approach, and the resulting system of ODEs is then integrated in time by using a Runge-Kutta method. This Runge-Kutta iteration in the implementation was determined to be a suitable EDF for differentiation by ADIC, particularly because

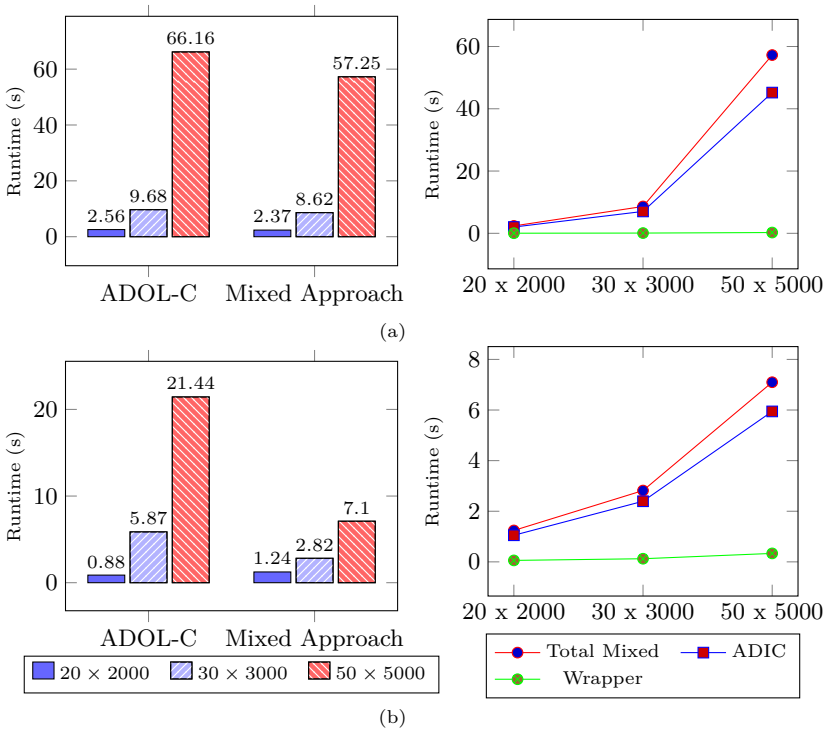


Fig. 4. Time required (in seconds) to compute (a) Jacobian of equality constraints in forward mode; (b) Jacobian of inequality constraints and gradient of objective in reverse mode

this routine is repeatedly called at each time step of the simulation and has a C-like implementation. The annotations for declaring this interface routine are shown in Fig. 3(b). Two other lower-level routines for computing the right-hand side of the ODE system are also processed by ADIC. The overall problem size depends on the spatial and temporal discretization (N_{space} and N_{time}). In Table 1 we show the memory required by the trace files created on disk in a purely ADOL-C implementation for various problem sizes, which are absent in the mixed approach. Both approaches preallocate memory of size 2.3 GB in all cases. The absence of trace files on disk in the mixed approach shows that the trace was small enough in all cases to fit into the preallocated memory. Additionally, the runtimes of the mixed approach show improvement over a pure ADOL-C implementation. In Fig. 4(a) the runtimes required in the computation of a equality constraint Jacobian with forward mode are plotted in the left figure for certain problem sizes. In the right side is the time required in the mixed approach is divided into the time required in the wrapper code of the EDF and the ADIC-processed part of the EDF. The same runtimes for the computation

of the inequality constraint Jacobian and the objective gradient are shown in Fig. 4(b).

Fluid dynamics – airfoil simulation: Recently, AD was successfully applied to the open source multiphysics suite SU2 [**?**], which uses a highly modular C++ code structure, to design an efficient adjoint solver [**?**] for optimization. The implementation is based on the fixed-point formulation of the underlying solver and requires only the recording of one iteration using the converged flow solution. Therefore, at least for steady-state problems, the necessity for checkpointing is eliminated. Still, because of the nature of operator overloading, the memory requirements increase by approximately a factor of 10 compared with the direct flow solver.

SU2 is based on a finite-volume method and offers several well-established combinations of spatial and temporal methods for discretizing the flow equations. Either the steady Euler or the Navier-Stokes equation can be used as the physical model. For this work we have used a second-order central discretization plus an artificial dissipation term (Jameson-Schmidt-Turkel scheme, JST) for the convective terms and a least-squares method for evaluating the gradients needed for the viscous terms. The explicit Euler method is used to advance in pseudo-time until convergence. The following two routines were identified as promising use cases for the mixed approach:

1. `CCentJST_Flow::ComputeResidual(su2double*val_residual)`:
   per edge convective residual, projects convective flux on the cell-face normal.
2. `CEulerSolver::SetPrimitive_Gradient_LS(CGeometry *geometry)`:
   per node gradient of non-conservative variables using least-squares (only Navier-Stokes).

Both routines contain mainly C-like code, which can be processed by ADIC. A potential drawback, however, is that they use mainly class member variables as input. Another difficulty is posed by calls of routines that return variables from other class objects. In such cases we manually copy the data back and forth into simple arrays and define interface routines that take extra inputs.

Figure 5(a) shows the runtime and memory requirements for the Euler solver with a 2D airfoil in transonic flow with 10,216 elements. While the time for tracing is clearly reduced, the evaluation time has significantly increased. This indicates that ADIC-generated derivative code is slower for that case compared with ADOL-C. However there is a decrease of disk usage, solely due to trace files, and an increase in the used RAM. For a 3D airfoil with 582,752 elements the workload for each element is much higher. In that case the total runtime for tracing plus evaluation decreases in the mixed approach as shown in Fig. 5(b). Still, a large fraction of the evaluation time is in the ADIC-generated code. Disk usage reduces by approximately 10% while the used RAM increases insignificantly. For the Navier-Stokes solver with a 2D airfoil with 13937 elements, as shown in Fig. 5(c), the time for tracing reduces by 36%, and the evaluation time increases slightly, resulting in a total reduction by 16%. Furthermore, the total memory usage decreases by 15%.

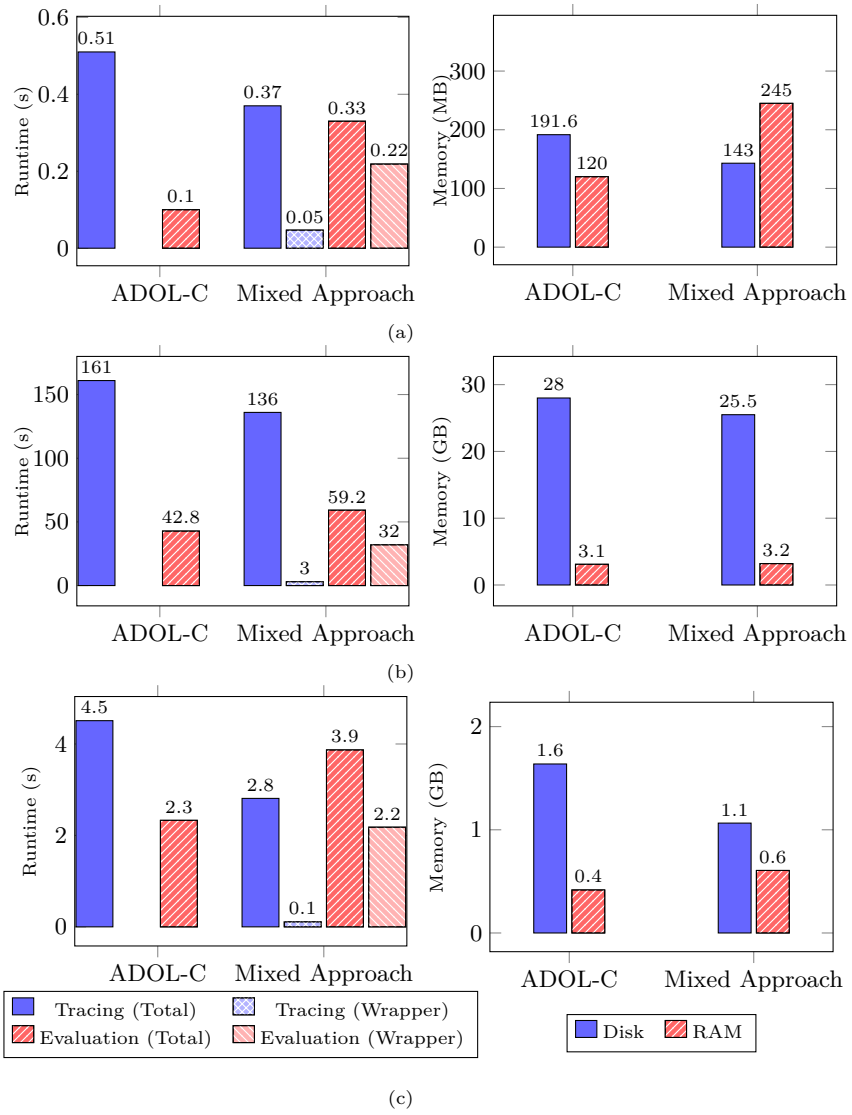


Fig. 5. Runtime and memory requirements for a 2D Euler case (a), 3D Euler case (b) and 2D Navier-Stokes case (c).

4 Conclusions and future work

We have implemented a mixed approach to AD that uses the operator overloading approach to differentiate most of an application and source transformation to differentiate just the computationally intensive portions. The user identifies these portions to be processed by ADIC and uses annotations and a preprocessor

to generate code that interfaces ADOL-C's internal data structures with ADIC generated code. The mixed approach has been applied successfully to medium-sized and large-sized applications, resulting in lower memory usage. We plan to apply the mixed approach to more applications. We will also study the benefit of differentiating most of an application using source transformation and only some C++ portions using ADOL-C.

Acknowledgments. This work was funded in part by a grant from DAAD Project Based Personnel Exchange Programme and by a grant from the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.